

Ekler Péter – Fehér Marcell – Forstner Bertalan – Kelényi Imre

Android-alapú szoftverfejlesztés

Az Android rendszer programozásának bemutatása



Ekler Péter - Fehér Marcell
Forstner Bertalan - Kelényi Imre

Android-alapú szoftverfejlesztés

Az Android rendszer programozásának bemutatása



2012

Android-alapú szoftverfejlesztés

Az Android rendszer programozásának bemutatása

Ekler Péter – Fehér Marcell – Forstner Bertalan – Kelényi Imre

Alkalmazott informatika sorozat

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Alkalmazott Informatika Csoport

© Ekler Péter, Fehér Marcell, Forstner Bertalan, Kelényi Imre, 2012.

Sorozatszerkesztő: Charaf Hassan

Lektor: Csorba Kristóf

Felelős kiadó: a SZAK Kiadó Kft. ügyvezetője

Felelős szerkesztő: Kis Ádám

Olvasószerkesztő: Laczkó Krisztina

Tördelő: Bukovics Zoltán

Borítóterv: Takács Dorottya és Takács Lídia

Grafikai munka kivitelezője: Flórián Gábor (Typoézis Kft.)

Terjedelem 25 (B5) ív.

Készült az OOK Press Nyomdában (Veszprém)

Felelős vezető: Szathmáry Attila

ISBN 978-963-9863-27-9

ISSN 1785-363X

A szöveg helyességét és az elválasztásokat a MorphoLogic Helyesek nevű programjával ellenőriztük.

Minden jog fenntartva. Jelen könyvet, illetve annak részeit a kiadó engedélye nélkül tilos reprodukálni, adatrögzítő rendszerben tárolni, bármilyen formában vagy eszközzel elektronikus úton vagy más módon közölni.



SZAK Kiadó Kft. ■ Az 1795-ben alapított Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja ■ 2060 Bicske, Diófa u. 3.
■ Tel.: 36-22-350-209 ■ Fax: 36-22-565-311 ■ www.szak.hu ■
e-mail: info@szak.hu ■ <http://www.facebook.com/szakkiado> ■
Kiadóvezető: Kis Ádám, e-mail: adam.kis@szak.hu ■ Főszerkesztő: Kis Balázs, e-mail: balazs.kis@szak.hu

DÉKÁNI ELŐSZÓ

A Budapesti Műszaki és Gazdaságtudományi Egyetem fennállása óta kiemelt figyelmet fordít arra, hogy minden téren támogassa a technológia fejlődését, az újdonságok mielőbbi beépítését az oktatási portfóliójába. A Villamosmérnöki és Informatikai Kar feladata ebben a kérdésben talán még nehezebb, figyelembe véve az informatika világának rohamos fejlődését.

A mobiltelefonok megjelenése tovább fokozta ezt a tempót, hiszen ezek az eszközök még szélesebb felhasználói réteg számára tették elérhetővé az újdonságokat. A mobilkészülékek fejlődése mellett a mobileszközökre szánt operációs rendszerek is rohamos mértékben változtak, megjelentek az úgynevezett okostelefonok, nyitottabbá váltak a platformok, és a mobiltelefonokra történő szoftverfejlesztés minden informatikus számára elérhetővé vált. A mobileszközök programozásának népszerűsége miatt a mobilalapú megoldások és az alkalmazások száma drasztikusan emelkedni kezdett. Ezt felismerve a nagy informatikai cégek is sorra felsorakoztak az iparág mögé, és olyan verseny alakult ki, amely egy dinamikusan változó és rendkívül gyorsan fejlődő informatikai ágazatot hozott létre.

Napjaink egyik legnépszerűbb mobil-operációsrendszere a Google gondozásában levő Android platform, amely számos technológiai újdonságot vonultat fel, és meghatározó szerepe van a mobilüzletágban. Jelen könyv részletesen bemutatja az Android platformra való fejlesztést, és a gyakorlati példáknak köszönhetően az olvasó valóban hasznosítható tudásra tehet szert.

A könyv szerzői a BME Villamosmérnöki és Informatikai Kar Automatizálási és Alkalmazott Informatika Tanszékén belül a mobilkutatócsoport tagjaiként úttörő munkát végeznek ezen a területen. Több mint tíz éve folyamatosan nyomon követik a mobiltechnológia fejlődését, és aktívan részt vesznek ipari kutatási és fejlesztési feladatokban is, így gyakorlati tapasztalatokkal felvértezve készítették el könyvüket, amely ezáltal garanciát biztosít a szakmai tartalom minőségére.

Budapest, 2012. június

Dr. Vajta László
dékán

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

TARTALOMJEGYZÉK

1. Az Android platform bemutatása (Ekler Péter)	1
1.1. Az Android sikerességének okai	1
1.2. Az Android platform története	3
1.3. Android-verziók	5
1.4. Android Market (Google Play)	13
1.5. A platform szerkezete	14
1.5.1. Az <i>apk</i> állomány felépítése	15
1.5.2. A platform jellemzői és a fordítás mechanizmusa	17
1.6. A fejlesztőkörnyezet bemutatása	19
1.6.1. Telepítés	19
1.6.2. A fejlesztőkörnyezet használata	22
2. Az Android-alkalmazások felépítése (Ekler Péter)	27
2.1. Android-alkalmazás-környezet	27
2.2. Az Android-alkalmazás komponensei	28
2.2.1. Activity	29
2.2.2. Service	30
2.2.3. <i>ContentProvider</i> komponens	32
2.2.4. <i>BroadcastReceiver</i> komponens	32
2.3. Az Android-alkalmazás felépítése fejlesztői szemszögből	33
2.3.1. A <i>manifest</i> állomány bemutatása	33
2.3.2. Erőforrás-állományok	36
2.3.3. Forráskód	39
2.3.3.1. Kivételkezelés	39
2.3.3.2. Finalizerek kerülése	41
2.3.3.3. Importok kezelése	42
2.3.3.4. Kódkommentezés: <i>JavaDoc</i>	42
2.3.3.5. A kód szerkezete	44
2.3.3.6. Annotációk használata	47
2.3.3.7. Naplózás	48
2.4. Activity-életciklus és -környezet	49
2.5. Több Activity kezelése egy alkalmazásban	55

2.6. Az első Android-alkalmazás	58
2.7. Gyakorlófeladat	62
2.7.1. Activity indítása	62
2.7.2. Activity megjelenítése felugró ablakban.....	63
2.7.3. Életciklusfüggvények nyomonkövetése	64
2.7.4. Alkalmazásikon lecserélése	64
3. Felhasználói felület tervezése és készítése (Ekler Péter).....	65
3.1. Különböző méretű és felbontású képernyők kezelése.....	65
3.2. Android-layoutok	73
3.3. Android UI-vezérlők	82
3.4. Menük készítése erőforrásból	90
3.5. Animációk készítése	92
3.6. Stílusok és témák	96
3.6.1. Stílusok készítése	96
3.6.2. Témák készítése	99
3.7. Lokalizáció támogatása	101
3.8. További Android alkalmazáskomponensek.....	102
3.8.1. Élő háttérkép.....	102
3.8.2. Widget	106
3.9. Összetettlista-alapú alkalmazás készítése	110
4. Komponensek közti kommunikáció (Fehér Marcell)	129
4.1. Az Intent fogalma	130
4.2. Intent felépítése.....	132
4.3. Activity indítása	138
4.3.1. Explicit Intent	139
4.3.2. Implicit Intent	139
4.4. Activity visszatérési értéke	141
4.5. Intent-szűrők	144
4.5.1. Intent-feloldás	146

4.6. További Intent-lehetőségek.....	147
4.6.1. <i>PendingIntent</i>	147
4.6.2. <i>Linkify</i>	148
4.6.3. Intent lekérése, delegálása, a feloldás előzetes eredménye....	149
4.6.4. Google-alkalmazások implicit Intentjei	150
4.7. Rendszerszintű események	151
4.7.1. <i>Broadcast</i> esemény generálása	151
4.7.2. Feliratkozás <i>broadcast</i> eseményre	151
4.7.3. <i>BroadcastReceiver</i> regisztrálása	152
4.7.4. Android <i>broadcast</i> eseményei.....	153
5. A perzisztens adattárolás eszközei (Forstner Bertalan).....	155
5.1. Alacsony szintű fájlkezelés	155
5.1.1. A privát tárterület használata	156
5.1.1.1. Fájlok írása és olvasása	156
5.1.1.2. Gyorsítótárzás	157
5.1.1.3. Feltelepített, csak olvasható nyers adatfájlok elérése....	157
5.1.2. A nyilvános lemezterület használata	158
5.2. Beállítások tárolása a <i>SharedPreferences</i> segítségével	160
5.2.1. Kulcs-érték párok tárolása	160
5.2.2. A <i>Preferences</i> keretrendszer	162
5.2.2.1. A keretrendszer célja	162
5.2.2.2. A beállításokat leíró XML-erőforrás	163
5.2.2.3. A beállításokhoz tartozó Activity.....	165
5.3. Példányszintű adatok elmentése	166
6. Strukturált adatok tárolása (Forstner Bertalan).....	169
6.1. Az SQLite-adatbázismotor	169
6.2. Az adatbázis-elérés általános menete	170
6.3. Az adatbáziskezelő használata	170
6.4. Teendőelemek tárolása adatbázisban	171
6.5. A <i>TodoAdapter</i> átalakítása	177
6.6. A vezérlőlogika átalakítása.....	179

7. Pozíciómeghatározás és térképkezelés (Ekler Péter).....	181
7.1. A helymeghatározás módszerei mobil eszközökön	181
7.1.1. Wifialapú helymeghatározás	182
7.1.2. Cellaalapú helymeghatározás	183
7.1.3. GPS-alapú helymeghatározás	183
7.2. Cella- és hálózati információk lekérdezése	184
7.3. Pozíciókezelés Android platformon.....	192
7.3.1. Pozíciómeghatározás	192
7.3.2. Közelségi riasztások kezelése.....	198
7.3.3. Átalakítás földrajzi koordináta és postacím között.....	200
7.4. Térképnézet	202
8. A hálózati kommunikáció lehetőségei (Ekler Péter).....	213
8.1. Hálózati kapcsolatok felügyelete	213
8.2. Értesítések megjelenítése	218
8.3. A <i>WebView</i> nézet bemutatása.....	222
8.4. HTTP-kapcsolatok kezelése	228
8.4.1. A HTTP GET támogatása.....	229
8.4.2. <i>AsyncTask</i> használata a HTTP-kommunikációban	235
8.4.3. HTTP POST támogatása	239
8.4.4. A HTTPS és a proxy beállítása.....	240
8.5. Szabványos kommunikációs formátumok feldolgozása.....	242
8.5.1. JSON-feldolgozás	242
8.5.2. XML-feldolgozás	245
8.6. Socket-alapú kommunikáció	247
8.7. Push-típusú értesítések kezelése.....	250
8.8. Hálózati adatforgalom felügyelete	252
9. Telefónia (Fehér Marcell)	255
9.1. Bevezetés.....	255
9.2. Mobilhálózattal kapcsolatos események	255
9.3. Hálózati paraméterek lekérdezése	263
9.4. Telefonhívás programozott indítása	265

9.5. Telefonhívások felügyelete.....	267
9.5.1. Bejövő hívás kezelése	267
9.5.2. Kimenő hívások kezelése	269
9.6. SMS és MMS üzenetek.....	272
9.6.1. SMS küldése	272
9.6.1.1. Implicit Intent használata	272
9.6.1.2. Az üzenet teljes életciklusának kezelése	273
9.6.2. MMS küldése	275
9.6.3. SMS fogadása	276
10. Médiaeszközök kezelése (Ekler Péter).....	279
10.1. Kamerakezelés Android platformon.....	280
10.1.1. A beépített kameraalkalmazás használata	281
10.1.2. Arcfelismerés	285
10.1.3. Saját kamerakezelő készítése	288
10.1.4. Kiterjesztett valóságalapok	295
10.1.5. Videofelvétel és -lejátszás	296
10.2. Multimédia-kezelés	297
10.2.1. Egyszerű hangok lejátszása és felvétele	297
10.2.2. Az <i>AudioManager</i> használata	300
10.2.3. A készülék erőforrásainak ébrentartása hosszú médialejátszás során	301
10.2.4. Hangfelvétel megvalósítása.....	302
10.2.5. MP3-lejátszás	304
11. Android-szolgáltatások (Kelényi Imre).....	307
11.1. Service-alapok.....	308
11.1.1. <i>Service</i> ósosztály	308
11.1.2. A Service-ek deklarálása a manifest állományban	308
11.1.3. A Service-ek két fő típusa: <i>Started</i> és <i>Bound</i>	309
11.1.4. Service-ek a fő szálon.....	310
11.1.5. Service-ek leállítása a rendszerrel	310

11.2. <i>Started Service</i> -ek írása	311
11.2.1. A <i>Service</i> indítása	312
11.2.2. <i>Started Service</i> leállítása	313
11.2.3. Kommunikáció a <i>Service</i> -szel	314
11.2.3.1. <i>Broadcast Intent</i>	314
11.2.3.2. <i>Messenger</i> és <i>Handler</i>	315
11.2.3.3. <i>Pending Intent</i>	315
11.2.4. Egy egyszerű <i>Started Service</i> -példa	315
11.2.5. <i>IntentService</i>	320
11.2.6. Példa az <i>IntentService</i> és a <i>Messenger</i> használatára	320
11.3. <i>Bound Service</i>	323
11.4. Előtérben futó <i>Service</i> -ek	327
11.5. Alkalmazáskomponens automatikus elindítása a készülék indulása (<i>boot</i>) folyamán	329
12. Az Android fejlett funkciói és natív programozása (Ekler Péter)	331
12.1. A <i>Fragment</i> ek bemutatása	331
12.1.1. A <i>Fragment</i> tulajdonságai	333
12.1.2. <i>Fragment</i> -életciklusmodell	333
12.1.3. <i>Fragment</i> ek a gyakorlatban	337
12.2. Fejlett felületi elemek: <i>ActionBar</i> , <i>ViewPager</i> , <i>ViewPagerIndicator</i>	347
12.2.1. Az <i>ActionBar</i> bemutatása	347
12.2.1.1. <i>ActionBar</i> alkalmazásikonjának kezelése	349
12.2.1.2. Egyszerű menüelemek elhelyezése	350
12.2.1.3. Egyedi <i>ActionItem</i> nézet definiálása	351
12.2.1.4. Menüelemek kiterjesztése	353
12.2.2. A <i>ViewPager</i> és a <i>ViewPagerIndicator</i> komponensek bemutatása	354
12.3. Az Android natív programozása	363
12.3.1. A natív programozás jellemzői	363
12.3.2. A fejlesztési környezet és az első natív modul	365
12.3.3. A készülék érzékelőinek használata natív oldalról	373

ELŐSZÓ

Amikor néhány évvel ezelőtt az első érintőképernyős okostelefonok megjelentek, még kevesen gondolták, hogy ezek az eszközök sok szempontból helyettesíthetik az asztali és a hordozható számítógépeket. Napjainkra azonban az újabb és újabb telefonok és táblagépek piacra kerülése jelzi, hogy a mobilkészülékeknek meghatározó szerepük van az informatikai eszközpalettán. A vezeték nélküli hálózati megoldások terjedésével már nemcsak szórakoztató elektronikai megoldásként gondolhatunk rájuk, hanem az aktív munkavégzésben is fontos szerepet töltenek be. Jól megfigyelhető például, hogy számos kis- és nagyvállalat bővítette eszközparkját modern mobilkészülékekkel, integrálva őket a vállalati infrastruktúrába. Ezeknek az eszközöknek a jelentősége a hardverképessegek mellett főként a fejlett szoftvermegoldásokban rejlik, amelyek gyors, látványos és könnyen használható felületen biztosítják, hogy a megszokott informatikai megoldások mobilkészülékeken keresztül is elérhetővé váljanak. A mobilalkalmazások fejlődésében komoly szerepet játszottak a sorra nyíló piacterek, amelyeken keresztül bárki könnyedén értékesíthette az alkalmazásait.

Napjaink egyik legnépszerűbb mobil-operációsrendszere a Google gondozásában készült Android platform, amelynek első verziója 2008-ban került a piacra, és fejlődése azóta is töretlen. A könyv írásakor már a 4-es főverziónál tart a platform, de az 5-ös Android megjelenése is a küszöbön van. A platform egyik legnagyobb ereje abban rejlik, hogy könnyen elérhetővé teszi a Google által biztosított gazdag, internetalapú szolgáltatásokat, és mindezt egy látványos, gyors és egyszerűen kezelhető mobil-operációsrendszerrel párosítja.

Könyvünk célja az, hogy részletesen bemutassa az Android platformot és a felépítését, ismertesse a fejlesztőeszközöket és a legfontosabb programozói interfészeket, valamint olyan gyakorlatorientált példákat mutasson, amelyek sokszor előfordulnak az Android-alkalmazások fejlesztésekor. A példák bemutatásában mindvégig törekedtünk arra, hogy azokat kellő magyarázattal is ellássuk, és a leírások ne csak az aktuális esetekre korlátozódjanak, hanem általános tudást is nyújtsanak a kapcsolódó helyzetekben. Továbbá a könyvben igyekszünk felhívni az olvasók figyelmét a gyakran előforduló hibalehetőségekre és a speciális esetekre is, amelyek figyelembevételével jelentős fejlesztési idő spórolható meg.

A könyv egyrészt a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatóinak készült, az *Android-alapú szoftverfejlesztés* című kurzus tananyagának elsajátításához, de a megírásakor a fő célunk az volt, hogy egy olyan magyar nyelvű könyvet készítsünk, amelyet bárki nyugodtan forgathat, aki szeretne elmélyülni az Android-alapú alkalmazásfejlesztés rejteiben. A mű nem tér ki alapvető programozási fogalmakra, de a részletes magyarázatoknak köszönhetően kezdő szoftverfejlesztők számára is hasznos olvasmány lehet. Az olvasó a könyv megismerésével olyan ismeretekre tehet szert, amelyekkel önállóan képes lesz Android-alapú mobilalkalmazások

tervezésére és fejlesztésére egyaránt, könnyebben átlátja az operációs rendszerrel kapcsolatos újdonságokat, valamint más mobilplatformokat is egyszerűbben elsajátíthat.

A könyv több különálló fejezetből épül fel, amelyeket egymás után érdemes olvasni, de a fejezetek önállósága miatt a munka akár kézikönyvként is használható, ha valamilyen konkrét technológia használatának megismerésére lenne éppen szükség. Az első fejezetben az Android platform kialakulását és alapvető felépítését ismertetjük, a második fejezet az Android-alkalmazások felépítését és az alkalmazáskomponenseket ismerteti. A harmadik fejezetben a felhasználói felület tervezésének és készítésének fogalmait és lépéseit mutatjuk be, míg a negyedik fejezet az alkalmazáskomponensek közti kommunikációt vizsgálja. Az első négy fejezet tehát egy olyan alaptudást nyújt az olvasóknak, hogy utána az összetettebb funkciók is érthetőbbek és egyszerűbben kipróbálhatók lesznek. Az ötödik és a hatodik fejezetekben a perzisztens adattárolás és az állománykezelés eszközeit mutatjuk be, majd a hetedik fejezetben a pozíciómeghatározást és a térképezést ismertetjük. Ezt követően a nyolcadik fejezetben a hálózati kommunikáció lehetőségeit írjuk le, illetve a kilencedik fejezet a mobilspecifikus kommunikáció világába kalauzolja az olvasót. A tizedik fejezet az Android kamerakezelési és multimédia-képességeit tekinti át, a tizenegyedik fejezet az Android szolgáltatáskomponensét írja le, míg végül a tizenkettedik fejezet az Android 3-as és 4-es újdonságait és az Android natív programozásának lehetőségét ismerteti.

A könyv 1., 2., 3., 7., 8., 10. és 12. fejezetének Ekler Péter a szerzője, a 4. és a 9. fejezetet Fehér Marcell írta, az 5. és a 6. fejezetnek Forstner Bertalan a készítője, a 11. fejezetet pedig Kelényi Imre dolgozta ki.

A szerzők törekedtek arra, hogy a könyvben szereplő kódrészek használatát lehetőség szerint önálló alkalmazásokon keresztül is bemutassák, ezek a példák az alábbi oldalon érhetők el: <http://szak.hu/android/peldak.zip>.

A szerzők köszönetüket fejezik ki családtagjaiknak, akik türelemmel kísérték a sokszor hétvégékbe és éjszakákba nyúló munkájukat, valamint köszönet illeti a hallgatókat, akik javaslataikkal növelték a könyv értékét.

Péter szeretné megköszönni menyasszonyának, Antal Évának, hogy támogatja a könyv írása során, valamint azt, hogy a szakmai tartalom átolvasásában is segítséget nyújtott.

Marcell köszöni családjának és barátnőjének a segítőkész támogatást a könyv írásával töltött hosszú és fárasztó napok során.

Bertalan különösen hálás feleségének, Erzsinek, hogy az újabb könyvírási időszakot rutinosan átvészelte, és ezalatt a gyerekeknek apjuk helyett is anyjuk volt. (Ígérem, már csak ezt a mondatot írom le, és átveszem az altatásukat.)

Imre köszöni Dorkának és a Takács családnak a könyvíráshoz a tiszalöki birtokon biztosított ideális körülményeket.

Külön köszönet illeti Csorba Kristófot, aki a szaklektori munkát kérdés nélkül elvállalta, és szabadidejét nem kímélve alaposan átnézte az egyes fejezeteket, precíz észrevételeivel pedig javította a művet.

Köszönjük a BME Automatizálási és Alkalmazott Informatikai Tanszék vezetésének, Dr. Vajk Istvánnak és Dr. Charaf Hassannak a támogatását, hogy biztosították a könyv elkészítéséhez szükséges körülményeket. Valamint köszönet illeti a tanszék összes kollégáját, hogy támogatásukkal segítették a munkánkat. Végül, de nem utolsósorban hálásak vagyunk Takács Dorottyának és Takács Lídiának a borítóterv elkészítéséért.

Végül köszönetet mondunk a SZAK Kiadó munkatársainak a könyv létrehozásához nyújtott segítségükért, továbbá Laczkó Krisztina olvasószerkesztőnek és Bukovics Zoltán tördelőnek lelkiismeretes és alapos munkájukért.

Bízunk benne, hogy olvasóink élvezettel forgatják majd könyvünket, és általa olyan tudásra tesznek szert, amelyet munkájuk során kamatoztatni tudnak.

A munka szakmai tartalma kapcsolódik a „Minőségorientált, összehangolt oktatási és K+F+I stratégia, valamint működési modell kidolgozása a Műegyetemen” című projekt szakmai célkitűzéseinek a megvalósításához. A projekt megvalósítását az Új Széchenyi-terv TÁMOP-4.2.1/B-09/1/KMR-2010-0002 programja támogatja.

Az Android platform bemutatása

A fejezet célja, hogy az Android platformot bemutassa, és a szerkezeti felépítését ismertesse, továbbá az olvasó megismerkedhet a platform kialakulásának történetével is, amely sok esetben magyarázatot ad a platform architektúráis megoldásaira.

1.1. Az Android sikerességének okai

Az Android platform napjaink egyik legsikeresebb mobil-operációsrendszere. A legutóbbi statisztikák szerint több mint 200 millió Android-alapú készülék van már a piacon, és ez a szám rohamosan nő. Naponta mintegy ~700 ezer új Android készüléket aktiválnak.

A rendszernek egy ideig külön verziója létezett telefonokra (2.x) és tábla-PC-kre (3.x) optimalizálva, ám a most megjelent (2012. január) 4.0-s verzió egyesíti ezt a két vonalat, így a jövőben várhatóan minden eszközön ugyanaz a verzió fut majd.

A platform a népszerűségét sok tényezőnek köszönheti, ezek közül kiemelendő a látványos felhasználói felület, az egyszerű használhatóság, a magas fokú kompatibilitás és a nyíltság. A népszerűség másik tényezője az Androidot futtató készülékek fejlett hardverképeségei, ez egyrészt a gyors processzort, valamint a nagyméretű memóriát, másrészt a multimédia-eszközök gazdagságát, harmadrészt pedig a fejlett vezeték nélküli kapcsolatok támogatását jelenti.

Könyvünk írásakor a Google gondozásában készülő, 4.0-t futtató Nexus Prime pontos specifikációja még nem volt ismert, ezért a korábbi Google-„zászlóshajó”, a Nexus One specifikációt ismertetjük példaként.

1.1. táblázat. A Nexus One specifikációja

Rendszer	Android 2.1
Technológia	GSM, UMTS
Méret	119 x 59,8 x 11,5 milliméter
Tömeg	130 gramm
Kijelző átlója	3,7 hüvelyk
Kijelző felbontása	480 x 800 pixel
Kijelző típusa	Kapacitív TFT-érintőkijelző multitouchsal
Memória	512MB RAM, 512MB ROM
Frekvenciasávok	GSM 850/900/1800/1900 MHz, UMTS 900/1700/2100 MHz
GPRS / EDGE	Class 10 (4+1/3+2) / Class 10
UMTS / HSDPA / HSUPA	Van / 7,2 Mbps / 2 Mbps
IrDA / Bluetooth	Nincs / 2.1 (A2DP is)
WiFi	802.11b/g
USB	2.0 (microUSB)
Push-to-talk / RSS	Nincs/van
GPS vevő	Van
Profilok	Nincsenek, csak néma mód
Fő kamera	5 megapixeles, autofókuszos, LED-es villanófény

A Nexus One készülék 2010 januárjában került a piacra, ám a fenti táblázatból látható, hogy hardverképességeivel még a mai igényeket is kielégíti.

Az Android platformot napjainkban is folyamatosan fejlesztik, és az Android-alapú készülékek egyik jellemzője az, hogy lehetőség van a rendszer frissítésére, ha az adott készülékgyártó úgy ítéli meg, hogy engedélyezi az eszközre a frissítést. Emellett a platform nyíltságából következik, hogy számos egyedi szoftververzió is készült már. Ennek eredményeképpen többek között a Nexus One készülék is frissíthető a 2.3-as Android-verzióra.

A fejezet további részeiben elsőként a platform rövid történetét mutatjuk be, ezt követi az Android-verziók ismertetése és az Android Market rövid leírása, majd rátérünk a platform mérnöki szempontú bemutatására, és ismertetjük az Android mobil-operációsrendszer szerkezetét, ahol kitérünk a telepítőállományok felépítésére és a biztonsági kérdésekre is. A fejezet végén az Android-alkalmazások fejlesztéséhez szükséges fejlesztőeszközök telepítését és használatát mutatjuk be röviden.



1.1. ábra. Nexus One készülék

1.2. Az Android platform története

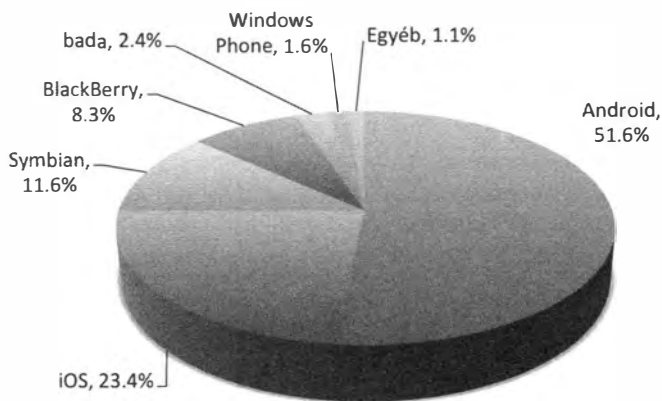
Az Android napjaink egyik legnépszerűbb, és bátran állíthatjuk, hogy egyik legfejlettebb operációs rendszere. A népszerűség egyik fő oka a rengeteg médiareklám. Az Android előtt kevés mobileszközt reklámoztak az azt futtató operációs rendszerrel, így az Android mint operációsrendszer-név kellően újszerű volt a felhasználók számára. Megfigyelhető, hogy új mobileszköz vásárlásakor a felhasználók már nem is feltétlenül a készülékgyártó alapján, hanem operációs rendszer szerint keresnek eszközöket. A sikerhez természetesen az is hozzátartozik, hogy az Android mögött, az „IT-óriás”, a Google áll. Az Android további előnye, hogy egy rendkívül egységes és kiválóan működő rendszerképet tükröz a felhasználóknak, és elfedi az egyes verziószámokat, valamint a köztük lévő különbségeket. Az Android az iOS-szel karöltve tulajdonképpen forradalmasította az operációs rendszerekről alkotott képet, és a felhasználói igények kielégítése is egyre nagyobb hangsúlyt kapott a mobiloperációsrendszerek piacán.

Az Android jellemzője, hogy nemcsak mobiltelefonokon, hanem táblagépeken is fut, ezek elterjedése pedig napjainkban rohamosan nő. Továbbá az Androidot úgy tervezték, hogy akár más eszközökön is könnyen futtatható legyen: akár televíziókról, gépjárművek fedélzeti számítógépéről, ipari automatizálási rendszerről vagy háztartási eszközökről. Tulajdonképpen az Android minden olyan helyen kényelmesen használható, ahol relatíve limitáltak az erőforrások, és az adatbevitel nem feltétlenül egerrel és/vagy billentyűzettel történik.

Az Androidon egyrészt magát a mobil-operációs rendszert értjük, másrészt pedig a rendszert futtató eszközök (telefonok, táblagépek stb.) összességét. A Google 2005-ben felvásárolta az *Android Incorporated* nevű vállalatot, és ezután saját maga kezdte meg a mai operációs rendszer fejlesztését, így tulajdonképpen az Android részben (vagy akár teljes egészében) a Google fejlesztése. 2007-ben láttak napvilágot az első olyan hírek, amelyek szerint a Google saját mobil-operációs rendszerrel kíván piacra lépni, majd 2007. november 5-én az akkor már létrejött Open Handset Alliance bejelentette az Android platformot. Az első készülék (HTC G1) a T-Mobile forgalmazásában 2008 végén került piacra.

Az Android így ingyenes és nyílt forráskódú operációs rendszerré nőtte ki magát, amelyet hivatalosan az Open Handset Alliance konzorcium fejleszt, és amelynek a Google vezetője. A konzorciumnak mintegy 80 különböző szoftver-, hardver- és telekommunikációs cég tartozik a tagjai közé. A rendszer egy monolitikus Linux-kernel köré épül, ahol az alkalmazásfejlesztés elsődleges nyelve a Java. A rendszer a Java nyelven megírt Android-alkalmazások futtatásához a Dalvik virtuális gépet használja: ez egy Java virtuális géphez hasonló VM, amely olyan hordozható eszközökre van optimalizálva, amelyek korlátozott memóriával és CPU-sebességgel rendelkeznek. A fejlesztőknek ingyenes fejlesztést biztosít, és az elkészített alkalmazások értékesítésére a központi Android Market alkalmazásbolton kívül is van lehetőség.

Az Android sikerének legfőbb okai közé sorolható a látványos felhasználói felület, a piacon megjelent sok és arányaiban olcsó modell, a kis hardverigény, a nyílt forráskód és az ingyenes használat, valamint a sikeres marketingstratégia. Ezek az okok mind hozzájárulnak ahhoz, hogy napjaink egyik legjelentősebb mobil-operációs rendszereként emlegessük az Androidot.



1.2. ábra. Az Android piaci részesedése 2012 első negyedében¹

¹ Forrás: Canalys

Az Android a megjelenésekor mind a nagy gyártók, mind pedig a kisebb vállalatok számára ideális alternatíva volt. A nagy gyártók szemszögéből nézve: az Android megjelenésének idején eladásaik csökkenőben voltak, és tulajdonképpen létkérdéssé vált számukra, hogy támogatják-e az Androidot. Emellett pedig a nagyobb cégek nem rendelkeztek erőforrással saját rendszer fejlesztésére (kivéve a Samsung Badát).

A kisebb vállalatok számára pedig az Android ugródeszkát jelentett, hiszen ingyenessége miatt ezek a vállalatok könnyen a piacra tudtak kerülni, és nagyon jó ár/érték arányú készülékeket tudtak nyújtani a felhasználóknak. Részben ennek köszönhető, hogy a világ új márkanéveket ismert meg (pl. ZTE, Huawei stb.). A kis cégek szempontjából további előny volt az is, hogy az Android-alapú készülékekkel a szolgáltatók/operátorok közé is be tudtak lépni, és készülékeikkel megjelentek az operátorok készülékpalettáin.

Az Android ingyenességét már többször hangsúlyoztuk, ám az ingyenségnek vannak bizonyos korlátai. A gyártók valóban ingyen hozzáférhetnek a rendszerhez, a Google Maps-alkalmazáshoz, a YouTube-hoz és az Android Markethez azonban csak akkor enged hozzáférést a Google, ha a készülék eleget tesz bizonyos minimális követelményeknek, így a rendszer terjedése és megbízhatósága kezelhető.

Ezek a minimális követelmények a következők:

- legalább QVGA-felbontású kijelző,
- 128 MB RAM és 256 MB flashmemória,
- Bluetooth,
- mini- vagy microSD,
- wifi.

Látható, hogy a felsorolt három alkalmazás nélkül a készülék funkciói jelentős mértékben korlátozottak, ezek nélkül csupán egy átlagos *feature phone* kategóriába sorolhatók azok a készülék, amelyek nem tesznek eleget a minimális követelményeknek.

Az Android sikere az eddig felsoroltak mellett még a rendszer mögött álló számos népszerű Google-szolgáltatásának is köszönhető, amelyek teljes értékű alkalmazásként elérhetők az Android-alapú készülékeken. Ilyen szolgáltatások például a *GMail*, a *GTalk*, a *Picasa* és a *Latitude*.

1.3. Android-verziók

Az Android platform értékelésekor az egyik legnagyobb hátránnyként a különféle verziókat és a verziók közti különbségeket szokták hangsúlyozni. Az érvelések sokszor azonban tévesek, hiszen az Android-verziók visszafelé teljes mértékben kompatibilisek, és egy korábban megírt alkalmazás garantáltan fut a legújabb Android-verziókon is. Mindezek mellett fejlesztőként nagyon fontos nyomon

követni az Android fejlődését, hiszen egy új verzió számos újítást is hozhat, ezek pedig megkönnyítik a fejlesztői munkát, illetve elképzelhető az is, hogy az egyes verzióváltásokkor a fejlesztésre vonatkozó szabályok/ajánlások is megváltoznak, amelyeket mindenképpen érdemes betartani. A következőkben ismertetjük a könyv írásakor aktuális Android-verziókat és ezek fő képességeit.

Android 1.0

- 2008. október 21-én jelent meg.
- Apache-licenc.
- Egy szűk fanatikusból álló rétegen kívül nem nyerte el igazán az átlagfelhasználók tetszését.
- A platform stabilitása megfelelő volt, a használhatósága azonban nehézkesnek bizonyult.
- A UI nem volt megfelelő.
- A HTC által gyártott G1 is inkább csak koncepciótelefon volt, hogy elősegítse a fejlesztők munkáját, illetve felkeltse a cégek érdeklődését.

Android 1.1

- 2009 februárjában jelent meg.
- Felkerült a G1 telefonokra (frissíthetőség tesztelése).
- Sok apró hibát javított, ezeket az 1.0-s kiadásától fogva gyűjtötték.
- Látványos változtatásokat nem tartalmazott.

Android 1.5 (Cupcake)

- 2009 áprilisában jelent meg.
- 2.6.27 verziójú Linux-kernelen alapul.
- A szoftveres billentyűzet automatikus kiegészítési funkciójával rendelkezik.
- A2DP Bluetooth-támogatása, illetve automatikus headsetcsatlakozása van.
- Új UI-komponensek jelentek meg benne.
- Animációkat vezettek be a képernyőváltások között.
- Feljavították másolás-beillesztés funkciót.
- Videók és képek közvetlen feltöltése vált lehetségessé a YouTube és a Picasa portáljaira.



1.3. ábra. *Android Cupcake-logó*

Android 1.6 (Donut)

- 2009 szeptemberében jelent meg.
- Az előző verzió javítása.
- Android Market-javításokat tartalmaz.
- Feljavított galériefunkcionalitásokkal rendelkezik (több kép kijelölése közös művelethez).
- Hangfelismerésen alapuló funkciók vannak benne.
- Teljes platformban képes keresni az alkalmazás megjelenését.
- A használt technológiák frissítését, a WVGA-felbontás támogatását, egyéb optimalizálásokat tartalmaz.



1.4. ábra. *Android Donut-logó*

Android 2.0 és 2.1 (Eclair)

- 2009 októberében jelent meg.
- Nagyobb verzióváltás történt.
- 2.6.29-es Linux-kernel-támogatás.
- Hardveroptimalizációt is tartalmaz.
- Változatos képernyőméretek és felbontások támogatása (netbook- és táblagép-támogatás).

- Újraértelmezett grafikus felület HTML5 támogatással.
- Multitouch támogatása.
- Bluetooth 2.1-es támogatás.
- „Élő” háttér megjelenése.
- A 2.0 kiadása után nem sokkal érkezett a 2.0.1-es verzió, amely több apró – de bosszantó – hibát javított.
- 2010 januárjában jelent meg a 2.1-es verzió, amely további javításokat hozott.
- Az Eclair legsikeresebb verziója a 2.1 lett, a 2.0 és a 2.0.1 minimálisan terjedt el, az OHA-tagok eszközeire csak 2.1 került fel.
- Néhány ismert eszköz: Samsung Galaxy Spica (GT-I5700), Galaxy 3 (GT-I5800), Galaxy S (GT-I9000), Motorola Defy (Motorola MB525).



1.5. ábra. Android Eclair-logó

Android 2.2 (Froyo)

- 2010 májusában jelent meg.
- Feljavított böngészője van: Flash 10.1 és akár háromszor gyorsabb JavaScript.
- JIT-támogatással rendelkezik, amely a CPU-igényes feladatokat 400–500 százalékkal gyorsíthatja.
- Stream és push támogatása.
- Ad hoc wifimegosztás.
- Teljesítménybeli és felületi javítások történtek.
- Az alkalmazások nagy részét a MicroSD-kártyára lehet másolni és ugyanígy vissza is helyezni.
- Hangalapú tárcsázás.
- Névjegymegosztás Bluetoothon keresztül.



1.6. ábra. *Android Froyo-logó*

Android 2.3 (Gingerbread)

- 2010. december 6-án jelent meg.
- A Samsunggal közös Nexus S telefon.
- Új felhasználói interfésze van.
- Nagyobb felbontású kijelzőket támogat.
- 2.6.35.7 Linux-kernelt alkalmaz.
- Támogatja a WebM-videolejátszást.
- Near Field Communication (NFC) támogatása.
- Továbbfejlesztett másolás-beillesztés funkció.
- Átalakított gyári virtuális billentyűzet, multitouch támogatás.
- Javított energiagazdálkodás, hosszabb üzemidő.
- Optimalizáció (gyorsabb, hatékonyabb működés).
- Internethívás (VoIP) támogatása.
- Letöltéskezelő a hosszú ideig tartó HTTP-letöltésekhez.
- Új szenzorok (pl. giroszkóp) támogatása és kezelése.
- YAFFS helyett ext4-es fájlrendszer használata.



1.7. ábra. *Android Gingerbread-logó*

Android 3.0 (Honeycomb)

- 2011 januárjában jelent meg.
- Táblagép-támogatással rendelkezik.
- Újragondolt felületet kapott.
- Táblagép PC-hez optimalizált kezelése van (pl. átalakított, megnövelt méretű virtuális billentyűzet).
- Többmagos processzorok támogatása.
- Teljes kompatibilitás a korábbi verziókra készült programokkal.
- Fejlettebb szövegkijelölés, másolás-beillesztés.
- USB és Bluetooth külső billentyűzetének kezelése.
- Javított wifihálózat-keresés és Bluetooth-tethering.
- Felújított, kibővített gyári alkalmazások (böngésző, kamera, galéria, névjegyzék, e-mail).
- 3.0.1: kisebb update a Flash Player 10.1-es támogatáshoz.



1.8. ábra. Android Honeycomb-logó

Android 3.1

- 2011 májusában jelent meg.
- Fejlettebb UI-effektek találhatók benne.
- Gyorsabb és látványosabb animációkkal rendelkezik.
- UI-elemek fejlesztése (szín, méret, kezelhetőség stb.).
- USB-eszközök támogatása (egér, billentyűzet, játékvezérlő, kamera stb.).
- Átméretezhető widgetek.
- Minden wifi *access poin*thoz külön HTTP-proxy-beállítás tartozik.

- Beépített alkalmazások fejlesztése.
- USB-host-API.
- Külső kameraintegráció (MTP – Media Transfer Protocol, PTP – Picture Transfer Protocol).
- RTP API (Real-time Transport Protocol): streaming támogatása.

Android 3.2 (Ice Cream Sandwich)

- 2011 júliusában jelent meg.
- Kisebb frissítés.
- További optimalizáció táblagépek számára.
- Nagyítás támogatás kisebb kijelzőkre készített alkalmazások számára (iPhone-iPadhez hasonlóan).
- *Media sync* támogatása SD-kártyára.
- További támogatás a táblagép-UI fejlesztéséhez.

Android 4.0 (Ice Cream Sandwich)

- 2011 októberében jelent meg.
- Új készenléti kijelzője, gyorsindítója és feladatkezelője van.
- Skálázható kezelőfelülettel rendelkezik.
- Az alkalmazások könnyen alkalmazkodhatnak az eltérő felbontású és fizikai méretű kijelzők adottságaihoz, amelyet az osztott képernyős megoldásokat támogató *Fragments* API is tovább segít.
- Az integrált arckövető megoldás révén a képernyőn megjelenő 3D-s alakzatok mindig a nézőnek megfelelő perspektívában jelennek meg.



1.9. ábra. Android Ice Cream Sandwich-logó

A platform tehát a megjelenésétől számítva olyan újításokon ment keresztül, amelyekre sokszor mérföldkőként tekinthetünk a mobilplatformok területén.

A következő adatok azt mutatják, hogy hogyan alakult azoknak a készülékverzióknak az eloszlása, amelyek 2011. november 3-ig bezárólag, egy 14 napos időszakban az Android Market alkalmazásboltot meglátogatták.

1.2. táblázat. *Android platform verziók eloszlása*²

Platform	Kódnév	API Level	Eloszlás
Android 1.5	Cupcake	3	0,9%
Android 1.6	Donut	4	1,4%
Android 2.1	Eclair	7	10,7%
Android 2.2	Froyo	8	40,7%
Android 2.3 2.3.2	Gingerbread	9	0,5%
Android 2.3.3 - Android 2.3.7		10	43,9%
Android 3.0	Honeycomb	11	0,1%
Android 3.1		12	0,9%
Android 3.2		13	0,9%

Az adatok között még nem szerepel a mostanában bejelentett Ice Cream Sandwich kódnevű 4.0-s változat elterjedtsége, ez ugyanis még olyan friss platformverzió, hogy hivatalosan nem volt elérhető egy készülékre sem az adatgyűjtéskor. Az adatokból jól látszik, hogy a készülékek közel 98%-a legalább Android 2.1-es verziójú, és közel 90%-a legalább 2.2-es verziójú. A 2.3-as verzió már csak körülbelül a készülékek felére érhető el, a 3.0-s, kizárólag tábla-PC-kre megjelent változat pedig mindössze néhány százalék elterjedtségű.

A gyártók szempontjából a platform mellett a legkomolyabb érv az ingyenes elérhetőség és a nyíltság. Emellett a Google folyamatos innovatív megoldásainak köszönhetően a legújabb és legnépszerűbb fejlesztések is szinte azonnal elérhetővé válnak az Android-alapú készülékeken (pl. arcfelismerés).

² <http://developer.android.com/resources/dashboard/platform-versions.html>

1.4. Android Market (Google Play)

Mielőtt a platform szerkezetét ismertetnénk, röviden bemutatjuk az Android Marketet, hiszen ez az alkalmazások publikálásának elsődleges felülete.

Az Android Market 2008. október 22-től érhető el a felhasználók számára. Ez tulajdonképpen egy Google által fejlesztett és karbantartott alkalmazásbolt Android készülékek számára. A Market mint alkalmazás, a platform nyíltságával ellentétben, nem nyílt forráskódú, ennek okai között biztonsági kérdések is vannak. A legtöbb Android-alapú készüléken, amely megfelel a minimális hardverkövetelményeknek, előre megtalálható a Market-alkalmazás, így a Google-azonosítónk megadásával azonnal használhatjuk is.

Egyes jóslatok szerint 2011 végére / 2012 elejére az Android Market az alkalmazások számát tekintve megelőzheti az Apple AppStore-t. 2011 végére körülbelül 10 milliárd letöltést számoltak meg.

Az Android Market főbb jellemzői a következők:

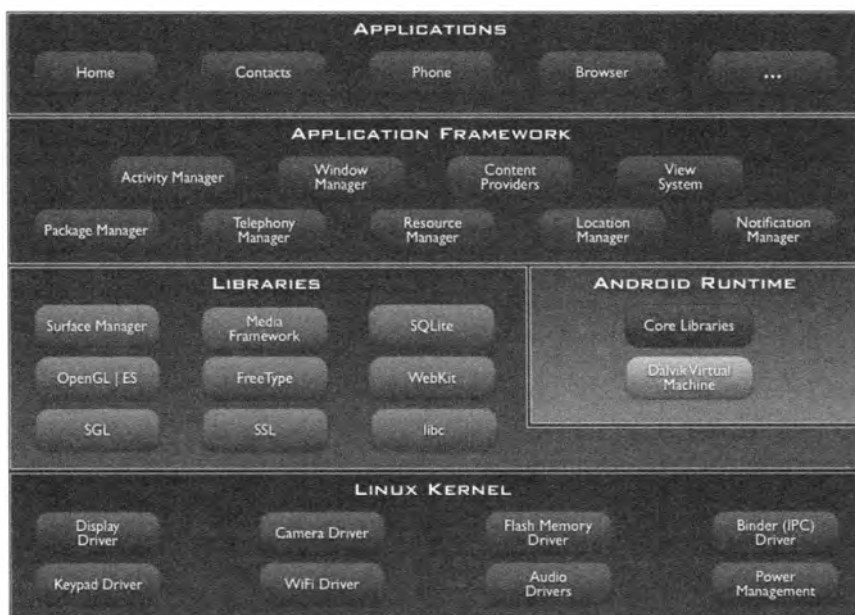
- A bevétel 70%-a a fejlesztőé, 30%-a pedig a szolgáltatóé és a fizetést biztosító cégé.
- A fejlesztő beállíthatja, hogy az adott alkalmazás milyen típusú készülékeken futtatható, és mely országokban kívánja publikálni.
- A Market szűri az alkalmazásokat a futtató készülék típusának megfelelően.
- A szolgáltatóknak lehetőségük van letiltani bizonyos tartalmakat.
- A Market biztosít egy úgynevezett *Android licensing service*-t, amelynek segítségével a letöltött alkalmazás futás közben ellenőrzi, hogy megvásárolta-e a felhasználó.
- A Market-eladásból származó nyereség 15 perc alatt a fejlesztőhöz kerül (2010 decemberétől).
- Weben keresztül is elérhető a Market (2011 februárjától).
- Újragondolt kategorizálási rendszere van, amelynek célja, hogy minél több alkalmazást előtérbe helyezzen.

Publikálás esetén az alkalmazás védelme komplex kérdést jelent. Az Android platformon lehetőség van egy úgynevezett *Market Licensing* szolgáltatás használatára, amellyel a fizetős alkalmazások ellenőrizhetik, hogy az adott készülékre valóban a Marketről töltötték-e le őket. A Market üzleti célú használata előtt mindenképp érdemes a megfelelő használati módról tájékozódni a Google adott oldalán.³

³ Market: <http://developer.android.com/guide/publishing>

1.5. A platform szerkezete

Az Android egy Linux-kernel-alapú mobil-operációsrendszer. A következőkben a platform szerkezetét tekintjük át, és megvizsgáljuk az egyes rétegek szerepét. A következő ábrát gyakran használják az Android fejlesztői táborában, ezért magyar fordítást nem adunk hozzá.



1.10. ábra. Az Android platform szerkezete⁴

Összességében a platform felépítése logikusnak és áttekinthetőnek mondható. A legalsó szinten található a Linux-kernel, amelynek feladata a memória kezelése, a folyamatok ütemezése és az alacsony fogyasztást elősegítő teljesítménykezelés. Ezen a szinten találhatók továbbá a hardvert kezelő eszközmeghajtók programjai. Ezeket a programokat tipikusan azok a cégek készítik el, amelyek az Android platformot saját készülékükön szeretnék használni, hiszen a gyártónál jobban más nem ismerheti a mobileszközbe integrált perifériákat.

A kernel fölött találhatók a különféle programkönyvtárak vagy szolgáltatások, például: *libc*, *SSL* (titkosítás), *SQLite*, *OpenGL/ES*, *WebKit* stb. Ezek a könyvtárak jellemzően C/C++ nyelven készültek. Részben a felsorolt könyvtárakra épül az Android-futtatókörnyezet, amelynek fő eleme a *Dalvik* virtuális gép. A Dalvik feladata az Androidra készített Java-alkalmazások futtatása a személyi számítógépek világában megszokott Java Virtual Machine-hez (JVM) hasonlóan. A Dalvik ezen JVM egyik jelentősen újratervezett, átdolgozott, optimalizált verziója.

⁴ Forrás: <http://developer.android.com/guide/basics/what-is-android.html>

A Dalvik fő jellemzői a következők:

- Nem kompatibilis a korábbi Sun virtuális géppel.
- Megújult utasításkészlettel dolgozik.
- A Java-programok nem egy-egy `.class` állományba kerülnek fordítás után, hanem egy nagyobb *Dalvik Executable* formátumba, amelynek kiterjesztése `.dex`, és általában kisebb, mint a forrásul szolgáló `.class` állományok mérete, mivel például a több Java-fájlban megtalálható konstansokat csak egyszer fordítja bele a Dalvik-fordító.
- A Java csak mint nyelv jelenik meg.

A Dalvik virtuális gépen tehát a Javában készített úgynevezett felügyelt kód (managed code) fut. Ez a megoldás az Android-alkalmazások futtatását rendkívül biztonságossá teszi, hiszen így egy alkalmazás nem vagy csak nagyon ritkán tudja megbénítani az egész rendszert. A Dalvik-környezetben a memóriakezelés tipikusan *garbage collector*tal történik, ám ennek ellenére ügyelnünk kell arra, hogy hatékony kódot írjunk, és kerülni kell a felesleges memórafoglalásokat.

A legfelső rétegben már csak Java-alapú megoldásokat találunk, amelyet a virtuális gép futtat, és ez adja az Android lényegét: a látható és tapintható operációs rendszert, illetve a futó programokat. A virtuális gép akár teljesen elrejtí a Linux által használt fájlrendszert, és csak az Android Runtime által biztosított fájlrendszert láthatjuk. Az Android Runtime két fő egységre bontható: az alkalmazás-keretrendszerre (Application Framework) és magukra a rendszeren futó alkalmazásokra. A keretrendszer feladata, hogy kiszolgálja az alkalmazásokat, és hozzáférést biztosítson a rendszer különféle erőforrásaihoz, a legfelső alkalmazásréteg feladata pedig a felhasználó által elérhető programok kezelése.

1.5.1. Az *apk* állomány felépítése

Android platformra úgynevezett *apk* állományokat telepíthetünk, amelyek tulajdonképpen becsomagolva tartalmazzák az Android-alkalmazást. Az *apk* állomány leginkább a Symbian platformon megszokott *sis* állományhoz hasonló. Bármilyen formában eljuttatva a telefonra az *apk*-t, utána könnyedén telepíthetjük. Ha a Marketről töltünk le, akkor is egy *apk* állomány települ. A telepítést az úgynevezett *PackageManagerService* végzi.

Az alkalmazás telepítésének a lépései a következők:

- metainformációk áttekintése,
- céltároló kiválasztása (készülékmemória vagy SD-kártya, ha a platformverzió támogatja),

- alkalmazás hozzáférési jogosultságának a jóváhagyása (milyen műveleteket hajthat végre a program), például:
 - internetelérés,
 - telefonhívás,
 - üzenetküldés,
 - telefonkönyv elérése,
 - írás/olvasás memóriakártyára/memóriakártyáról.

Ki kell emelni a felsorolás utolsó pontját: ennek értelmében tehát telepítés-kor ellenőrizhetjük, hogy pontosan milyen technológiákat is használ az alkalmazás. Érdemes gondosan áttekinteni ezt a listát, hiszen előfordulhat, hogy valamilyen kártékony alkalmazást telepítünk. Tipikusan gyanús például, ha egy játék telefonhívás-jogosultságot tartalmaz. Egyes felmérések szerint a Marketen lévő alkalmazások 5%-a képes telefonhívást indítani a felhasználó beavatkozása nélkül, ezekre mindenképpen oda kell figyelni.

Az *apk* tulajdonképpen egy tömörített állomány, amely a lefordított forráskódot, az erőforrásokat és néhány metainformációt tartalmaz.

Az *apk* tipikus tartalma a következő:

- *META-INF* könyvtár:
 - *CERT.RSA*: alkalmazástanúsítvány,
 - *MANIFEST.MF*: metainformációk kulcs-érték párokban,
 - *CERT.SF*: erőforrások listája és SHA-1 hashértékük, például:
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA1-Digest-Manifest: wxqnEAI0UA5nO5QJ8CGMwj kGGWE=
* * *
Name: res/layout/exchange_component_back_bottom.xml
SHA1-Digest: eACjMjESj7Zkf0cBFTZ0nqWrt7w=
* * *
Name: res/drawable-hdpi/icon.png
SHA1-Digest: DGEqylP8W0n0iV/ZzBx3MW0WGCA=
- *Res* könyvtár: az erőforrásokat tartalmazza;
- *AndroidManifest.xml*: név, verzió, jogosultság, könyvtárak;
- *classes.dex*: lefordított osztályok a Dalvik számára érthető formátumban;
- *resources.arsc*: erőforrásadatok.

1.5.2. A platform jellemzői és a fordítás mechanizmusa

A következőkben röviden bemutatjuk az Android-alkalmazások felépítését fejlesztői szempontból. (Az egyes elemeket részletesebben lásd a későbbi fejezetekben.) Android platformon tehát az alkalmazások fejlesztéséhez általánosan a Java nyelvet használhatjuk, amelyhez egy SDK-t (Software Development Kit) biztosít a Google. Alacsonyabb szintű funkciók eléréséhez lehetőségünk van natív kódot is készíteni az NDK (Native Development Kit) segítségével. A natív kód hívásához használhatjuk a JNI-t (Java Native Interface), ám a rendszer támogatja az osztott könyvtárak (shared libraries) használatát is. A fejlesztés megkönnyítésére egy projekten belül elhelyezhetjük a Java- és C++-kódrészeket is.

Android-alkalmazások fejlesztésekor magasabb szintű Java nyelvi elemeket is használhatunk, ellentétben például a Java ME-vel, ahol csak a 3-as nyelvi eszköztár van támogatva.

A Java nyelven írt alkalmazások külön a Dalvik virtuális gép példányon futnak felügyelten, a memóriakezelésért a futtatókörnyezet és a virtuális gép a felelős. A memória felszabadítását ennek megfelelően egy GC (Garbage Collector) végzi, ám ez nem jelenti azt, hogy felelőtlenül bánhatunk az objektumok létrehozásával. Törekedni kell azok folyamatos felszabadítására, hiszen a GC csak a már nem hivatkozott és nem használt objektumok memóriaterületét tudja felszabadítani. Az eseménykezelés a Javában megszokott módon történik, a megfelelő objektumokhoz úgynevezett *Listenereket* definiálhatunk, és a megfelelő *interface* függvényeken keresztül kapunk értesítéseket az események bekövetkezésekor. A kivételkezeléshez szintén a standard Java *try-catch-finally* kivételkezelő módszer használatos.

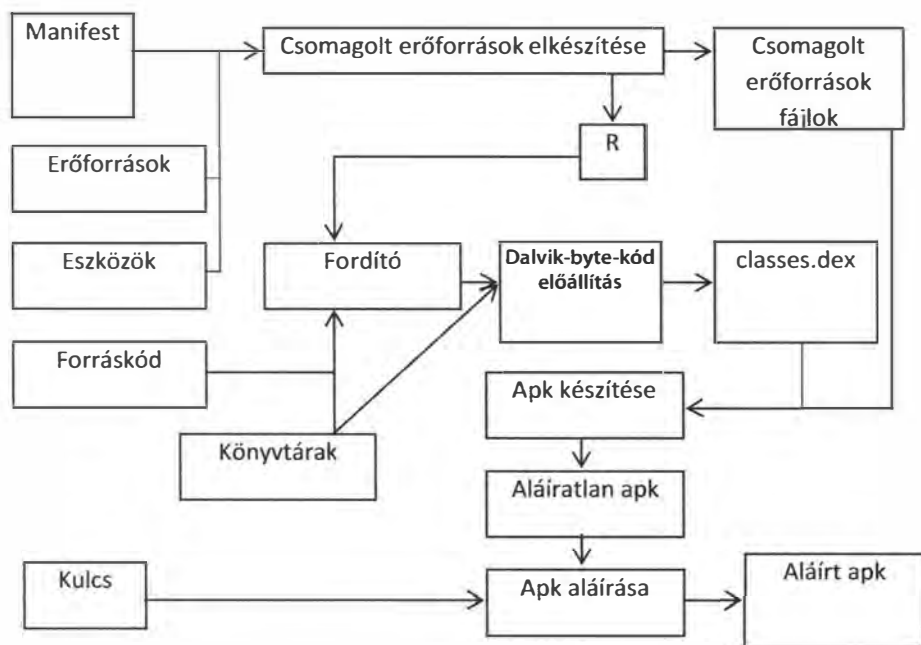
Android-fejlesztés esetén a forráskód és a felhasználói felület definíciója különválik, a felhasználói felület definiálására XML-állományokat használhatunk, így lehetőségünk van arra, hogy a felületet deklaratív módon adjuk meg. Ez azonban nem zárja ki, hogy a forráskód szintjén is létrehozzuk, elérjük és manipuláljuk a felhasználói felületet. „Ökölszabályként” elfogadott: a felhasználói felület definíciója minél inkább különöljön el a forráskódtól. A projektleíró állomány szintén XML-formátumban érhető el (Android Manifest).

Egy új Android-projekt létrehozása után a forráskód az *src* könyvtárban, míg a felhasználói felület leírására szolgáló XML-állományok a *res* könyvtárban találhatók. Az erőforrás-állományokat egy *R.java* állomány köti össze a forráskóddal, így könnyedén elérhetjük Java-oldalról az XML-ben definiált felületi elemeket. Az Android-projekt fordításának eredménye a korábban bemutatott *apk* állomány, amelyet tehát közvetlenül telepíthetünk mobil eszközre.

A fordítás mechanizmusa a következő lépésekből áll:

- A fejlesztő elkészíti a Java-forráskódot, valamint az XML-alapú felhasználói felületleírást a szükséges erőforrás-állományokkal.
- A fejlesztőkörnyezet az erőforrás-állományokból folyamatosan naprakészen tartja az *R.java* erőforrásfájlt a fejlesztéshez és a fordításhoz.
- A fejlesztő a *manifest* állományban beállítja az alkalmazás hozzáférési jogosultságait (pl. internetelés, szenzorok használata stb.).
- A fordító a forráskódból, az erőforrásokból és a külső könyvtárakból előállítja a Dalvik virtuális gép byte-kódját.
- A byte-kódból és az erőforrásokból előáll a nem aláírt *apk* állomány.
- Végül a rendszer végrehajtja az aláírást, és előáll a készülékekre telepíthető aláírt *apk*.

A következő ábrán bemutatjuk a fordítás lépéseit.



1.11. ábra. A fordítás lépései

A fordítás teljes folyamata a fejlesztői gépen megy végbe, a készülékekre már csak a bináris állomány jut el. A külső könyvtárak általában JAR állományként vagy egy másik projekt hozzáadásával illeszthetők az aktuális projekthez.

A *manifest* állományban meg kell adni a támogatandó Android-verziót, amely felfelé kompatibilis az újabb verziókkal, régebbi verzióra azonban már nem telepíthető a program. Látható tehát, hogy a teljes folyamat a szoftverfejlesztők számítógépein megy végbe, az ügyfélhez a futtatható gépi kód jut el.

1.6. A fejlesztőkörnyezet bemutatása

A következőkben ismertetjük az Android-fejlesztőkörnyezet telepítésének fő lépéseit, valamint a fejlesztőeszköz és az emulátor legfőbb funkcióit.

1.6.1. Telepítés

A lépéseket a Windows operációs rendszerhez adjuk meg, ezek azonban Linux és MacOS alatt is nagyon hasonlóak.

Az Android SDK-t (Software Development Kit) érdemes a *C* meghajtóra telepíteni egy szóközoeket nem tartalmazó könyvtárba, valamint azt javasoljuk, hogy a *windows felhasználónévben* se legyen szóköz. Ellenkező esetben a virtuális gépek alapértelmezett helyét meg kell változtatni a telepítés után (*ANDROID_SDK_HOME* környezeti változó), ugyanis az SDK hibásan kezeli a szóközt tartalmazó könyvtárakat.

Az SDK egy olyan teljes fejlesztői csomag, amely nemcsak a fordításhoz szükséges eszközöket tartalmazza, hanem emulátort, dokumentációt, példa-programokat, USB *drivert* és még számos eszközt, köztük az *adb*-t, amellyel konzolos interfészen lehet az Android rendszerrel kommunikálni, legyen szó akár emulátorról, akár konkrét eszközről.

A telepítés fő lépései a következők:

- Számítógép előkészítése, előkövetelmények ellenőrzése
- SDK letöltése és telepítése
- Eclipse ADT plugin telepítése
- Megfelelő Android platformverzió telepítése az SDK segítségével
- Emulátor létrehozása (AVD)

Az előkövetelmények a következők:

- JDK 6: <http://www.oracle.com/technetwork/java/javase/downloads/jdk-6u32-downloads-1594644.html>
- Eclipse: <http://www.eclipse.org/downloads/>

A telepítés részletes lépései az alábbi weboldalakon találhatóak (itt nem fejtjük ki részletesen, hiszen a folyamatos fejlődés következtében esetenként változhat egy-egy lépés):

- <http://developer.android.com/sdk/index.html>
- <https://dl-ssl.google.com/android/eclipse/>

A telepítéskor elsőként az Android SDK-t kell telepítenünk, utána az Eclipse plugint, amely összekapcsolja az Eclipse-et az Android SDK-val, és lehetővé teszi az Android-projektek kezelését az Eclipse-en belül.

A telepítést követően érdemes az Android SDK könyvtárát szemügyre venni:

- *add-ons/*: kiegészítők külső könyvtárak használatához
- *docs/*: offline dokumentáció
- *platform-tools/*: eszközök, például az *adb* az emulátorok/készülékek vezérléséhez
- *platforms/*: az egyes platformverziók
- *samples/*: példakódok platformverzióként
- *tools/*: platformverzió-független eszközök, például: emulátor, *ddms* stb.
- *SDK Manager.exe*: SDK- és AVD- (emulátor) kezelő eszköz

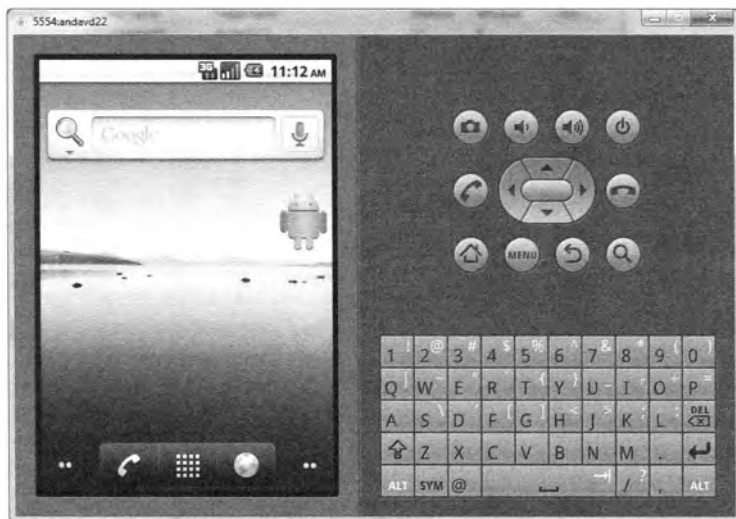
Az SDK Manager segítségével frissíthetjük az SDK-komponenseket, újakat tölthetünk le, valamint kiválaszthatjuk, hogy melyik Android-verziót szeretnénk letölteni az SDK-val. Csak a letöltött Android-verzióknak megfelelő emulátorokat (AVD) lehet létrehozni. Az emulátor egy úgynevezett Android Virtual Device (AVD) emulátorkonfiguráció megadásával indítható. Az SDK Manageren keresztül elérhető AVD Manager segítségével állítható össze egy AVD-konfiguráció, ahol tipikusan a következőket kell megadni:

- név,
- Android platform verziója,
- SD-kártya,
- skin / megjelenítési mód,
- képernyő-tulajdonságok.



1.12. ábra. Új AVD létrehozása

Az Android-emulátor a teljes Android rendszert emulálja, éppen ezért nem minden esetben olyan gyors, mint egy megszokott alkalmazás, ám a rendszer teljes viselkedésének megismerésére jól használható, továbbá az Android beépített alkalmazásait is tartalmazza.



1.13. ábra. Android-emulátor

Az emulátor elindítása után érdemes kipróbálni a korábban említett *adb* eszközt (\android-sdk\platform-tools\adb.exe) például a következő parancsokkal:

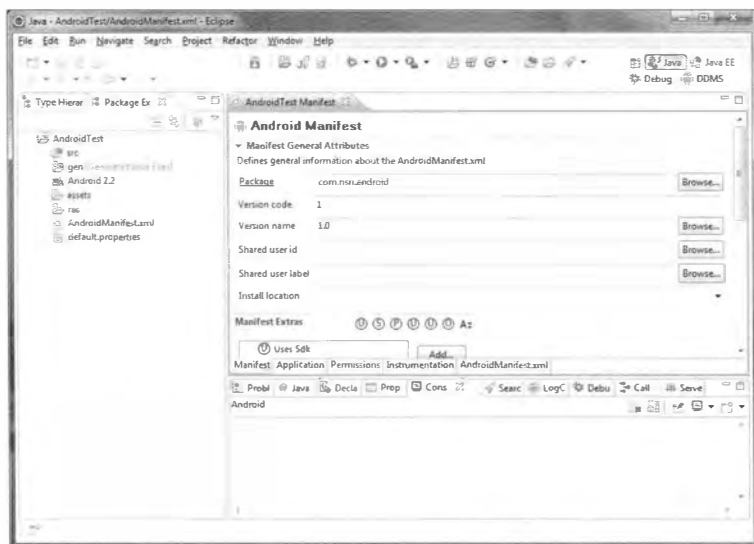
- ✦ *adb devices*: számítógép által látható Android-eszközök listája (az emulátort is listázza),
- ✦ *adb shell*: shellparancssor nyitása.

Az emulátor bal felső sarkában látható egy portszám is, amelyen az emulátor figyel, és különféle parancsok küldhetők neki. Ha például egy Telnet-klenssel rácsatlakozunk, akár SMS-t és hanghívást is szimulálhatunk:

- ✦ *sms send* <küldő száma> <üzenet>
- ✦ *gsm call* <hívó száma>

1.6.2. A fejlesztőkörnyezet használata

Android-alkalmazások fejlesztéséhez tehát az Eclipse fejlesztőkörnyezetet használjuk. Az Eclipse közkedvelt fejlesztő környezet Java-, Java EE-, C++- és egyéb projektek fejlesztéséhez, sőt akár LaTeX-projektek kezelésére is használható. Létezik Windows-, Linux- és Mac OS-verzió egyaránt. Számos pluginnal kiegészíthető, így a fejlesztők rugalmasan testre szabhatják. Az Android SDK és az Eclipse plugin megfelelő telepítése után a fejlesztőkörnyezet képes az elkészült projekteket egy emulátorban elindítani, ez pedig nagymértékben megkönnyíti a fejlesztést. A fejlesztőkörnyezet támogatja az USB-vel csatlakoztatott készülékeken való fejlesztést is, továbbá az on-device debug is támogatva van.



1.14. ábra. Android-projekt szerkesztése Eclipse-ben

Az Eclipse fejlesztői környezet felhasználói felületére nagymértékben jellemző a rugalmasság, hiszen minden segédablak áthelyezhető és átméretezhető. Ez a szabadság azonban néha problémákkal, illetve nem várt működéssel is jár. A következőkben az alapértelmezett elrendezés szerint mutatjuk be a környezetet.

- Tipikusan bal oldalt található a Project Tree, ahol a Workspace-ben megnyitott projektek találhatók.
- A fenti menüsorban az Eclipse-re vonatkozó funkciók érhetők el.
- Középen a szerkesztőnézet található, ahol az éppen megnyitott állomány jelenik meg, vagy éppen egy megfelelő dizájn.
- Az Eclipse különféle olyan perspektívákat támogat, amelyek a teljes elrendezést és a megnyitott ablakokat fogják egybe. Tipikus példa a Debug-perspektíva, amely csak a debugoláshoz szükséges ablakokat jeleníti meg, hogy a fejlesztő azonnal láthassa a szükséges információkat.
- Android esetében a DDMS-perspektívát szokás még használni, ahol az emulátor vagy a mobilkészülék beállításai, jellemzői érhetők el. Például: fájlrendszer, aktuális pozíció stb.
- Az alkalmazások futtatása és a kódkiegészítés a megszokott módon működik, az emulátor első indításkor eléggé lassú.

A fentiekből látható az Eclipse fejlesztőkörnyezet számos előnye, amely az évek során sokat fejlődött a fejlesztői közösség véleményei alapján. A fejlesztőkörnyezet rendkívül sok, jól használható billentyűkombinációt (*hotkey*) támogat, ezek alkalmazását feltétlenül javasoljuk például a fejlesztői idő lecsökkentésére.

Néhány gyakran használt billentyűkombináció a következő:

- *Ctrl + click*: navigálás a kódban
- *Ctrl + PageUP/PageDown*: navigálás a megnyitott állományok között
- *Ctrl + Bal/Jobb*: ugrás a kód korábban használt szakaszára
- *Ctrl + I*: Eclipse automatikus javaslat az adott helyen
- *Ctrl + space*: IntelliSense és „kódgenerálás”, ha egy felüldefiniálandó metódus első pár betűjét ütjük le, majd Entert nyomunk
- *Alt + Shift + R*: átnevezés (mindenhol)
- *Ctrl + Shift + T*: típus keresése
- *Ctrl + Shift + R*: erőforrás keresése
- *F3*: deklaráció megnyitása metóduson, osztályon
- *Ctrl + Alt + H*: híváshierarchia megjelenítése
- *Ctrl + Shift + O*: importfix,

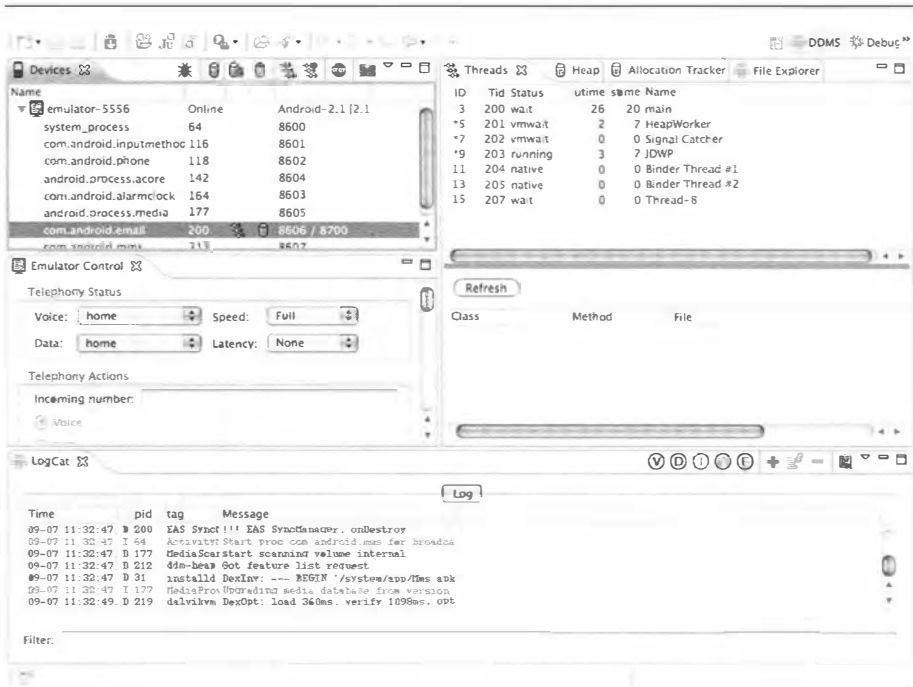
- *Ctrl + Shift + F*: forrás formázása
- *Ctrl + F6*: forrásfájlok közti váltás
- *Ctrl + Shift + G*: használat listázása
- *Alt + Shift + A*, majd *S*: szöveges konstans kihelyezése a szöveges értékeket tartalmazó Android *strings.xml* állományba
- *Ctrl + Shift + L*: gyorsbillentyű (hotkey) lista

Emellett az Eclipse használatakor javasoljuk az úgynevezett *Save Actions* beállítását, ezekben megadhatjuk, hogy mi történjen egy adott forrásállomány elmentésekor (például automatikus importfix, formázás stb.).

Új Android-mobilalkalmazás létrehozásához válasszuk a „New -> Android project” funkciót, amely alapértelmezetten egy *HelloWorld* alkalmazást generál a megfelelő állományokkal és könyvtárstruktúrával. A futtatás szokásos módon a „Run” paranccsal történik, ennek hatására alapértelmezetten elindul az emulátor. Az emulátor használatához elsőként létre kell hoznunk egy Android Virtual Device-t (AVD) az Android SDK- és AVD Manager-alkalmazásban. Létrehozáskor az a legfontosabb, hogy megadjuk az virtuális eszköz nevét, az emulált SD-kártya méretét, valamint a célfelbontást. Ha USB-kábellel csatlakoztatunk egy androidos készüléket a számítógéphez, lehetőség van az alkalmazás azonnali telefonos tesztelésére is, ha a telefon beállításaiban engedélyezzük az „USB-hibakeresés” funkciót.

Az Eclipse plugin feltelepítése után egy úgynevezett Dalvik Debug Monitor Server- (DDMS) perspektíva is elérhetővé válik, amely számos eszközt ad a fejlesztők kezébe:

- debugeszköz az Eclipse alá,
- port forwarding,
- képernyőképmentés,
- szál-, heap- és egyéb memóriainformációk,
- *LogCat* nézet,
- processzfelügyelet,
- mobilrádió-állapot,
- hívás és SMS kezelése,
- helymeghatározás,
- stb.



1.15. ábra. DDMS-perspektíva

MÁSODIK FEJEZET

Az Android-alkalmazások felépítése

Ebben a fejezetben bemutatjuk az Android-alkalmazások futási környezetét és a négy Android-alkalmazás-komponenst, amelyek különböző esetekre nyújtanak megoldásokat. Ezt követően ismertetjük az *Activity* komponens életciklusát, amelyre a legtöbbször szükségünk lesz. A fejezet végén bemutatjuk egy egyszerű Android-projekt szerkezetét, majd egy összetettebb példát is mutatunk Activityk közti együttműködésre. A többi komponensre részletesen a későbbi fejezetekben térünk ki.

2.1. Android-alkalmazás-környezet

Napjainkban megfigyelhető, hogy a mobilalkalmazások egyre népszerűbbek. Érdekes tendencia, hogy a korábban mobil eszközökre optimalizált webes megoldások helyett is egyre több mobilkliens jelenik meg. Egy általános mobilalkalmazásra az jellemző, hogy a készülék menüjéből valamilyen parancsikont kiválasztva elindítható. Az Android esetében azonban egy mobilalkalmazás ennél sokkal összetettebb lehet, gondoljunk például a háttérben futó szolgáltatásokra vagy a különféle események hatására induló komponensekre.

Telepítés után minden általános Android-alkalmazás egy saját biztonságos környezetben fut (security sandbox). Minthogy az Android olyan Linux-alapú operációs rendszer, amely támogatja a többfelhasználós viselkedést, így tulajdonképpen minden alkalmazás külön felhasználónak is tekinthető. Alapértelmezetten a rendszer minden alkalmazáshoz egy egyedi Linux felhasználói azonosítót rendel. Minden processznek saját virtuálisgép-példánya van, így az alkalmazás kódja elkülönülten fut a többi alkalmazástól. Az Android ennek megfelelően akkor indítja el a processzt, amikor az alkalmazás valamelyik komponense elindul, és akkor zárja be, amikor már nincs rá szükség, vagy a rendszernek memóriát kell felszabadítania.

Az Android memóriakezelése rendkívül összetett. A rendszer alapfelfogása, hogy alapértelmezetten engedi a háttérben is futni az alkalmazásokat, készenlétben tartja őket, hogy a felhasználó számára minél gyorsabban elérhetőek legyenek. Ám ha a rendszer rendelkezésre álló memóriája csökkenne, akkor egy jól definiált prioritási sort figyelembe véve az Android leállítja a kevésbé fontos processzeket, és az előtérben levő alkalmazások zökkenőmentes futtatására törekszik.

Az Android az alkalmazások kezelése szempontjából a „szükséges legkevesebb jogosultság” elvét igyekszik érvényesíteni, ez abban nyilvánul meg, hogy minden alkalmazás alapértelmezetten csak azokhoz a komponensekhez fér hozzá, amelyek a feladatához szükségesek. Ennek következménye az, hogy az Android biztonságos környezetet biztosít az alkalmazásoknak, védve mind az alkalmazást, mind pedig a rendszert. Az alkalmazások tehát nem férhetnek hozzá olyan rendszerszolgáltatásokhoz, amelyekhez nincs jogosultságuk. Ennek kezelésére az Android-alkalmazás fejlesztésekor az alkalmazás leírójában a fejlesztő felelőssége megjelölni azokat az engedélyeket, amelyekre az alkalmazásnak szüksége lesz. Telepítéskor a felhasználónak ezeket jóvá kell hagynia.

2.2. Az Android-alkalmazás komponensei

Android platformon egy alkalmazás több különböző típusú komponensből épülhet fel, amelyek közül egyet is elég az alkalmazásnak tartalmaznia. Az egyes komponenstípusok különböző célt szolgálnak egy alkalmazáson belül, és legfontosabb jellemzőjük, hogy egyedi területet biztosítanak, amelyen keresztül a rendszer el tudja indítani őket. Fontos még kiemelni, hogy nem minden belépési pont érhető el közvetlenül a felhasználók által. Egy Android-alkalmazás képességeit legegyszerűbben úgy érthetjük meg, ha megvizsgáljuk a tartalmazott komponensek típusát és szerepét.

Az Android négy fő alkalmazáskomponens-típust támogat, amelyek eltérő életciklusmodellel rendelkeznek. Ezek az életciklusmodellek definiálják a komponensek létrehozását és megsemmisülését.

Az Android-alkalmazás komponenstípusai a következők:

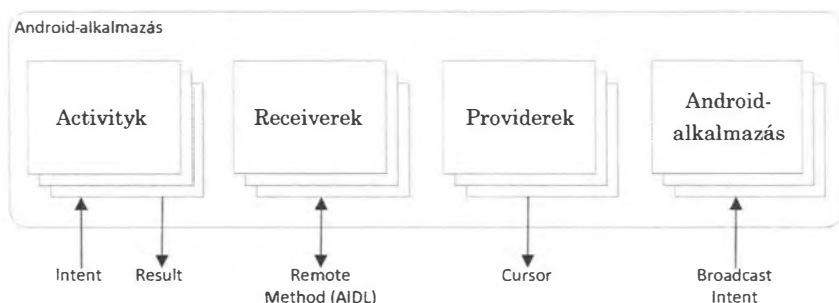
- *Activity*,
- *Service*,
- *ContentProvider*,
- *BroadcastReceiver*.

Az Android-alkalmazás ugyanabból a komponenstípusból több komponenst is tartalmazhat.



2.1. ábra. Android-alkalmazás komponenstípusai

Minden komponens különböző feladatokat lát el, és bármelyik komponens önállóan is aktiválódhat egy alkalmazáson belül anélkül, hogy a többi komponens ez befolyásolná. Emellett az is gyakran előfordul, hogy egy másik alkalmazásból aktiválódik az alkalmazás valamelyik komponense. A következő ábra az Android-alkalmazás-komponensek kapcsolódási pontjait jelöli.



2.2. ábra. Android-alkalmazás-komponensek kapcsolódási pontjai

Minden Android-alkalmazás rendelkezik egy úgynevezett *manifest* XML-állománnyal, amelyben az alkalmazás képességei, engedélyei, valamint komponensei vannak felsorolva (részletesen lásd később). A következőkben először röviden bemutatjuk az egyes alkalmazáskomponensek funkcióját, majd az Activity komponenst ismertetjük részletesen. (A többi komponenst lásd a következő fejezetekben.)

2.2.1. Activity

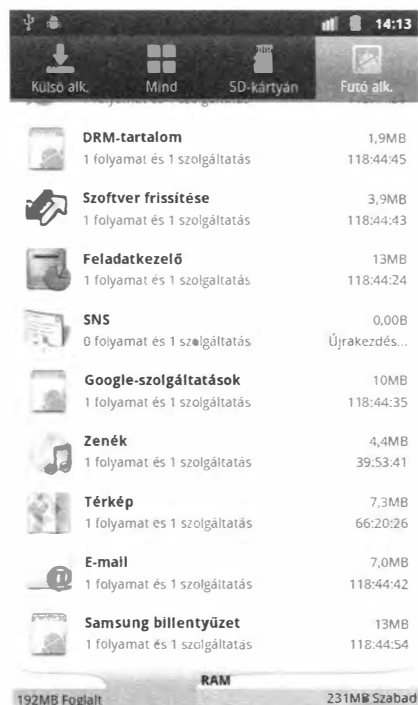
Az Activity a legtöbbször használt alkalmazáskomponens. Saját, önálló fellettel rendelkezik, de tipikus, hogy egy alkalmazáson belül több Activity is található, amelyek különböző, jól elhatárolt célokat szolgálnak, de együtt adják az alkalmazás teljes funkcionalitását.

Példaként nézzünk meg egy *ToDo* kezelőalkalmazást, ahol a kezdőnézet egy lista a teendőkkal, ahonnan elérhetjük az „új teendő felvitele” és a „teendőrészetek” nézetet. Ezek a nézetek jelentősen különböznek egymástól, mégis együtt valósítják meg a teljes funkcionalitást - ez jó példa egy több Activityből álló alkalmazás illusztrálására.

Az Android lehetőséget ad arra, hogy egy alkalmazás elindítson egy másik alkalmazásban található Activityt, így a platform kellő rugalmasságot és átjárhatóságot biztosít az alkalmazások között. Erre a viselkedésre tipikus példa a következő: ha egy fényképet kell készítenünk az alkalmazásunkban, akkor leggyakrabban a beépített fényképező alkalmazás megfelelő Activityjét használjuk. A platform efféle rugalmassága bizonyos szempontból a hatékony fejlesztést is támogatja, hiszen ha tudjuk, hogy egy másik alkalmazás már tartalmaz egy Activityt arra a feladatra, amelyre éppen szükségünk van, akkor ezt minden további nélkül használhatjuk.

2.2.2. Service

A *Service* komponens egy hosszabb ideig a háttérben futó feladatot jelképez. Legfőbb jellemzője, hogy nincs különálló felhasználói felülete, de természetesen indíthat Activityket, vagy megjeleníthet más „felugró” felületi elemeket (pl. *Notification*, *PopUp* stb.). A rendszer alapértelmezetten is több különböző szolgáltatást futtat a háttérben, ez biztosítja a megfelelő működést, a háttérben futó funkciókat. Érdekes, hogy a rendszerbeállítások menüjéből is megnézhetők az aktuálisan futó szolgáltatások, ez jól mutatja a rendszer nyíltságát.

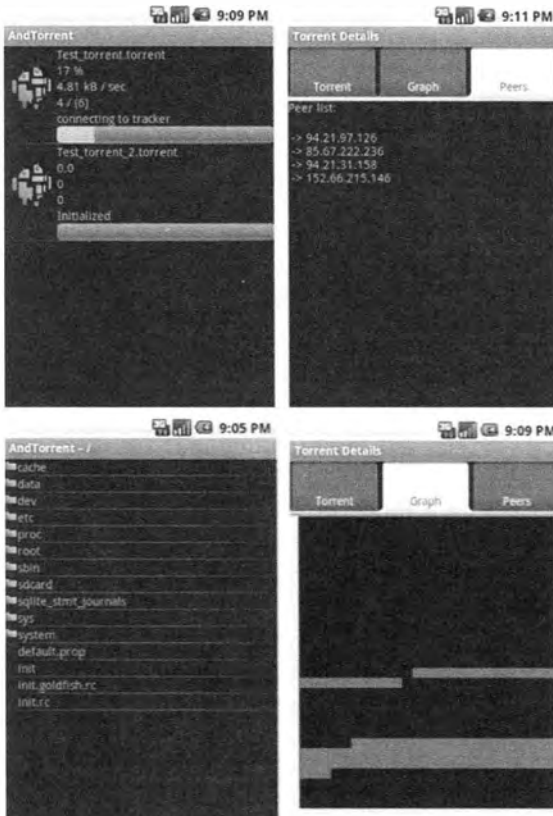


2.3. ábra. Futó szolgáltatások

A tipikus funkciók, amelyeket szolgáltatásokkal szokás megvalósítani, a következők:

- ✦ háttértartalom szinkronizálása,
- ✦ pozíció/használat nyomonkövetése,
- ✦ zenelejátszás,
- ✦ nagyobb tartalomletöltések.

A szolgáltatások használatával kapcsolatban érdemes megemlíteni a Budapesti Műszaki és Gazdaságtudományi Egyetem Automatizálási és Alkalmazott Informatikai Tanszékén fejlesztett *AndTorrent*⁵ klienst, amely egy teljes értékű BitTorrent-implementáció. Az *AndTorrent* esetében például a torrenttartalmak kiszolgálása (le- és feltöltés) tipikus háttérben futó feladat, hiszen a felhasználók nem kívánják folyamatosan nyomon követni a le- és feltöltés állapotát, hanem csak a végállapotról szeretnének értesítést kapni, így közben az Android-eszközt bármilyen más feladat elvégzésére használhatják.



2.4. ábra. *Service-példa: AndTorrent*

Android platformon a szolgáltatások szerepe rendkívül fontos, a fejlesztésük azonban nagy körültekintést igényel, mivel a háttérben futó képességük miatt könnyen írhatunk olyan alkalmazást, amelyik feleslegesen nagy adatforgalmat generál, és például túl gyorsan lemeríti a készülék akkumulátorát. (A Service-fejlesztéssel kapcsolatban részletesebben lásd a Service-ekről szóló fejezetet.)

⁵ Erdődy-Nagy Zsombor, Ekler Péter: Efficient Content Sharing on Android Platform, XXIV. MicroCAD International Scientific Conference. Miskolc, Magyarország, 121–126, 2010.

2.2.3. *ContentProvider* komponens

A *ContentProvider* komponens tartalomszolgáltató komponensként értelmezhető, amelynek feladata egy adatforrás kezelése és az adatforrásra vonatkozó kérések kiszolgálása. A komponens összetettsége abból ered, hogy az adatforrás bármi lehet, kezdve a készüléken lévő adatbázistól a weben keresztül elérhető RSS-ig.

A *ContentProvider* feladata tehát az, hogy a kérést intézők elől elfedje az adatforrás összetettségét és elhelyezkedését, illetve hogy a megfelelő sebességű kommunikációt bonyolítsa le az adatforrással, és az adatokat visszaadja a kérést indító fél számára. A rendszeren belül minden *ContentProvider* egy egyedi URI-azonosítóval rendelkezik, amelyen keresztül akár más alkalmazásból is elérhető. Az Android rendszer beépítve is tartalmaz számos *ContentProvidert*, elegendő például a névjegyzékre vagy a hívási naplóra gondolni. (A *ContentProvider*ek használatát részletesebben lásd később a használatukra vonatkozó konkrét példákkal.)

A *ContentProvider*ek használatára jó példa, ha a korábban említett *ToDo* alkalmazást egészítjük ki egy *ContentProvider* komponenssel, amely a tennivalóink listáját teszi elérhetővé más alkalmazások számára. Így például egy közösségi hálózati kliens hozzáférhet a tennivalóinkhoz, és megoszthatja az általunk engedélyezett tennivalókat másokkal.

2.2.4. *BroadcastReceiver* komponens

A *BroadcastReceiver* komponensnek az a feladata, hogy a különféle események hatására aktiválódjon, és valamilyen feladatot végrehajtson. Az Android rendszer számos eseményt jelez *broadcast*ok (sugárzás) formájában, amelyekre feliratkozhatunk a saját alkalmazásból. Ilyen esemény lehet például az alacsony akkumulátorszint jelzése, egy elkészült fotó vagy akár egy bejövő hívás is. A *BroadcastReceiver* jellemzője az, hogy nem rendelkezik saját felhasználói felülettel, hanem például megjeleníthet valamilyen figyelmeztetést, vagy elindíthat egy másik komponenst.

Amikor tehát egy *broadcast* esemény bekövetkezik, az Android megvizsgálja, hogy mely alkalmazások tartalmazznak olyan *BroadcastReceiver* komponenst, amelyet ez az esemény érinthet, és elindítja ezeket a komponenseket. A komponens viselkedésének megvalósítása már teljes mértékben a fejlesztő feladata. Ilyenkor a saját *BroadcastReceiver* például elindíthat egy szolgáltatást, vagy megjeleníthet valamilyen üzenetet. Az Android lehetőséget biztosít saját *broadcast* esemény indítására is, amelyek küldéskor ugyanúgy viselkednek a rendszerben, mint a beépített események.

A *BroadcastReceiver*ek működéséhez képzeljünk el egy olyan alkalmazást, amely egy szolgáltatás segítségével folyamatosan nyomon követi a készülék pozícióját, és bizonyos időközönként elküldi egy szerverre. Ebbe az alkalmazásba érdemes egy *BroadcastReceiver* komponenst ágyazni, amely alacsony akkumulátorszint esetén például ritkábban ellenőrzi a pozíciót, és ritkábban kommunikál a szerverrel, hogy csökkentse az energiahasználatot.

Az Android rendszer sajátossága, hogy egy alkalmazás el tudja indítani egy másik alkalmazás komponensét is, ezzel valósul meg a rendszer magas fokú rugalmassága. Egy fénykép készítéséhez például nem kell új Activityt készíteni, hanem elegendő az előre telepített kameraalkalmazás képalkészítő Activityjét meghívni, ahol sikeres fényképezés után a képet megkapja a hívóalkalmazás komponense. Ennek a mechanizmusnak köszönhetően a felhasználó úgy érzi, mintha a kamerafunkció az alkalmazás része lenne.

Amikor a rendszer új komponenset indít, először megnézi, hogy fut-e a komponens tartalmazó alkalmazás processze, és ha nem, akkor elindítja. Ezt követően megtörténik a szükséges osztályok példányosítása. Például az előző kamerapélda esetén az egész kameraalkalmazást fel kell éleszteni, ha még nem futott. Látható tehát, hogy Androidban egy alkalmazásnak több belépési pontja is lehet, nincs egy egyszerű *main()* függvény, mint a megszokott Java- vagy C++-alkalmazásoknál.

Egy komponens indításához a rendszer felé egy üzenetet/szándékot (*Intent* – lásd később) kell intéznünk, ennek hatására végrehajtódik az a kérés, amelyre válasz is érkezhethet, és amely természetesen megfelelően eljut az indító félhez.

2.3. Az Android-alkalmazás felépítése fejlesztői szemszögből

Egy Android-alkalmazás forrása fejlesztői szempontból a következő fő csoportokba osztható:

- Metainformációk: főként az alkalmazás projektleíró *manifest* állománya
- Erőforrások: XML-alapú felhasználói felületi elemek, XML-alapú nyelvi állományok, egyéb XML-erőforrások, valamint különféle multimédia-elemek (képek, hangok stb.)
- Forráskód: Java-alapú forrás (vagy C/C++ NDK használata esetén)
- Külső osztálykönyvtárak: lefordított könyvtárak, amelyeket az alkalmazásunk forrása felhasznál

2.3.1. A *manifest* állomány bemutatása

A *manifest* állomány legfőbb szerepe az alkalmazás komponenseinek definiálása. Egy XML-állományról van szó, amelynek alapján a rendszer tudja, hogy egy adott alkalmazás milyen komponenseket tartalmaz, és milyen külső események lehetnek számára relevánsak. Emellett a *manifest* állományban található az alkalmazás futtatásához szükséges minimális követelmények (például minimális Android-verzió), valamint azok a jogosultságok, amelyek

az alkalmazás futtatásához szükségesek (pl. internetelérés, helymeghatározás stb.). Összefoglalva a *manifest* tartalma a következő:

- csomagnév (alkalmazásonként egyedi),
- verzió,
- minimális API-verzió (*uses-sdk*),
- engedélyek (*uses-permission*),
- külső könyvtárak használata (*uses-library*),
- alkalmazáskomponensek listája,
- jelzés arra, hogy mely alkalmazáskomponens milyen eseményt kezel (*intent-filter*).

A következőkben tekintsük át egy egyszerű, egy Activityből álló Android-alkalmazás *manifest* állományát.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="hu.bute.daai.amorg.examples"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="7" />

    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".ActivityMain"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name=
                    "android.intent.action.MAIN" />
                <category android:name=
                    "android.intent.category.LAUNCHER"
            />
        </intent-filter>
    </activity>
    </application>
</manifest>
```

A *manifest* egy olyan szabványos XML-állomány, amelynek a gyökéreleme a *manifest*. A *manifest* attribútumai között található a csomagazonosító (*package*), amely alatt az Eclipse-projektben az alkalmazás forrása helyezkedik el (Java-osztályok). Ezt követik a verzió- és verziónév-azonosítók.

A *manifest* elemen belül található a különféle jogosultságok leírása. Példánkban a 7-es kódnevű SDK-verziót jelöltük meg, amely az Android 2.1-es platformnak felel meg. Az engedélyek felsorolása után található az *application* elem, amelynek attribútumaiban definiálnunk kell az alkalmazás ikonját és nevét, ezek példánkban az erőforráskönyvtárban lévő képre és egy szöveges erőforráselemre hivatkoznak. Az *application* elemen belül helyezkednek el az alkalmazásban megvalósított komponensek, ez példánkban az egyetlen Activity. A használható komponenselemek a következők:

- `<activity>`
- `<service>`
- `<provider>`
- `<receiver>`

Az Activity elem *android:name* attribútuma megadja a komponenst megvalósító osztályt, az *android:label* elemben pedig a komponens látható neve definiálható. A komponenselemen belül az *intent-filter* elemekben definiálhatók a komponens egyéb tulajdonságai, illetve az, hogy milyen külső eseményekre ébredjen fel (pl. bejövő hívás egy *BroadcastReceiver* komponens esetén). Az `<action>` tag jelzi, hogy ez az Activity az alkalmazás fő belépési pontja, míg a `<category>` tag azt jelzi, hogy az Activity jelenjen meg az indítható alkalmazások listájában.

A *manifest* állomány szerkezetének áttekintéséhez nézzünk meg egy bonyolultabb példát. A példában az új elemeket vastag betűvel jelöltük. Példánkban egy olyan alkalmazás *manifest* állományát adjuk meg, amely megjeleníti az aktuális pozíciókat egy térképen.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="hu.bute.daai.amorg.examples"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name=
        "android.permission.INTERNET"/>
    <uses-permission android:name=
        "android.permission.ACCESS_FINE_LOCATION"/>

    <application android:icon="@drawable/icon"
        android:label=
            "@string/app_name">
        <uses-library android:name="com.google.
            android.maps"/>
```

```

<activity android:name=".ActivityMyMap"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name=
            "android.intent.action.MAIN"/>
        <category android:name=
            "android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
</application>
</manifest>

```

Az egyik változás a *uses-sdk* elem alatt szereplő két *uses-permission* elem, amelyek az internethasználatot (*android.permission.INTERNET*) és a pontos helymeghatározási (*android.permission.ACCESS_FINE_LOCATION*) engedélyeket jelképezik. A másik változtatás az *application* elemen belül található *uses-library* elem, amely a külső, Google-térkép-könyvtár használatát jelzi.

2.3.2. Erőforrás-állományok

Az Android platform egyik legnagyobb előnye, hogy fejlesztői szinten lehetővé teszi a forráskód (üzleti logika) és a felhasználói felület kettéválasztását. Az Android platformon egy rendkívül jól használható módszert dolgoztak ki a felhasználói felület és minden egyéb erőforrás-jellegű elem rugalmas kezelésére. Minden Android-projekt tartalmaz egy *res* elnevezésű erőforráskönyvtárat, amelybe XML-ben leírt erőforrások és egyéb médiaelemek (képek, hangok stb.) helyezhetők el. A teljesség igénye nélkül az alábbi erőforráselemeink lehetnek:

- képek (preferált formátum: *png*),
- hanganyagok,
- UI-elrendezések (layout),
- animációk,
- menük,
- témák, stílusok,
- szöveges erőforrás-állományok (többnyelvűség támogatásához).

Az erőforrások használatakor a fejlesztőkörnyezet automatikusan frissít egy *R.java* állományt, amelyen keresztül a Java-forráskódból elérhető minden azonosítóval ellátott erőforráselem. Az *R.java* állományban tulajdonképpen statikus *int* típusú változók találhatók, amelyek minden erőforráshoz egy egyedi értéket rendelnek. Az *R.java* állományt soha ne módosítsuk, ezt

minden esetben a rendszer kezeli. Ha úgy látjuk, hogy nem frissült új erőforrás felvételekor, ellenőrizzük az erőforrás-állományokat, ugyanis hibás szintaxis esetén a rendszer nem frissíti az *R.java* állományt.

Az erőforrásokról részletesebben a következő fejezetben szólnunk, a következőkben csak egy egyszerű példát mutatunk be egy egyszerű felhasználói felület XML-es leírására.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/txtWelcome"/>
    <EditText
        android:id="@+id/editTextUserName"
        android:hint="@string/hintName"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <requestFocus />
    </EditText>
    <Button
        android:id="@+id/btnSayHello"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnSayHello"/>
</LinearLayout>
```

Az XML-ben megadott felületi elrendezést a következő képernyőkép szemlélteti:



2.5. ábra. Felhasználói felület erőforrás alapján

A fenti erőforrás alapján generálódott *R.java* állomány tartalma a következőkben látható. (A későbbi példák során az *R.java* tartalmát már nem tüntetjük fel, hiszen a tartalmát mindig a rendszer generálja.)

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found. It
 * should not be modified by hand.
 */

package hu.bute.daa1.amorg.examples;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int btnSayHello=0x7f050001;
        public static final int
        editTextUserName=0x7f050000;
    }
}
```

```

    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int btnSayHello=0x7f040003;
        public static final int hintName=0x7f040002;
        public static final int
txtHelloText=0x7f040004;
        public static final int txtWelcome=0x7f040001;
    }
}

```

2.3.3. Forráskód

Sajnos a felhasználók általános véleménye az Android-alkalmazásokról nem minden esetben pozitív. A Marketen való publikálást a Google nem ellenőrzi komolyabban az alkalmazásminőség szempontjából, ezért sok lassú vagy hibásan működő alkalmazást publikálnak, és ez rontja a platform megítélését. Android platformra való fejlesztéskor mindenképpen javasoljuk az ismert Java-konvenciók és hatékonysági módszerek betartását, hogy minél egységesebb, hatékonyabb és átláthatóbb kódot készíthessünk. A következőkben a legfontosabb konvenciókat ismertetjük.

2.3.3.1. Kivételkezelés

A kivételkezeléssel kapcsolatban a legfontosabb alapszabály az, hogy ne hagyjuk figyelmen kívül a kivételeket.

```

void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) { }
}

```

A fenti kódrész például nagyon veszélyes, hiszen hibás érték esetén semmilyen tájékoztatást/visszajelzést nem kap a hívó fél. Ha előre tudjuk, hogy csak számértéket kaphat a függvény, a kód akkor sem lesz jó, hiszen előfordulhat, hogy később egy másik fejlesztő máshonnan hívja meg a függvényt, és nem számol a hibakezeléssel.

Ehelyett alternatív megoldásként jelezhetjük a függvényben, hogy az kivételt dobhat, például:

```
void setServerPort(String value) throws
NumberFormatException {
    serverPort = Integer.parseInt(value);
}

Vagy további megoldás egy saját Exception osztály
használata, például:

void setServerPort(String value) throws
ConfigurationException {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        throw new ConfigurationException("Port " +
            " érték nem érvényes.");
    }
}
```

Egy másik, már kevésbé ajánlott lehetőség, hogy hiba esetén egy alapértelmezett értéket állítunk be.

```
/** Port beállítás, hiba esetén alapértelmezetten a
80-as port kerül tárolásra. */
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        serverPort = 80; // alapértelmezett server
port
    }
}
```

Legvégső esetben, ha úgy tervezzük, hogy mindenképp figyelmen kívül hagyjuk a hibát, akkor kommentben tüntessük fel a döntésünk megfelelő magyarázatát, például:

```
/** Hibás érték esetén nincs változtatás. */
void setServerPort(String value) {
    try {
```



```

        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        // Hibás bemenet esetén nem változik meg
        // a serverPort értéke.
    }
}

```

A kivételkezelés egy másik tipikus hibája, ha minden általános kivétel elkapása egy *catch* ágban történik. Ügyelni kell arra, hogy ilyen ne szerepeljen a kódban, például:

```

try {
    someIOFunction(); // IOException-t dobhat
    someParsingFunction(); // ParsingException-t dobhat
    someSecurityFunction(); // SecurityException-t dobhat
} catch (Exception e) {
    handleError(); // hibásan egy hibakezelő
}

```

Nagyon veszélyes az általános hibakezelés, hiszen legtöbb esetben a különböző hibákat különbözőképpen kell kezelni, és máshogy kell reagálni rájuk. Továbbá ugyancsak rossz stratégia, ha az *Exception*nél általánosabb *Error* vagy *Throwable* objektumot kapunk el a *catch* ágban, hiszen ezek más jellegű hibákat jelentenek, és nem lehet általánosan kezelni őket.

Néha azonban kellő indoklás (komment) mellett előfordulhat általános hibakezelés a kódban, ha például semmiképp nem szeretnénk egy hibát eljuttatni a felhasználói felületre, vagy ha egy *batchet* mindenképp le kell futtatni.

Az ajánlott módszerek az általános hibakezelés kiküszöbölésére a következők:

- Minden kivételt külön *catch* ágban kezeljünk (kerüljük a sok kódDuplikálást).
- Javítsuk/finomítsuk a kódot, hogy több *try-catch* ág legyen. Tipikusan érdemes az I/O műveleteket különválasztani a feldolgozástól.
- A kivételek tovább dobása valósuljon meg.

2.3.3.2. *Finalizerek* kerülése

A *finalizerek* segítségével egy kis kódrészt futtathatunk le akkor, amikor az adott objektumot a Garbage Collector (GC) felszabadítja. Általában azzal szoktak hibásan érvelni a *finalizerek* használata mellett, hogy segítségükkel

például könnyű erőforrásokat felszabadítani. Ám szem előtt kell tartanunk, hogy nincs arra garancia, hogy mikor és hogy egyáltalán lefut-e a finalizer, így nem érdemes számítani rá.

2.3.3.3. *Importok kezelése*

Csomagok importálásakor kétféle módszert választhatunk: vagy importálunk egy teljes csomagot, vagy megadjuk azt a pontos osztályt, amelyet importálni szeretnénk.

```
import foo.*;
```

Vagy:

```
import foo.Bar;
```

Teljes csomag importálása mellett az szokott az érv lenni, hogy ekkor az importkódrész rövidebb. Ám ha kiírjuk az osztályneveket, pontosan lehet látni, hogy az adott osztály milyen más osztályokat használ.

A fentiek alapján tehát a teljes osztály importálást javasoljuk. Kivételt jelentenek ez alól a *java.util.**, a *java.io.** és hasonló általános csomagok.

Az importcsomagok feltüntetését az alábbi sorrendben javasoljuk:

1. Android-csomag-importok,
2. külső csomagok importálása (*com*, *junit*, *net*, *org* stb.),
3. Java-csomagok importálása.

A fenti csoportok elkülönítéséhez egy üres sort javasolunk az importszakaszban. Ezek betartásával a legfontosabb csomagok felülre kerülnek, így bármilyen fejlesztő láthatja, hogy milyen Android-specifikus célt szolgálhat az adott osztály.

2.3.3.4. *Kódkommentezés: JavaDoc*

Minden Java-állomány praktikus, ha az elején tartalmaz egy copyright leírást, majd azt követi a *package*-név és az importok. Ezután egy *JavaDoc* stílusnak (*/** ... */*) megfelelő komment javasolható, amely egy-két mondatban leírja az adott osztály szerepét. Például:

```

/*
 * Copyright (C) 2012 ...
 *
 * License verzió ...
 *
 */

package hu.bute.daai.amorg.examples;

import android.os.Blah;
import android.view.Yada;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * Az osztály feladata az adatbázis kezelése
 */
public class MyDBHandler {
    ...
}

```

A kommentezéssel kapcsolatban további javaslat, hogy minden konstruktor és minden statikus és publikus metódus előtt legyen szabványos komment, amely legalább egy mondatban kifejti a függvény funkcióját.

```

/** Visszatér egy szám négyzetével. */
static double sqr(double num) {
    ...
}

```

A fenti konvenciók figyelembevételével könnyen lehet szabványos *javadoc*-ot generálni a projektekből. A kommentezéssel kapcsolatban javasoljuk még a szabványos Java-konvenciók betartását [39].

Gyakori, hogy a kódban úgynevezett *TODO* kommenteket helyeznek el a fejlesztők. A *TODO* kommentek használatát akkor javasoljuk, ha azok ideiglenes kódrészekre vagy jó, de nem tökéletes megoldásra utalnak.

Néhány tipikus példa:

```

// TODO: Ez a kódrész eltávolítható, ha majd a
// megfelelő média osztályt használjuk

```

Vagy:

```
// TODO: Legyen ez a változó később konstans
```

Mindazonáltal nem célszerű bőbeszédűen és feleslegesen kommentezni a kódot, érdemes inkább a beszédes függvény- és változóneveket használni, hiszen az *IntelliSense* ezeket jeleníti meg fejlesztés közben, és sokszor egy beszédes név nagyon nagy segítség a programozóknak.

2.3.3.5. A kód szerkezete

A függvényekkel kapcsolatban alapszabályként érdemes betartani, hogy minél rövidebb és minél inkább lényegre törő függvényeket készítsünk. Általánosságban javasolható, hogy egy képernyőnél hosszabb (~40 sor) függvények esetén érdemes átgondolni, hogy nem lehetne-e az adott metódust szétbontani anélkül, hogy sértenénk a kód integritását.

További javaslat a függvényekkel szemben, hogy érdemes azok ciklomatikus számát (a függvényben található ciklusok száma) minél alacsonyabban tartani (ideális esetben maximum 1).

A függvényekhez hasonlóan a kódsorokat is érdemes minél rövidebbre hagyni. Alapszabályként általában maximum 100 karakterre szokás a sorhosszokat méretezni, de természetesen kivételes esetekben ettől el lehet térni.

Az attribútumok definiálását érdemes az adott osztály legelején feltüntetni, hogy bárki könnyedén áttekinthesse, mik tartoznak az adott osztályhoz. Továbbá javasolható, hogy egy változó nevét érdemes annál beszédesebbre választani, minél nagyobb a láthatósága.

A változók elnevezésével kapcsolatban az a legfontosabb, hogy a projekt során következetesen ugyanazt a konvenciót használjuk. Érdemes például megfontolni a következő szabályokat:

- A nem publikus és a nem statikus változók *m* betűvel kezdődnek.
- A statikus változók *s* betűvel kezdődnek.
- A további mezők kisbetűvel kezdődnek.
- A *public static final* mezők végig nagybetűsek, és érdemes őket az osztály elején feltüntetni.

Például:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

Sokszor a változónév valamilyen rövidítést (mozaikszót) tartalmaz, utalva a változó szerepére. Ilyenkor azt javasoljuk, hogy a rövidítés betűit a szöveges írással ellentétben ne csupa nagybetűvel írjuk le a változók nevében, ugyanis így sokkal jobban olvashatók lesznek a változók. A következő táblázat néhány példát mutat.

2.1. táblázat. Mozaikszavak használata változónevekben

Java-solt	Nem java-solt
XmlHttpRequest	XMLHttpRequest
getCustomerId	getCustomerID
class Html	class HTML
String url	String URL
long id	long ID

További megjegyzés a változókkal kapcsolatban, hogy érdemes az érvényességüket/láthatóságukat minél kisebbre hagyni, így a kód sokkal biztonságosabbá válik, és a hibavalószínűség is kisebb lesz. Emellett javasoljuk, hogy a változók a használatukhoz igazodóan a lehető legbelsőbb blokkban legyenek definiálva.

Helyi változókat akkor érdemes deklarálni, amikor először történik meg a felhasználásuk. Általában minden helyi változó deklarációja tartalmaz valamilyen inicializálást. Ha az adott helyen nem lehet megfelelően inicializálni a változót, érdemes a deklarációt is későbbre helyezni.

A fenti szabály alól kivétel a *try-catch* blokk. Ha egy változót egy másik függvény visszatérési értékével inicializálunk, amely kivételt dobhat, akkor azt egy *try-catch* blokkon belül kell inicializálni, ám deklarálni valamilyen módon a blokkon kívül érdemes. Például:

```
// cl objektum egy Set halmazt reprezentál
Set s = null;
try {
    s = (Set) cl.newInstance();
} catch (IllegalAccessException e) {
    throw new IllegalArgumentException(cl + "nem elér-
hető");
} catch (InstantiationException e) {
    throw new IllegalArgumentException(cl +
    "nem példányosítható");
}

// Művelet a halmazon
s.addAll(Arrays.asList(args));
```

Ilyen esetben is javasolható azonban a *try-catch* szakaszt egy függvénybe ágyazni, például:

```
Set createSet(Class cl) {
    // Instantiate class cl, which represents some
    sort of Set
    try {
        return (Set) cl.newInstance();
    } catch (IllegalAccessException e) {
        throw new IllegalArgumentException(cl +
        "nem elérhető");
    } catch (InstantiationException e) {
        throw new IllegalArgumentException(cl +
        "nem példányosítható");
    }
}

...
// Művelet a halmazon
Set s = createSet(cl);
s.addAll(Arrays.asList(args));
```

Ciklusok esetén a változó deklarálása a *for* utasításon belül javasolható.

```
for (int i = 0; i < n; i++) {
    művelet(i);
}

illetve,

for (Iterator i = c.iterator(); i.hasNext(); ) {
    művelet(i.next());
}
```

A kód szerkesztése szempontjából az egyik legfontosabb követelmény a következetes és megfelelő szerkesztés, és ez nemcsak a kódolási stílusra, hanem a formázásra is vonatkozik. A kódon belüli behúzás az egyik legfontosabb tényező a jól áttekinthető forma eléréséhez. Behúzásokhoz javasoljuk a szokásos konzisztens használatát a forráskódokban, hiszen az minden operációs rendszeren ugyanúgy jelenik meg, ellentétben a tabulátorral. Továbbá a forráskód egy részét sokszor valamilyen dokumentumban is meg kell jeleníteni, és ez tabulátorok használatakor néha nehézkes.

2.3.3.6. Annotációk használata

A forráskódban mindenképp javasoljuk az annotációk használatát, amely átláthatóbbá és precízebbé teszi a kódolást.

Javasolt annotációk a következők:

- **@Deprecated:** A *@Deprecated* annotációt akkor kell használni, ha az adott kódrészt/metódust a fejlesztő elavultnak ítéli meg, és nem javasolja a használatát (tipikusan publikus függvényeknél fontos). A *@Deprecated* jelölés esetén a függvény JavaDoc-szabványú kommentjében egy *@deprecated* tagban meg kell nevezni az alternatív függvényt, amelyet a fejlesztő javasol. Továbbá mindig szem előtt kell tartani, hogy a *@Deprecated* jelző ellenére az adott függvény még mindig működik, és elérhető a többi fejlesztő számára, ezért helyesen kell működnie.
- **@Override:** Az *@Override* annotációt akkor kell használni, ha az adott metódus egy ősosztályban lévő metódust definiál felül.
- **@SuppressWarnings:** A *@SuppressWarnings* annotációt akkor kell használni, ha az adott warningot semmiképpen nem lehet elkerülni. Ilyenkor mindenképpen javasoljuk a használatát, hogy a kódban található többi warning minden esetben valós problémát jelezzon.

A `@SuppressWarnings` annotáció esetén feltétlenül javasoljuk egy `TODO` komment elhelyezését, amely megmagyarázza, hogy miért nem lehet ezt a warningot elkerülni. Például:

```
// TODO: Az Utility osztály rotate() függvénye
// generikus módon viselkedik
@SuppressWarnings("generic-cast")
List<String> blix = Utility.rotate(blax);
```

2.3.3.7. Naplózás

A naplózás sok esetben rendkívül hasznos, és megkönnyíti a hibák felderítését, valamint a hatékonyság fokozását, ám sokszor a feleslegesen bent felejtett naplózó kódrészek csökkentik a teljesítményt, és erre nagyon kell vigyázni.

Android platformon öt különböző naplózási szint létezik, ezeket a megfelelő módon kell használni. Naplózáshoz a `Log` osztály statikus `e()`, `w()`, `i()`, `d()`, `v()` függvényeit kell használni, ez az osztály a szabványos `LogCat` kimenetre naplóz.

- **ERROR:** A naplózásnak ezt a szintjét akkor javasolt használni, ha komoly problémát szeretnénk megfigyelni, olyat, amelynek a felhasználó által is látható következményei lehetnek, és nem lehet újra előidézni például valamilyen adattörlés vagy más komoly lépés elvégzése nélkül. Az **ERROR** szint naplózása mindig megtörténik.
- **WARNING:** A naplózásnak ezt a szintjét akkor érdemes használni, ha valami komoly és váratlan jelenség történt, és ez nagy valószínűséggel rekonstruálható adattörlés vagy más hasonló komolyságú lépés. A **WARNING** szint naplózása is mindig megtörténik.
- **INFORMATIVE:** Ezt a szintet akkor érdemes használni, ha jelezni szeretnénk, hogy valamilyen, több szempontból is érdekes esemény történt. Például valamilyen jelenség, amelynek széles skálán is lehet hatása, de nem feltétlenül hibát jelent. Ez a szint szintén mindig naplózódik.
- **DEBUG:** Ezt a szintet akkor érdemes használni, ha el szeretnénk tárolni, hogy milyen események történtek a készüléken, ez fontos lehet valamilyen hiba vagy jelenség felderítéséhez. Azt javasoljuk, hogy csak a valóban releváns információk naplózása valósuljon meg, hogy ne pazaroljuk feleslegesen az erőforrásokat. Ez a szint *release* fordítás esetén is naplózódik; ha olyan információkat naplózunk, ahol ezt szeretnénk elkerülni, a **VERBOSE** szintet érdemes használni. A **DEBUG** szint esetén érdemes valamilyen feltételtől függő

szerkezetbe helyezni a naplózó kódrészeket (például egy statikus *boolean* változó függvénye a tényleges naplózás).

- **VERBOSE:** A naplózásnak ez a szintje csak *debug* fordításkor valósul meg, és ajánlatos körülvenni például egy „*if (LOCAL_LOGV)*” blokkal (vagy hasonlóval) az alapértelmezett fordítás céljából, ahol a *LOCAL_LOGV* változó például egy *boolean* érték. Minden szöveg összeállítás ezen a blokkon belül érdemes elhelyezni, hiszen ez nem kerül be a *release* fordításba, így nem terheli feleslegesen a kiadásra szánt verziót.

A fenti, naplózásra vonatkozó javaslatok betartását mindenképp javasoljuk, mivel sokszor tapasztaltunk már hibás és nem hatékony működést naplózó kódrészek miatt.

A naplózás a következőképp történik:

```
Log.d("MYTAG", "Esemény naplózása...");
```

A függvény első paramétere egy egyedi *TAG* azonosító, amelyet érdemes egy konstansban tárolni, a második paraméter pedig az, amit naplózni szeretnénk. Az Eclipse-ben található *LogCat* nézetben lehetőség van *TAG*-ek alapján szűrni a naplózott adatokat, ez nagymértékben megkönnyíti az események/hibák visszakövetését, ha jól használjuk a *TAG*-eket (például alkalmazásonként egyedi *TAG*).

2.4. Activity-élelciklus és -környezet

Az *Activity* tulajdonképpen nem más, mint egy a képernyőn megjelenített oldal, amely a felhasználói interakciókat fogadja. Leginkább talán egy ablakként képzelhető el a PC-k világából ismert terminológia alapján, amely vagy teljes képernyősen, vagy pop-up jelleggel jelenik meg. Fontos azonban kiemelni, hogy Android platformon nincs klasszikus értelemben vett ablakkezelő rendszer, átlapolódó, áthelyezhető/méretezhető ablakok, de erre igazából nincs is szükség.

Egy Android-alkalmazás tipikusan több *Activity*ből áll, amelyek egymáshoz lazán vannak csatolva. Legtöbbször létezik egy „fő” *Activity*, amelyik indításkor megjelenik, és ahonnan valamilyen navigációs út mentén a többi elérhető. Bármelyik *Activity* indíthat újabbakat.

Ha egy új *Activity*t indítunk, az előző *Activity* leáll (*Stopped*), de a rendszer megőrzi az úgynevezett *Back Stack*en. Amikor az új *Activity* elindul, rákerül a *Back Stack*re, és megkapja a vezérlést (*focus*). A *Back Stack* tulajdonképpen egy „last in, first out” veremként fogható fel, amely a példányosított *Activity*ket tartalmazza, de mindig csak egy van aktív állapotban.

Az Android-eszköz beépített *Vissza (Back)* gombjának megnyomásakor a *stacken* legfelül lévő Activity kikerül a veremből, és az alatta lévő lesz aktív.

Amikor egy Activity leáll egy másik indulása miatt, mindkét Activity-oldal értesítést kap az eseményről az úgynevezett életciklus- (*callback*) metódusokon keresztül. Számos callback metódus támogatva van (*onCreate()*, *onStop()*, *onResume()*, *onDestroy()* stb.), ezekre az Activity megfelelően reagálhat. Például *onStop()* esemény hatására tipikusan a nagyobb objektumokat érdemes elengedni (adatbázis vagy hálózati kapcsolat). Amikor az Activity visszatér (*onResume()*), újra el kell kérni az erőforrásokat. Ezek az átmenetek tipikus részei minden Activity életciklusának, és nagyfokú körültekintést igényelnek a fejlesztés szempontjából.

Egy Activity 3 fő állapotban lehet, és minden állapotváltáskor a megfelelő életciklus- (*callback*) függvény hívódik meg, amelyet a fejlesztő felüldefiniálhat, és elvezetheti benne a szükséges feladatokat.

- *Resumed (running)*: az Activity előtérben van, és a focus rá irányul.
- *Paused*: az Activity él, de egy másik Activity előrébb van, ám ez még látszik (transparens a felső, avagy pop-up jellege miatt nem fedi el teljesen). A rendszer kivételesen alacsony memóriaállapot esetén felszabadíthatja az Activityt.
- *Stopped*: az Activity még él, de már egy másik Activity van teljesen előtérben, és a *Stopped* állapotban lévőből semmi nem látszik. Alacsony memóriaállapot esetén a rendszer felszabadíthatja az Activityt.

Egy Activity esetében sarkalatos kérdés, hogy jól felkészítettük-e az életciklus-állapotváltozásokra. A rendszer kevés memória esetén egy meghatározott prioritási elvet követve a kevésbé szükségesnek ítélt alkalmazáskomponenseket leállítja. A korábban említett *Paused* és *Stopped* állapotban lévő Activityk például ki vannak téve ennek a veszélynek. Kevés memória esetén a rendszer a *Stopped* állapotban lévő Activityket állítja le először, majd extrém kevés memória esetén a *Paused* állapotban levőket. Tapasztalataink szerint memóriahiány esetén a rendszer tipikusan nem az Activityket állítja le, hanem a teljes processzt, tehát még fontosabb az alkalmazás állapotának megfelelő tárolása.

Sokszor azonban már előre tudható, hogy egy adott Activityre nem lesz szükség, ha egy másikra történik meg a váltás. Ilyenkor a *finish()* függvényhívással leállíthatjuk. Ha az Activity mégis újra előtérbe kerül, a rendszer újra létrehozza.

Amikor egy Activity állapotot vált, a megfelelő callback függvényeket a rendszer meghívja. Ennek megfelelően fontos a metódusok megfelelő implementációja, valamint soha nem szabad elfeledkezni az ősoosztály egyező *callback* metódusának meghívásáról sem (ha szükség van rá). A rendszer felölőssége tehát állapotváltásokkor a callback függvények meghívása, ám azok helyes implementációja már a fejlesztők feladata.

A következőkben nézzük meg egy Activity kódját, amelyben jelezzük az egyes callback metódusok szerepét és azok meghívásának az idejét.

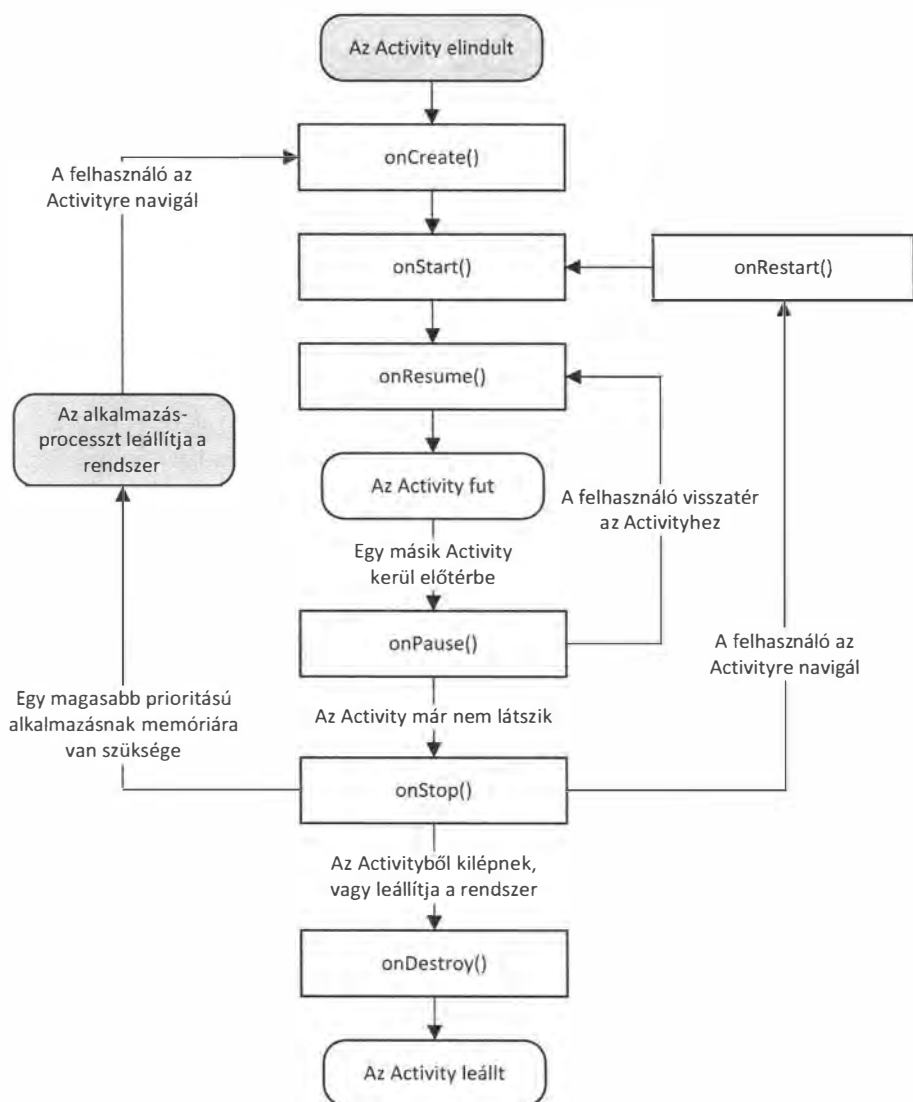
```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Most jön létre az Activity
    }
    @Override
    protected void onStart() {
        super.onStart();
        // Most válik láthatóvá az Activity
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Láthatóvá vált az Activity
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Másik Activity veszi át a focus-t
        // (ez az Activity most kerül "Paused" áll-
potba)
    }
    @Override
    protected void onStop() {
        super.onStop();
        // Az Activity már nem látható
        // (most már "Stopped" állapotba van)
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Az Activity meg fog semmisülni
    }
}
```

Az életciklusfüggvények tipikus feladatai a következők:

- *onCreate()*: Az Activity létrejön, és beállítja a megfelelő állapotokat (*layout*, munka szálak létrehozása stb.).
- *onDestroy()*: Minden még lefoglalt erőforrás felszabadítása megtörténik.
- *onStart()*: Az Activity látható és kezelhető a rajta lévő vezérlőkkel. Tipikus feladat például a feliratkozás a *BroadcastReceiver*ekre, amelyek módosíthatják a felhasználói felületet.
- *onStop()*: Az Activity nem látható, ilyenkor például érdemes a *BroadcastReceiver*ekről leiratkozozni. Az Activity élete során többször válhat látható és nem látható állapotok között, ezért a megfelelő állapotok elmentése is a fejlesztő feladata.
- *onRestart()*: Az Activity leállítása (*onStop()*), majd újraindítása után hívódik meg, még a konkrét indulás (*onStart()*) előtt.
- *onResume()*: Meghívódásakor az Activity láthatóvá válik, és előtérbe kerül, a felhasználó eléri a vezérlőket, és kezelni tudja őket.
- *onPause()*: Az Activity háttérbe kerül, de valamennyire látszik a háttérben, például egy másik Activity pop-up jelleggel előjön, vagy alvó állapotba kerül a készülék.

A fentiek alapján tehát az Activity állapotai három szakaszra bonthatók. Az Activity teljes életciklusa az *onCreate()* és az *onDestroy()* közötti állapotokat jelképezi. A látható életciklus az *onStart()* és az *onStop()* közötti szakasz, míg az előtérciklus az *onResume()* és az *onPause()* közti állapot.

A következő ábra az Activity életciklusmodelljét mutatja be az előzőekben leírtak összefoglalásaként.

2.6. ábra. Activity-életciklusmodell⁶

Activityk esetében fontos kérdés, hogy mikor válik „sebezhetővé”, mikor kell számolni azzal, hogy a rendszer felszabadíthatja. A következő táblázat összefoglalja, hogy melyik életciklusfüggvény meghívódása után válik az Activity sebezhetővé, valamint hogy az életciklusfüggvények egymáshoz viszonyítva milyen sorrendben futnak le.

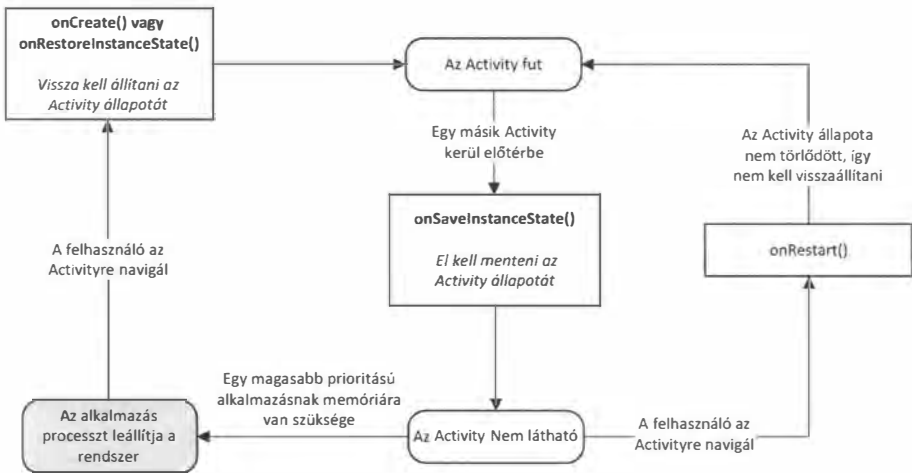
⁶ Forrás: <http://developer.android.com/reference/android/app/Activity.html>

2.2. táblázat. Activity-életciklusfüggvények

Metódus	Felszabadítható utána?	Következő metódus(ok)
<i>onCreate()</i>	Nem	<i>onStart()</i>
<i>onRestart()</i>	Nem	<i>onStart()</i>
<i>onStart()</i>	Nem	<i>onResume()</i> vagy <i>onStop()</i>
<i>onResume()</i>	Nem	<i>onPause()</i>
<i>onPause()</i>	Igen	<i>onResume()</i> vagy <i>onStop()</i>
<i>onStop()</i>	Igen	<i>onRestart()</i> vagy <i>onDestroy()</i>
<i>onDestroy()</i>	Igen	–

Az előző táblázatban a „Felszabadítható utána?” oszlop tehát meghatározza, hogy a rendszer a függvény visszatérése után leállíthatja-e a processzt anélkül, hogy további kódrészt hívna az Activity kódjából. Az Activity létrehozása után így az *onPause()* az egyetlen függvény, amely biztosan meghívódik még a processz felszabadítása előtt (az *onStop()* és az *onDestroy()* meghívódása nem garantált a hivatalos dokumentáció szerint). Ennek ellenére nem szabad túlterhelni az *onPause()* függvényt, mivel ez lassíthatja az átmenetet az Activityk között. Az oszlop „Nem” értéke esetén a rendszer védi a processzt a bezárástól, ám extrém körülmények között bármikor bezárhatja. Nagyon ritkán fordul elő extrém kevés memória, éppen ezért nem erre kell tipikusan felkészülni, hanem az általános működésre, valamint extrém körülmények kezelésekor a főbb adatok védelme válik fontossá.

Amikor egy Activity újból előtérbe kerül, a felhasználó szempontjából lényeges, hogy elfedjük, vajon a rendszer újra létrehozta-e az Activityt, vagy csak a memóriából nyitotta meg. Ezek kezelésére szolgál az *onSaveInstanceState()* callback függvény, amely akkor hívódik meg, mielőtt az Activity sebezhetővé válna a rendszer általi bezárásra. Újraindításkor (ha fel kellett szabadítani a memóriát) pedig az Activity *onCreate(Bundle savedInstanceState)* függvénye kap egy nem null értékű *Bundle* objektumot, amely az elmentett állapotváltozókat becsomagoltan tartalmazza. Az újraindításról az *onRestart()* függvény felüldefiniálásából is értesülhetünk (ha nem kellett még felszabadítani a memóriát). Az *onSaveInstanceState()* tipikusan az *onPause()* és az *onStop()* előtt hívódik meg. Nincs rá garancia, hogy mindig meghívódik, például nem fut le, ha a felhasználó a Vissza gombbal lép ki jelezve, hogy véget ért az adott Activityvel, nincs mit elmenteni. Tipikusan belső tagváltozók és kiemelt UI-elemek értékét szokás ilyenkor elmenteni, semmiképpen se használjuk perzisztens adatok mentésére, arra az *onPause()* és az *onStop()* függvényeket szokás alkalmazni. Ne feledkezzünk el az *ős (super)* implementációját is meghívni az *onSaveInstanceState()* esetében. Az *onSaveInstanceState()* működéséről legegyszerűbben úgy győződhetünk meg, ha az Activity futása közben elforgatjuk a képernyőt, ugyanis ilyenkor a rendszer újraindítja az Activityt. A mechanizmus működését a következő ábrán foglaljuk össze.

2.7. ábra. Activity állapotának mentése⁷

A készülék/rendszer fontos paramétere tehát néha változhat futás közben is, ilyen például a képernyő-orientáció megváltozása, a külső billentyűzet csatlakoztatása, lokalizációváltás stb. Ezeknek a változásoknak az esetében a rendszer újraindítja az Activityt (*onDestroy()* és egyből *onCreate()* hívás), ugyanis ilyenkor új erőforrásokra lehet szükség az új konfigurációhoz (például más lesz a háttér, ha változik az orientáció). Ebben az esetben tehát az állapot elmentésére az *onSaveInstanceState()* használata adhatja a legkézenfekvőbb megoldást. Visszatöltéshez használható még az *onRestoreInstanceState()*, de az *onCreate()* használata az elterjedtebb.

2.5. Több Activity kezelése egy alkalmazásban

A legtöbb Android-alkalmazás tipikusan nem egy, hanem több Activityből áll. Képzeljünk el például egy tennivalókat kezelő alkalmazást, amelynek a fő Activityje a tennivalók listája, majd egy tennivalót kiválasztva az adott elem részleteit megjelenítő Activity kerül előtérbe, illetve egy új tennivaló felvételére szolgáló felületet is egy másik Activity tartalmazhatja. Ennek az egyszerű alkalmazásnak az esetében is tehát már három Activityt használtunk.

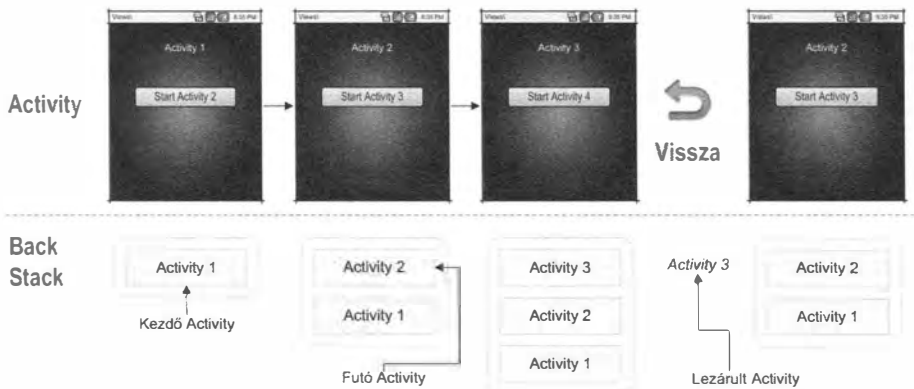
⁷ Forrás: <http://developer.android.com/guide/topics/fundamentals/activities.html>

Tételezzük fel, hogy az alkalmazásunkban az A Activityről átváltunk a B Activityre. Ekkor az életciklusfüggvények meghívódási sorrendje a következő:

1. A Activity *onPause()* függvénye,
2. B Activity *onCreate()*, *onStart()* és *onResume()* függvénye (B Activity-n van már a fókusz),
3. A Activity *onStop()* függvénye, mivel az már nem látható, és az elemei nem is vezérelhetők (például a gombok nem nyomhatók le).

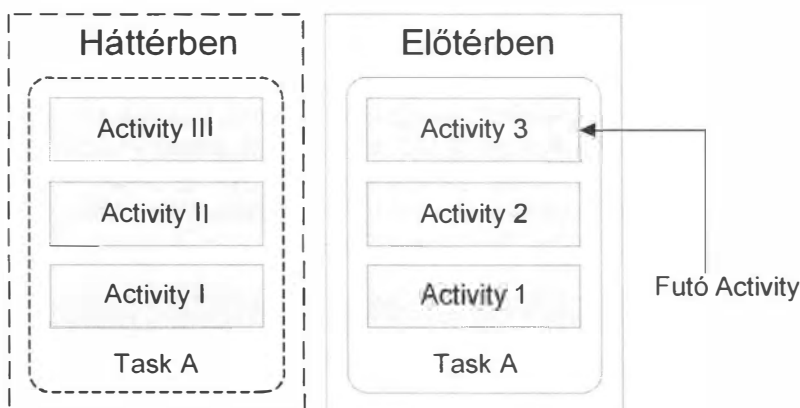
Az előzőekből következik tehát, hogy ha a B Activity valamit adatbázisból olvas ki, amit az A ment el, akkor ennek a mentésnek az A Activity *onPause()* függvényében kell megtörténnie, hogy a B aktuális legyen, mire az a felhasználó előtt megjelenik.

A rendszer egy alkalmazáson belül az Activityket egy úgynevezett *last-in, first-out* típusú *Back Stack*en tárolja. Minden esetben az előtérben levő Activity van a *Back Stack* tetején. Ha a felhasználó vagy az alkalmazás átvált egy másik Activityre, akkor eggyel lejjebb kerül a Stackben, és a következő Activity lesz legfelül. *Vissza* gomb esetén a rendszer legfelülről veszi ki a megjelenítendő Activityt. A *Back Stack* működését a következő ábrán foglaljuk össze, ahol három Activity között történik váltás.



2.8. ábra. Activity Back Stack

Egy *Back Stack* tulajdonképpen egy Android-taskot jelképez, ám a rendszer multitasking képességei miatt több task is futhat egyszerre, és váltáskor a *Back Stack*ek is természetesen helyet cserélnek. Tipikusan például az Andoid-készülékek beépített kezdő képernyőre *Váltás (Home)* gombjának lenyomásakor háttérbe kerül az aktuális alkalmazás, és egy újat indíthatunk.



2.9. ábra. Multitasking Android platformon

Legtöbbször az alapértelmezett *Back Stack*-viselkedés kielégíti az igényeket, néha azonban szükség lehet ennek az alapértelmezett viselkedésnek a felüldefiniálására. Előfordulhat például a *Back Stack* törlése is, ha a *Vissza* gomb hatására mindig egy kezdő Activityre kellene visszalépni. Az alapértelmezett viselkedés felülírására két lehetőségünk is van. Egyrészt a *manifest* állományban az `<activity>` elembe állíthatók be különféle paraméteretek, másrészt pedig a másik Activityt indító `startActivity()` függvény paraméterezhető fel különféle módon.

Az `<activity>` elem lehetséges attribútumai meghatározzák, hogy az új Activity hogyan viselkedjen a többihez képest:

- *taskAffinity*: eldönti, melyik taskhoz tartozik;
- *launchMode*: indítási mód (mindig új példány, előző használata stb.);
- *allowTaskReparenting*: új taskhoz kerül át;
- *clearTaskOnLaunch*: minden Activityt töröl a taskból;
- *alwaysRetainTaskState*: eldönti, hogy a rendszer kezelje-e a task állapotát;
- *finishOnTaskLaunch*: le kell-e állítani az Activityt ha a felhasználó kilép a taskból (például a Home gomb megnyomásakor).

A `startActivity()` függvény paraméterértékei meghatározzák, hogy az újonnan indított Activity hogyan viselkedjen az éppen futóhoz képest (tehát ahhoz képest, aki indítja):

- `FLAG_ACTIVITY_NEW_TASK`,
- `FLAG_ACTIVITY_CLEAR_TOP`,
- `FLAG_ACTIVITY_SINGLE_TOP`.

Ha az alapértelmezett viselkedést módosítjuk, mindenképp teszteljük az alkalmazást navigálás és felhasználói élmény szempontjából, mert sokszor a fejlesztés szempontjából jó megoldás nem ideális a felhasználói oldalról.

Egy új Activity indítása meglehetősen egyszerű feladat, pusztán csak egy *Intent* objektumot kell létrehozni, meg kell adni a hívó komponenscsomag környezetét, valamint a cél-Activityt jelképező osztályt, és egy alkalmazás-komponensen belül az őssztálytól örökölt *startActivity()* függvény hívásával kezdeményezhető a másik Activity indítása, például:

```
Intent myIntent = new Intent();
myIntent.setClassName(this, SecondActivity.class);
// Adat átadása
myIntent.putExtra("MyValue", "my data");
startActivity(myIntent);
```

Emellett lehetőség van adat átadására is a hívandó Activity számára az *Intent* objektum *putExtra()* függvényének segítségével, ahol elegendő a kulcs és az átadandó érték megfelelő beállítása. Ezután ezt a hívott Activity a paraméterként kapott *Bundle*-ben kapja meg.

2.6. Az első Android-alkalmazás

Végül bemutatjuk egy egyszerű Activity felépítését, és megnézzük, hogy ez hogyan bővíthető néhány alapvető funkcióval. Érdeemes a következő példákat működés közben kipróbálni.

Első lépésként készítsünk egy egyszerű Activityt, amely csak egy szöveget jelenít meg mindenféle erőforrás-állomány használata nélkül.

```
package hu.bute.daai.amorg.examples.hello;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello Android Fejlesztő!");
        setContentView(tv);
    }
}
```

A fenti kódrész egy önálló, működő Android-alkalmazást eredményez. A `setContentView()` függvény segítségével megadhatjuk, hogy mi legyen az Activity fő nézete, amely jelen esetben egy egyszerű `TextView`. A legtöbbször azonban valamilyen erőforrás-állománnyal szoktuk a felhasználói felületet megadni, valamint a szöveges értékeket is erőforrásban (*strings.xml*) kell tárolni a többnyelvűség támogatása miatt.

Második lépésként hozzunk létre egy erőforrás-állományt a projekten belül a *res/layout* könyvtárba, például *main.xml* néven.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

Ez esetben az Activity forráskódja a következőképpen módosul, és éri el az Activity az erőforrást:

```
package hu.bute.daai.amorg.examples;

import android.app.Activity;
import android.os.Bundle;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Egészítsük ki az erőforrás-állományban a `TextView` szövegelemet egy azonosítóval, ahol a `@+id` jelölésben a `+` jellel jelezzük, hogy a rendszer generáljon egy új egyedi azonosítót.

```
<TextView
    android:id="@+id/textViewStatus"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
/>
```

Módosítsuk az Activity *onCreate()* függvényét, hogy változtassa meg a *TextView* tartalmát.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    TextView textViewStatus =
        (TextView)findViewById(R.id.textViewStatus);
    textViewStatus.append("\n--MODIFIED--");
}
```

Végezetül egészítsük ki szintén az *onCreate()* függvényt, hogy a szövegelemre történő kattintáskor egy meghatározott szöveg fűződjön az elem tartalmához.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    final TextView textViewStatus =
        (TextView)findViewById(R.id.textViewStatus);
    textViewStatus.append("\n--MODIFIED--");
    textViewStatus.setOnClickListener(new
OnClickListener() {
        public void onClick(View v) {
            textViewStatus.append("\n--CLICKED--");
        }
    });
}
```

Az alkalmazásunk könnyedén kiegészíthető menük támogatásával, a menük szerepe az Android 3.0-s platformtól azonban folyamatosan csökken.

A menü támogatásához tipikusan az *onPrepareOptionsMenu()* és az *onOptionsItemSelected()* függvényeket kell felüldefiniálni az Activityben. Előbbi meghatározza, hogy melyik menüpontok jelenjenek meg dinamikusan a menügomb lenyomásakor, az utóbbi akkor hívódik meg, amikor egy konkrét

menüelemet kiválasztottunk (paraméterként kapjuk meg a kiválasztott menüelemet).

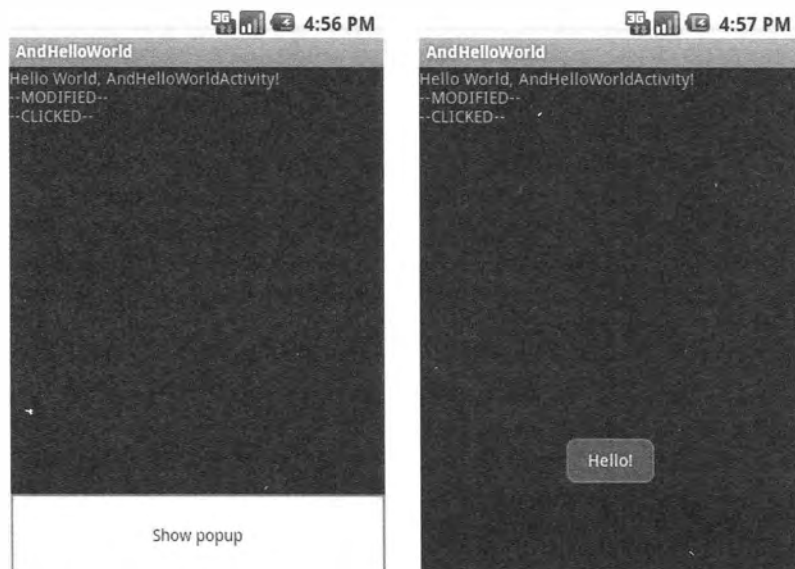
Egy egyszerű üzenetet megjelenítő menü megvalósítása például a következőképpen néz ki:

```
private final int ITEMID_SHOWMESSAGE = 1;

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    menu.clear();
    // paraméterek: csoportID, menüID, sorszám, cím
    menu.add(Menu.NONE, ITEMID_SHOWMESSAGE, 0, "Show
popup").
        setAlphabeticShortcut('e');
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case ITEMID_SHOWMESSAGE:
            Toast.makeText(this, "Hello!",
                Toast.LENGTH_LONG).show();
            break;
    }
    return true;
}
```

Figyeljük meg, hogy a pop-up ablakot egy *Toast* objektummal valósítottuk meg, amelynek eredményét a következő ábra szemlélteti.



2.10. ábra. Az elkészült alkalmazás

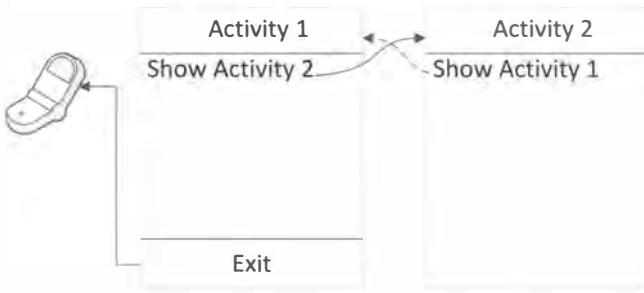
Ha az alkalmazásunkban több menüt is használunk, lehetőség van arra, hogy a menüket erőforrásból is megadjuk.

2.7. Gyakorlófeladat

A fejezetben megismertek gyakorlásához készítsünk egy több Activityből álló Android-alkalmazást. Az Activityk között a navigáció egy-egy *TextView*-ra való kattintással legyen megvalósítva.

2.7.1. Activity indítása

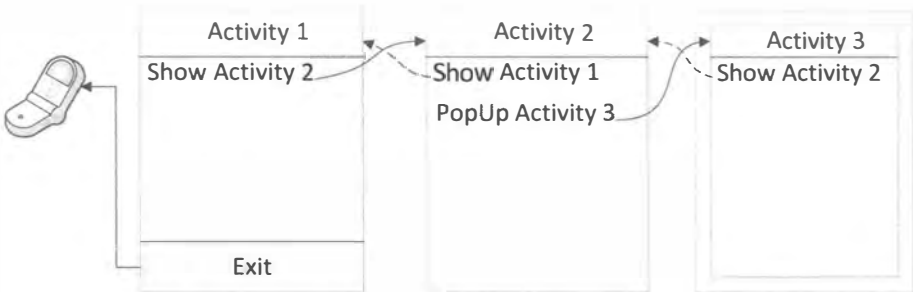
Első lépésként készítsünk egy alkalmazást, amely két Activityből áll. Az első Activity tartalmazzon egy „Show Activity2” *TextView*-t és egy „Exit” menüpontot. A „Show Activity2” *TextView*-ra kattintva jelenjen meg a második Activity, míg az „Exit” menüpontot választva lépjen ki az alkalmazás. A második Activityn egyetlen „Show Activity1” *TextView* szerepeljen, erre kattintva az első Activityre kerüljön a vezérlés, úgy, hogy a második Activity befejeződik. A második Activityben a visszalépéshez, valamint az *Exit* menüpont kiválasztásakor használjuk a *finish()* függvényt. Ne felejtsük el a *manifest* állományban is definiálni az új Activityt.



2.11. ábra. Két Activity közti váltás

2.7.2. Activity megjelenítése felugró ablakban

Adjunk hozzá az előző feladatban létrehozott alkalmazáshoz egy harmadik Activityt, amely felugró ablakban jelenik meg. A második Activity felületére helyezzünk el egy „Show Activity3” *TextView*-t, erre kattintva jelenítsük meg a harmadik Activityt a felugró ablakban. A harmadik Activity-n szerepeljen egy „Show Activity2” *TextView*, erre kattintva a második Activity jelenjen meg, úgy, hogy a harmadik Activity befejeződik.



2.12. ábra. Három Activity kezelése

A második Activity Layout-jának XML-felületleírása a következő (figyeljük meg a *LinearLayout* `android:orientation="vertical"` tulajdonságát):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">
    <TextView
        android:id="@+id/show1from2"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/show1"
    />
    <TextView
        android:id="@+id/show3from2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/show3"
    />
</LinearLayout>
```

A harmadik Activity felugróablak-jellegű megjelenítéséhez *manifest* beállítás szükséges.

```
<activity android:name=".ActivityThird"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Dialog">
</activity>
```

2.7.3. Életciklusfüggvények nyomonkövetése

Az életciklusfüggvények meghívódásának nyomonkövetésére helyezzünk az egyes Activityk életciklusfüggvényeibe *LogCat* hívásokat egy saját taggel, és elemezzük az Activityk életciklusait az Eclipse *LogCat* nézeten úgy, hogy szűrőt hozzon létre a saját tagjére.

2.7.4. Alkalmazásikon lecserélése

Végezetül cseréljük le az alkalmazás ikonját, amelyet egy új Android-projekt esetében a projekten belül a *res/drawable-[ldpi/mdpi/hdpi]* könyvtárakban található *ic_launcher.png* állományok lecserélésével lehet a legegyszerűbben megtenni. A kötőjel utáni minősítőkkal biztosíthatjuk, hogy nagyobb pixelsűrűségű készülékekre jobb minőségű kép kerüljön (lásd a következő fejezetben részletesen).

Az alkalmazás ikonja a *manifest* állomány *<application>* tagjának *android:icon* elemében van definiálva:

```
android:icon="@drawable/ic_launcher"
```


Felhasználói felület tervezése és készítése

A fejezet célja az, hogy az Android felhasználói felület tervezéséhez és készítéséhez tartozó elveket bemutassa és példákkal illusztrálja. Nem célunk az Android által támogatott összes felhasználófelület-elem bemutatása, ugyanis a használatuk könnyen érthető, továbbá a későbbi fejezetekben is lesz róluk szó.

A felhasználói felület kialakítására minden mobilalkalmazásban különös figyelmet kell fordítani, hiszen elsőként ez alapján ítélik meg a felhasználók az alkalmazást, ezzel érhetőek el az alkalmazás funkciói, valamint nem utolsósorban ezzel találkoznak először a felhasználók, amikor az alkalmazások között böngésznek az Android Marketen.

A felhasználói felület kialakításának legnagyobb nehézsége az Android platformon a gyártók és a készülékek különbözőségéből ered. Számos különféle készülék létezik, eltérő képernyőméretekkel, pixelsűrűségekkkel, továbbá a beviteli eszközök is különböznek, illetve az érintőképernyők minősége is eltérő lehet. Mindezek mellett a tervezett alkalmazásnak mégis ugyanúgy kell megjelennie minden készüléken, hiszen a cél a minél szélesebb felhasználói tábor megszerzése, ehhez pedig elengedhetetlen a gazdag és látványos felhasználói felület kialakítása.

3.1. Különböző méretű és felbontású képernyők kezelése

Joggal merülhet fel a kérdés, hogy az Android hogyan tudja kezelni a sok különböző gyártótól származó eszközt. Ez a különbség vajon azt jelenti-e, hogy a különböző felbontásokra külön meg kell tervezni az alkalmazás felhasználói felületét?

Szerencsére a kérdésre a válasz egyértelműen nem. Az Android platform az 1.6-os verziótól felfelé egy jól átgondolt mechanizmust biztosít a felhasználói felület kezelésére, és a munka nagy részét leveszi a fejlesztők válláról. Ahhoz azonban, hogy ki tudjuk használni a rendszer nyújtotta lehetőséget, alaposan meg kell ismernünk a rendszer viselkedését, és meg kell értenünk bizonyos fogalmakat. Ezután elegendő lesz csupán a szükséges felületek és grafikák elkészítése a megfelelő méretekben, és a kijelzőhöz igazítást, valamint a szűkség szerinti skálázást a rendszer már rugalmasan kezeli a fejlesztők helyett.

Mindemellett az Android lehetővé teszi a felhasználói felület megkülönböztetését is az adott készülék valamilyen paramétere alapján. Android platformon a felhasználói felületet XML-erőforrásként is definiálhatjuk, és a platform lehetőséget biztosít arra, hogy például különböző felületet tervezünk a telefonokhoz és a táblagépekhez úgy, hogy a felület mögött lévő üzleti logikát ne kelljen megváltoztatnunk, és a rendszer automatikusan a telefonra vagy a táblagépre tervezett erőforrást válassza futtatáskor.

A következőkben tekintsünk át néhány olyan fogalmat, amelyeket a későbbiekben használunk:

- **Képernyőméret (*screen size*):** Tulajdonképpen azt a fizikai képátlót jelenti, amelyből az Android négy kategóriát különböztet meg: small, normal, large és extra large.
- **Képernyősűrűség (*screen density – dpi*):** A pixelek száma egy adott fizikai területen belül, tipikusan inchenként (dpi – dots per inch). A rendszer négy sűrűségkategóriát különböztet meg: low, medium, high és extra high.
- **Orientáció:** A képernyő orientációja a felhasználó nézőpontjából lehet álló (portrait) vagy fekvő (landscape). Az orientáció futási időben is változhat, például a készülék eldöntésével, de lehetőség van rá, hogy rögzítsük egy alkalmazás orientációját.
- **Felbontás (*resolution – px*):** A képernyőpixelek száma. A felhasználói felület tervezésekor azonban nem felbontással dolgozunk, hanem mérettel és pixelsűrűséggel.
- **Sűrűségfüggetlen pixel (*density-independent pixel – dp*):** Virtuális pixelegettség, amelyet a felhasználói felület tervezésekor szokás használni, tehát a méreteket tipikusan nem pixelben kell megadni az Android platformon, hanem dp értékben. Egy dp egy fizikai pixelnek felel meg egy 160 dpi-s képernyőn. A rendszer feladata az, hogy futási időben kezeljen minden szükséges skálázást a definiált dp-nek megfelelően. Általános képlet: $px = dp * (dpi / 160)$. Például egy 240 dpi-s képernyőn 1 dp 1,5 fizikai pixelnek felel meg.

Sokszor szükség lehet egy távolság dp-ben való megadására, majd pixelé alakítására például gesztúrák felismerésekor (nem mindegy, hogy milyen hosszan kell húzni az ujjunkat). A képernyősűrűség skálázásának mértékét a következőképpen kérdezhetjük le:

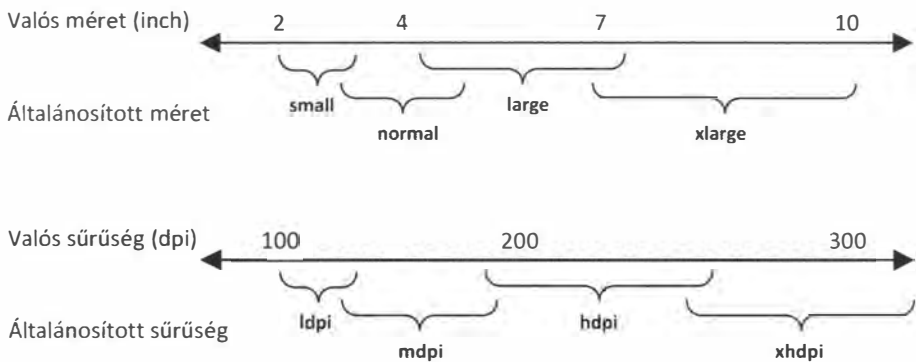
```
final float scale =
    getResources().getDisplayMetrics().density;
```

A dp érték pixellé alakítása megfelelő skálázás alapján így a következőképpen néz ki:

```
private static final float GESTURE_THRESHOLD_DP = 16.0f;
mGestureThreshold =
    (int) (GESTURE_THRESHOLD_DP * scale + 0.5f);
// Az mGestureThreshold változó használata már pixel
mennyiségként...
```

Közepes sűrűségű képernyőn (medium-density) a *DisplayMetrics.density* értéke általában 1,0, míg nagy sűrűségűn (high-density) 1,5.

Tehát az Android négy általános képernyőméretet különböztet meg, amelyek azonosítói: *small*, *normal*, *large* és *xlarge*. Ezeket az azonosítókat a későbbiekben is használjuk az erőforráskönyvtárak megjelölésére. Emellett a négy különböző pixelsűrűség azonosítói: *ldpi*, *mdpi*, *hdpi* és *xhdpi*. A következő ábra összefoglalja, hogy az egyes kategóriák mely képernyőméreteket és sűrűségeket fedik le.



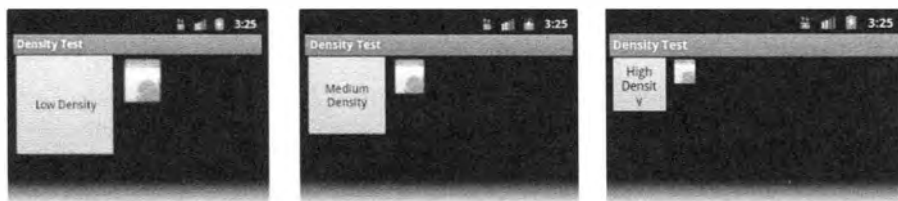
3.1. ábra. Képernyőméretek és sűrűségek kategorizálása

Az Android a következő minimumküszöböket definiálja az egyes méretekhez:

- *xlarge*: legalább 960dp x 720dp
- *large*: legalább 640dp x 480dp
- *normal*: legalább 470dp x 320dp
- *small*: legalább 426dp x 320dp

Az okostelefonok rohamos fejlődésével a készülékkijelzők is óriási fejlődésen mennek keresztül. Manapság a kijelző tulajdonságát már nemcsak a mérete és a felbontása határozza meg, hanem a pixelek közti távolság is. Az alkalmazás akkor lehet sűrűségfüggetlen, ha a felhasználói felületi elemek a felhasználó szemszögéből megőrzik a fizikai méretüket a különböző sűrűségeken. A sűrűségfüggetlenség fenntartása nagyon fontos, hiszen például egy gomb fizikailag nagyobbnak tűnhet egy alacsonyabb sűrűségű képernyőn. A képernyősűrűséghez kapcsolódó problémák jelentősen befolyásolhatják az alkalmazás felhasználhatóságát. Az Android kétféleképpen is segít elérni a sűrűségfüggetlenséget. Egyrészt a rendszer a dp kiszámítása alapján skálázza a teljes felhasználói felületet az aktuális képernyősűrűségnek megfelelően, másrészt pedig a képernyősűrűség alapján automatikusan átskálázza a képernyőforrásokat.

A következőkben nézzünk meg két példát. Az első egy olyan felületet mutat be különböző pixelsűrűségű képernyőkön, amelyekben nem követtük a sűrűségfüggetlenség elveit (pl. nem dp-ben adtuk meg a méreteket), míg a második a sűrűségfüggetlenséget támogató felületet illusztrálja.



3.2. ábra. Felhasználói felület-sűrűség függetlenségtámogatás nélkül⁸



3.3. ábra. Felhasználói felület-sűrűség függetlenségtámogatással

Általános érvényű elv a felhasználói felületen alkalmazott grafikus erőforrásokkal kapcsolatban, hogy nem szerencsés, ha a rendszerre bízunk az átméretezést, hiszen így nagy felbontáson elmosódottak lehetnek például az alacsony minőségű képek. Az Android platform egy nagyon jól működő mechanizmust alkalmaz, amelynek segítségével választ az előre megadott különféle felbontású képek közül, és azt jeleníti meg, amelyik a legjobban igazodik a készülék kijelzőjének a tulajdonságaihoz.

⁸ Forrás: http://developer.android.com/guide/practices/screens_support.html

Minden erőforrást a *res* könyvtárba kell helyoznunk, és minden erőforrás-típusnak (kép, elrendezés, szöveges állomány stb.) külön alkönyvtára van. Például a képeket a *drawable* alkönyvtárba kell elhelyezni.

Az Android azonban lehetővé teszi, hogy az egyes alkönyvtárak neve után minősítőket fűzzünk, ezekkel megadjuk, hogy az abban a könyvtárban lévő erőforrások felhasználása milyen rendszertulajdonság esetén történjen meg. Képek esetén például *large* képernyők támogatásához egy *drawable-large* könyvtárban kell elhelyoznunk a megfelelő képeket. Hasonlóan, a többi képernyőméret támogatásához is külön könyvtárakat kell létrehoznunk. Továbbá egy könyvtárhoz több minősítőt is egymás után fűzhetünk, tehát például készíthetünk *drawable-large-hdpi* könyvtárat is.

A minősítők általános szerkezete tehát a következő: `<resources_name>-<qualifier1>-<qualifier2>-...`, ahol a `<resource_name>` az erőforrástípushoz tartozó alkönyvtár (például: *drawable*, *layout* stb.), a `<qualifier>` pedig maga a minősítő.

A következőkben tekintsünk át néhány minősítőt:⁹

- Méret:
 - small
 - normal
 - large
 - xlarge
- Sűrűség:
 - ldpi
 - mdpi
 - hdpi
 - xhdpi
 - nodpi (a rendszer az ezen könyvtárban lévő elemeket nem méretezi át)
 - tvdpi
- Irány:
 - land (fekvő nézet)
 - port (álló nézet)
- Képarány:
 - long (jelentősen szélesebb vagy magasabb kijelzőkhöz)
 - notlong

⁹ További minősítők: <http://developer.android.com/guide/topics/resources/providing-resources.html> és http://developer.android.com/guide/practices/screens_support.html

- Szélességi és magassági megkötések:
 - *w<N>dp* (available screen width): A legkisebb szélesség, amellyel az erőforrást használni lehet.
 - *h<N>dp* (available screen height): A legkisebb magasság, amellyel az erőforrást használni lehet.
 - *sw<N>dp* (smallest width): A képernyőn elérhető legkisebb magasság és szélesség, például: *sw600dp*, *sw720dp*.

Az Android egy alkalmazáshoz futás közben egy meghatározott logika alapján választja ki a megfelelő erőforrásokat. Első lépésként megkeresi, hogy a futtató készülék paramétereire leginkább igazodó erőforrások rendelkezésre állnak-e (a minősítők alapján). Ha nincs illeszkedő erőforrás, akkor egy kisebb méretűt, illetve alacsonyabb sűrűségűt választ (például a *large* mérethez *normalt*). Ha az elérhető erőforrások csak nagyobb képernyőkhöz léteznek, mint a készülék képernyője, akkor az alkalmazás hibát jelez, tehát például, ha az összes egy típusú erőforrás *xlarge*-dzzsal van megjelölve, akkor *normal* képernyős eszközökön hiba keletkezik. A négyféle képernyősűrűség támogatásához érdemes a 3 : 4 : 6 : 8 arányú elveket követni.

Egy ikon esetében például ez a következő méreteket jelenti különböző sűrűségeken:

- *low-density*: 36*36
- *medium-density*: 48*48
- *high-density*: 72*72
- *extra high-density*: 96*96

Ha olyan alkalmazást készítünk, hogy azt csak bizonyos szélességű vagy magasságú készülékekre lehessen telepíteni, akkor ezt a *manifest* állományban jelezhetjük. Ha a kijelző nem felel meg ezeknek a kitételeknek, akkor vagy nem lehet telepíteni az alkalmazást, vagy kompatibilis módban futtatja a rendszer.

```
<manifest ... >
    <supports-screens android:requiresSmallestWidth
Dp="600" />
    ...
</manifest>
```

Néhány fontosabb jelölő:

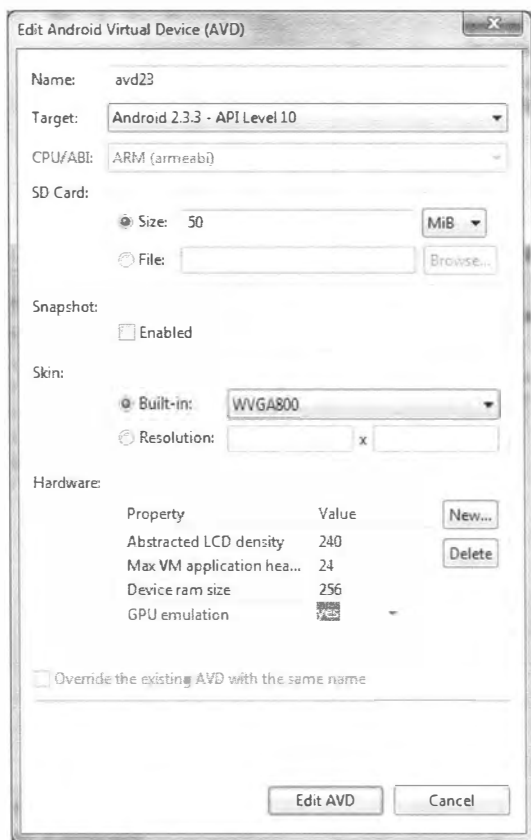
- *android:requiresSmallestWidthDp*: A legkisebb dimenzió, amellyel a képernyőnek rendelkeznie kell.

- *android:compatibleWidthLimitDp*: A maximum legkisebb szélesség, amelyet még az alkalmazás támogat.
- *android:largestWidthLimitDp*: A maximum legnagyobb szélesség, amely még támogatott.

Az erőforrások kezelésével kapcsolatban összefoglalásképpen mindenképp érdemes megemlíteni a legfontosabb szempontot, azt, hogy a tervezett felhasználói felület és a képi elemek arányaiban jól mutassanak minden kijelzőn, valamint ha táblagépeket is támogatni szeretnénk, akkor a felhasználói felület használja ki a táblagépek adta lehetőséget. A felhasználói felület tervezésekor javasolható a *wrap_content*, a *fill_parent* és a *match_parent* jelzőket használni. Fontos, hogy soha ne használjuk a pixel (px) mértékegységet, hanem helyette a dp mértékegység használatára törekedjünk. Szövegek megjelenítésére pedig a dp-hez hasonlóan az sp (scale-independent pixel) mértékegységet érdemes alkalmazni. Továbbá érdemes kerülni az *AbsoluteLayout* elrendezés használatát, ugyanis már elavultnak számít. Végül pedig fontos kiemelni, hogy úgy tervezzük meg a felhasználói felületet, hogy az támogassa az álló és a fekvő képernyőt is.

Mielőtt az alkalmazást kiadnánk, fontos, hogy teszteljük különböző képernyőméreteken és -felbontásokon. Szerencsére az Android SDK nagyon jó környezetet biztosít erre, és egy új virtuális gép (AVD) létrehozásakor lehetőségünk van a képernyő tulajdonságainak megadására, amelyet akár később is módosíthatunk. Létrehozáskor tetszőlegesen megadhatjuk a képernyő méretét, felbontását és pixelsűrűségét is, valamint akár több AVD-t is megvalósíthatunk, ez még inkább megkönnyíti a fejlesztést. Továbbá az emulátor indításakor parancssorból is megadhatunk egy skálázási és sűrűségi értéket, például:

```
emulator.exe -avd <avd_name> -scale 96dpi
```



3.4. ábra. AVD tulajdonságainak a beállítása

Egy Android-alkalmazás fejlesztésekor a felhasználói felületi elemek és erőforrások egy külön könyvtárban (*/res*) helyezkednek el a projekten belül. Ezen a könyvtáron belül az egyes erőforrástípusok külön egy-egy megfelelő alkönyvtárban találhatóak. Erőforrásként gondolunk mind a különféle képi elemekre, mind pedig az XML-ben megadott felhasználói felületi elrendezésekre (*layout*) és a szöveges erőforrásokra is.

Egy tipikus Android-alkalmazás több Activityből áll, és minden Activityhez egy XML-ben leírt felületelrendezést szokás csatolni. Egy ilyen layoutállomány tipikusan több felületi elemet tartalmaz, például gombokat, szövegbeviteli mezőket stb. Ezeket az elemeket általában valamilyen azonosítóval szokás ellátni, amelyekhez a rendszer a korábban bemutatott *R.java* állományban rendel egy egyedi *int* azonosítót. Ezt követően az Activity forráskódjából a *findViewById(int id)* függvénnyel kereshetők meg a felületi elemek, és dinamikusan vezérelhetők, valamint módosíthatók.

A legfontosabb erőforrásokat az alábbi könyvtárakba szokás elhelyezni:¹⁰

- Felületeírások: *res/layout*
- Szöveges erőforrás: *res/values/strings.xml*
- Képerőforrások: *res/drawable*
- Animációk: *res/anim*
- Menük: *res/menu*

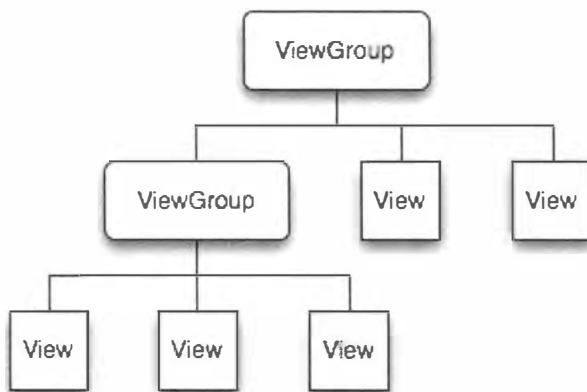
3.2. Android-layoutok

Android platformon a felhasználói felület kialakítása tipikusan úgynevezett *layout*ok segítségével történik. A layout szerepe az, hogy megadja, hogy a benne lévő elemek milyen szabály szerint rendeződnek el (például egymás alatt, táblázatosan stb.). Minden layout tulajdonképpen egy *ViewGroup*, amely a benne lévő *View* osztályból leszármazott felületi elemek elrendezéséért felelős. Egy *ViewGroup* további *ViewGroup*okat is tartalmazhat gyermekelemként, így komplex felületi struktúra alakítható ki. Egy Activityhez tipikusan egy layout tartozik, amely így meghatározza az adott Activity felhasználói felületét.

Amikor az adott Activity fókuszt kap, a rendszer közvetlenül az átadott *View*-hierarchia gyökérelemét utasítja, hogy végezze el a rendereléshez szükséges területmérést és rajzolást a teljes fára. A gyökérelem – amennyiben *ViewGroup*-leszármazott – továbbítja ezt az igényt saját gyermekeihez, majd a hívás rekurzívan halad tovább a fában, egészen a primitív *View*-levélelemek eléréséig. Ha a gyökérelem maga egy egyszerű *View*-leszármazott (ahogy például a bevezető példában bemutatott, egyetlen *TextView*-ből álló XML-felülettervben), kirajzolása triviális.

A *ViewGroup*, ahogy arra az elnevezéséből következtethetünk, speciális típusú *View*, amelynek célja, hogy csoportba szervezzen *View*-kat és *ViewGroup*okat, lehetővé téve ezzel, hogy az UI-elemek komplex, hierarchikus struktúráját alakítsuk ki. Egy összetett felhasználói felület vázlatos felépítését így képzelhetjük el:

¹⁰ Forrás: <http://developer.android.com/guide/topics/resources/available-resources.html>

3.5. ábra. View-hierarchia¹¹

A *ViewGroup* elemei felelősek az általuk igényelt terület méréséért, közvetlen gyermekeik kirajzolásáért és a saját layoutstratégiájuk szerinti elrendezéséért. Így komplex UI esetén a renderelés a fában az alsóbb szinteken található *ViewGroup*-ok szintjén kezdődik, a levélelem-*View*-k méretének beállításával és kirajzolásával, majd a folyamat felfelé halad végig a fán, a gyökerelem eléréséig. Végül tehát a teljes felhasználói felület kinézete és elrendezése ismertté válik.

Összefoglalva tehát, az egy *Activity*-hez tartozó felhasználói felület tipikusan egy *View*-kből és *ViewGroup*-okból felépülő fa. Felépítése során használhatjuk mind a platform SDK-val szállított, előre implementált widgeteket és layoutokat, mind pedig a saját magunk által implementált *View*-típusokat (például saját osztályt származtatunk a *View* osztályból). A későbbiekben az *Activity*-felület kialakításának elvét a táblagépek támogatására kidolgozott *Fragment API* módosítja (ezt lásd a könyv végén). Továbbiakban a beépített *View*-kra (*Button*, *TextView* stb.) widgetként is hivatkozunk.

A megfelelő paraméterek használatával a layoutban beállíthatjuk az elrendezés irányát, megadhatunk margót az egyes gyermekek között, súlyok kiosztásával dönthetünk arról, hogy az esetleges szabad helyeket hogyan töltsék ki a *View*-k, továbbá a *gravity* tulajdonság használatával a megfelelő irányba igazíthatjuk őket.

A legegyszerűbb nézet a *LinearLayout*, amely tulajdonképpen az *orientation* tulajdonságától függően egymás mellett vagy egymás alatt tudja elhelyezni az elemeket. A következő példa egy egyszerű *LinearLayout*-ot szemléltet, amelyben egymás alatt található egy *TextView* és egy *Button* felületi elem. Ha a *LinearLayout orientation* tulajdonságát a *vertical*-ról *horizontal*-ra cserélnénk, az elemek egymás mellett helyezkednének el.

¹¹ Forrás: <http://developer.android.com/guide/topics/ui/index.html>

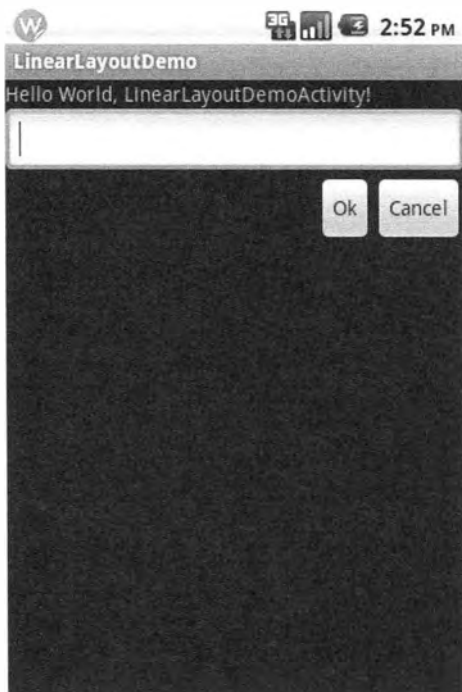
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"/>
    <Button
        android:id="@+id/btnOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnOk"/>
</LinearLayout>
```

Az Android lehetővé teszi, hogy egymásba ágyazzunk *ViewGroup*okat is. A következő példában két *LinearLayout*ot ágyazunk egymásba:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <requestFocus />
    </EditText>
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="right">
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/btnOk" />
        <Button
            android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"  
        android:text="@string/btnCancel" />  
    </LinearLayout>  
</LinearLayout>
```

Figyeljük meg, hogy a második *LinearLayout* elrendezése már horizontális, és benne az elemek a *gravity* tulajdonsággal jobbra vannak igazítva.



3.6. ábra. *LinearLayout ViewGroupok egymásba ágyazása*

Az előbb mutatott elrendezésre azonban tipikusan nem *LinearLayout*ot szokás használni, hanem *RelativeLayout*ot, amelyben megadhatjuk az elemek egymáshoz képesti pozícióviszonyát. A *RelativeLayout* segítségével tehát úgy állíthatunk össze egy felhasználói felületet, hogy az egyes *View*-k egymáshoz vagy a layouthoz viszonyított relatív helyzetét adhatjuk meg. Beállíthatjuk például, hogy egy *View* egy megadott elem mellé/alá, a képernyő közepére stb. igazodjon. Rendereléskor az UI összeállítása a megadás sorrendjében történik, a beállított kényszerek sorról sorra teljesülnek, és ez végül meghatározza a teljes felület kinézetét.

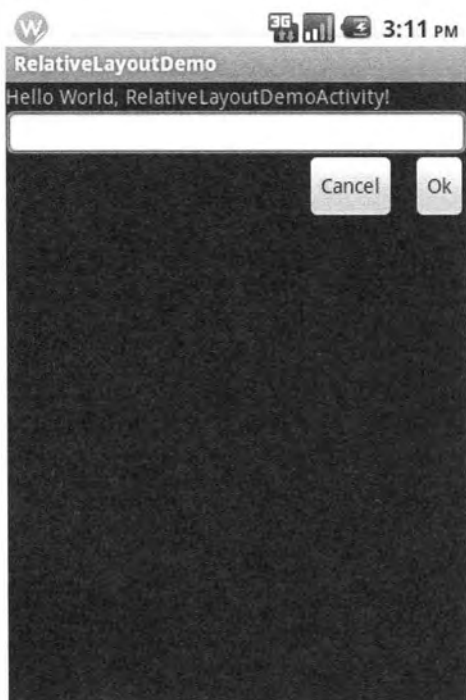
RelativeLayout készítése esetén fontos, hogy azonosítót adjunk azoknak a komponenseknek, amelyekhez a többi komponens igazítani szeretnénk. Nézzük meg a következő példát:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="
            @android:drawable/editbox_background"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/btnOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
        android:text="@string/btnOk" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/btnOk"
        android:layout_alignTop="@id/btnOk"
        android:text="@string/btnCancel" />
</RelativeLayout>

```

Figyeljük meg, hogy hogyan vannak az egyes elemek egymáshoz igazítva a *layout_bellow*, a *layout_alignParentRight*, a *layout_toLeftOf* és a *layout_alignTop* attribútumokkal.



3.7. ábra. RelativeLayout elrendezés

Viszonylag még gyakran használt nézet a *TableLayout*. A *TableLayout* *ViewGroup* elemeit táblázatszerűen rendezi oszlopokba és sorokba. Egy *TableLayout* példány több-kevesebb *TableRow* objektumból épül fel, amelyek mindegyike a táblázat egy-egy sorát definiálja. Minden sor tartalmazhat 0, 1 vagy több cellát, minden cella egyetlen *View* objektumot tartalmazhat. A táblázat oszlopainak száma megegyezik a legtöbb cellát tartalmazó sorának az oszlopszámával. A *TableLayout*-ot használhatjuk szabálytalan (egyesített/felosztott cellákat tartalmazó) rácsszerkezet összeállításához is, például a HTML nyelv esetében. Hozzáadhatunk a táblázatunkhoz akár teljes sort kitöltő *View* objektumokat is, ekkor még arra sincs szükség, hogy a *TableRow* objektumba csomagoljuk.

A *TableLayout* alapelveinek megismeréséhez nézzünk meg egy egyszerű példát:

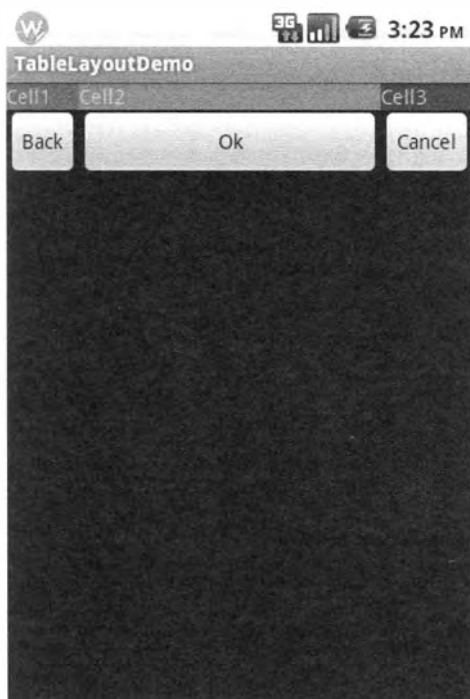
```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">
```

```

<TableRow>
    <TextView android:background="#ff0000"
        android:text="Cell1"/>
    <TextView android:background="#00ff00"
        android:text="Cell2"/>
    <TextView android:background="#0000ff"
        android:text="Cell3"/>
</TableRow>
<TableRow>
    <Button android:text="Back"/>
    <Button android:text="Ok"/>
    <Button android:text="Cancel"/>
</TableRow>
</TableLayout>

```

Az eredmény a következő:



3.8. ábra. TableLayout példa

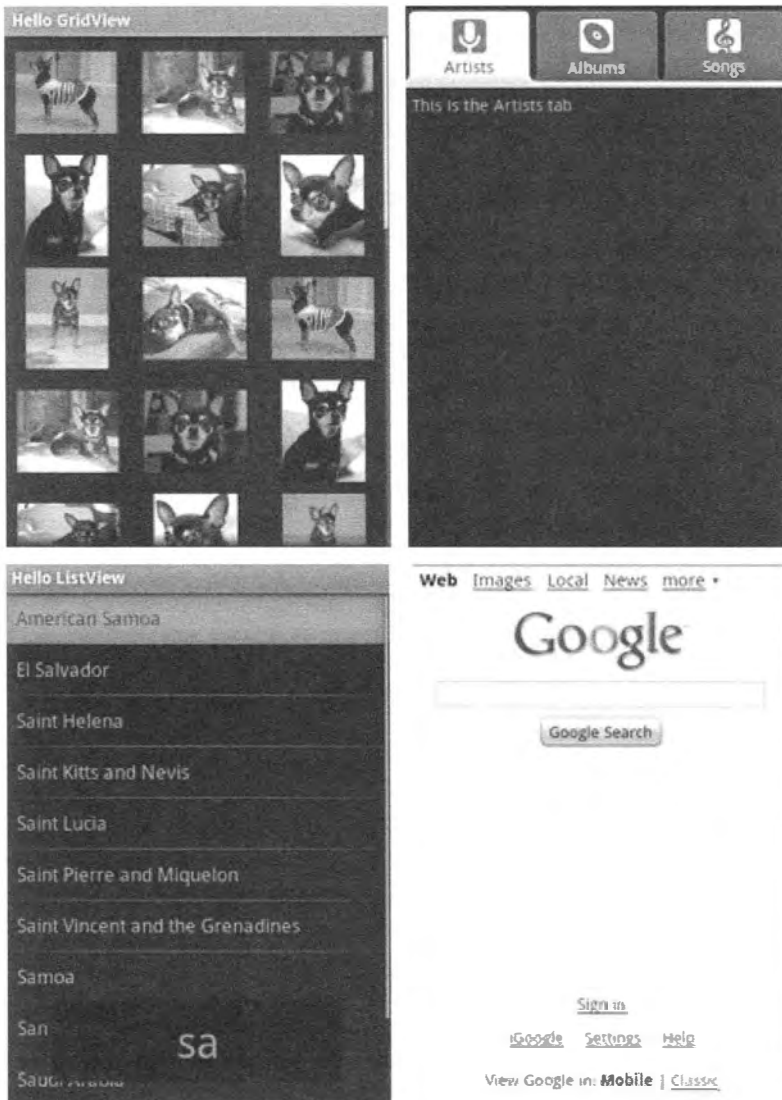
Figyeljük meg a *TableLayout stretchColumns* elrendezését, amely megadja, hogy 0-tól kezdődően hányadik oszlop legyen kinyújtva ahhoz, hogy a képernyő szélességében ki legyen töltve.

Az *AbsoluteLayout* segítségével a *View* objektumok helyét explicite megadott x/y koordinátákkal állíthatjuk be, ahol a kijelző bal felső sarka a (0, 0) pont, míg a jobb alsó sarok a (width-1, height-1). A *View*-k megadásánál a pozicionáláson kívül nem használhatunk semmilyen elrendezésre vonatkozó *property*-t.

Az *AbsoluteLayout* használatát nem javasoljuk. Egyrészt a koordinátaalapú pozicionálás esetén semmi sem véd meg a widgetek esetleges egymásba csúszásától, másrészt az eltérő felbontású készülékeken az ezzel a módszerrel tervezett felhasználói felületek tényleges képe jelentősen eltérhet egymástól, az UI akár használhatatlanná is válhat. A *LinearLayout*, a *TableLayout*, a *RelativeLayout* használata esetén a felhasználói felület kinézete ezzel szemben (viszonylag) készülékfüggetlen.

A felsoroltak mellett az Android még több olyan elrendezést is támogat, amelyeket most nem mutatunk be részletesen (lásd ezeket a későbbi példákban). A további elrendezések a következők lehetnek:

- *GridView*: táblás elrendezés,
- *TabView*: *Tab*-elrendezés, amelyen tipikusan a fülekkel válthatunk a tartalmazott nézetek között,
- *ListView*: listanézet,
- *WebView*: webtartalmat (HTML-t is) megjelenítő nézet.



3.9. ábra. GridView, TabView, ListView és WebView példák

További érdekes nézet a *MapView*, amely a Google API segítségével lehetővé teszi térképnézetek elhelyezését az alkalmazásunkban. (A *MapView* használatát lásd a helymeghatározással foglalkozó fejezetben.)



3.10. ábra. MapView példa

3.3. Android UI-vezérlők

A következőkben röviden áttekintjük a legfontosabb widgetelemeket. A képernyőképek a platform SDK *View* elemeket bemutató példaprogramjaiból származnak (API Demos / View).

A bemutatott vezérlők némelyike megszokott, sok más grafikus felhasználói felülettel rendelkező platformon is elérhető egyszerű elem, például: *Button*, *CheckBox* stb. Emellett azonban kitérünk néhány olyan Android-specifikus megoldásra, ahol a fejlesztők egy-egy összetettebb funkciót megvalósító komponenszt implementáltak egyszerűen felhasználható widgetként, ilyenek például a *Gallery*, az *AnalogClock* és a *DatePickerDialog* felületi elemek.

A konkrét felhasználói felület kialakításához kétféle módszert követhetünk. Használhatjuk egyrészt a programozott megoldást, ahol az egyes *Widget*eknek megfelelő osztályokat programkódból példányosítjuk, majd a megfelelő metódusok hívásával linkeljük össze ezeket az elemeket, hogy

a kijelzőn az általunk tervezett grafikus felület jelenjen meg. Ám már szóltunk ezen módszer alkalmazásának a hátrányairól, és bemutatunk egy másik eljárást (XML-felület-tervezés), amellyel az Android platform fejlesztői hatékony megoldást kínálnak a felhasználói felület összeállítására.

Célszerű tehát a második módszert használni (persze az első is kivitelezhető, a hozzá szükséges metódusok mind elérhetők a platformon). A gyakorlatban az UI készítésekor tehát az Android *Widget* könyvtárból válogatunk *View* elemeket, majd XML-ben, a megfelelő *ViewGroup*ok definiálásával és a *View*-k megjelenési tulajdonságait beállító propertyk használatával állítjuk össze a felhasználói felületet.

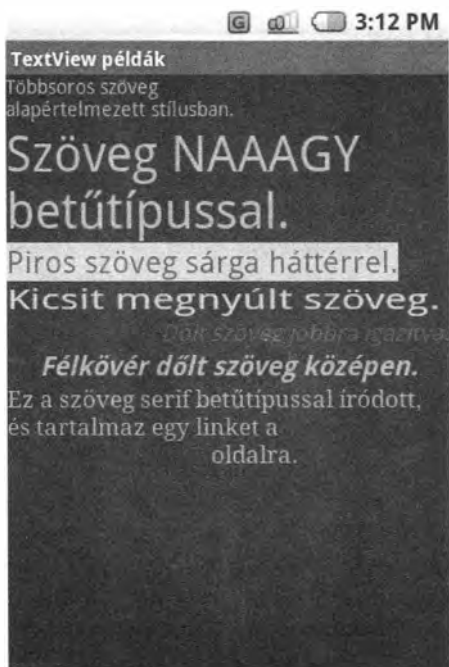
A leggyakrabban használt felhasználói felületi elemek a következők:

- *TextView*: szövegmegjelenítő elem;
- *Button*: a szinte minden grafikus platformon megszokott nyomógomb;
- *EditText*: szövegdoboz; hasonlóan használható, mint egy egyszerű *TextView* (valójában belőle van származtatva), a különbség csak az, hogy a szöveget itt a felhasználó is módosíthatja;
- *CheckBox*, *RadioButton*: szintén a megszokott módon viselkednek; a *CheckBox* esetében egyedi stílust is beállíthatunk;
- *ToggleButton*: gyakorlatilag egy nyomógombként megjelenített *CheckBox*;
- *ImageButton*: a nyomógombnak megfelelő viselkedést mutat, a különbség az, hogy a felhasználót felirat helyett egy képpel tájékoztathatjuk a gomb funkciójáról;
- *ListView*: a segítségével *View* elemek listáját jeleníthetjük meg a képernyőn, túllógás esetén automatikusan biztosítja a görgetősáv funkcióját; a feltöltéséhez célszerű az adatkötés használata, amely a platformon az *AdapterView*-k és az *Adapter* osztályok összekapcsolásával valósítható meg (lásd később);
- *Spinner*: egyfajta választéklista, szintén *AdapterView*.



3.11. ábra. Legfontosabb Android Widgetek

A következőkben nézzünk meg egy egyszerű példát, amely a következő *TextView*-kat tartalmazó felületet jeleníti meg. A példán keresztül bemutatjuk a különféle színezési és betűtípus-beállítási technikákat, ezekre sokszor szükségünk lehet.



3.12. ábra. *TextView*-alapú gyakorlófelület

A következő XML-felületterv Activitybe töltésével pontosan az előbb bemutatott képernyőképnek megfelelő eredményt kapjuk. A kódban megfigyelhető, hogy minden fenti példaszöveg a forrásban egy-egy külön *TextView*-nak felel meg, amelyeket egy *ViewGroup* elem használatával rendeztünk el egymás alá.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=
            "Többsoros szöveg\نالapértelmezett stílusban."/>
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Szöveg NAAAGY betűtípussal."
        android:textSize="36sp"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Piros szöveg sárga háttérrel."
        android:textColor="#ff0000"
        android:background="#ffff00"
        android:textSize="22sp" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Kicsit megnyúlt szöveg."
        android:textSize="20sp"
        android:textScaleX="1.5"
        android:textColor="#ffff00" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Dőlt szöveg jobbra igazítva."
        android:textSize="17sp"
        android:textStyle="italic"
        android:gravity="end"
        android:textColor="#ff00ff" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
```

```

        android:text="Félkövér dőlt szöveg közepen."
        android:textSize="19sp"
        android:textStyle="italic|bold"
        android:gravity="center" />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoLink="all"
    android:text="Ez a szöveg serif betűtípussal író-
dott, és tartalmaz egy linket a www.aut.bme.hu oldal-
ra."
    android:textSize="17sp"
    android:typeface="serif" />
</LinearLayout>

```

A kódban a felületterv-XML alapvető felépítése mellett azt is megfigyelhetjük, hogy milyen XML-propertyket használhatunk fel néhány *View*-alaptulajdonság (szöveg, betűméret, kitöltse-e szülőjét stb.) beállítására, és az értékek megadásánál milyen módokat (szöveg, színkód, numerikus érték) alkalmazhatunk. A méretet, illetve a távolságot jelölő numerikus értékek (margóméret, betűtípusméret stb.) beállításakor fontos lehet a mértékegységek ismerete.

A platform fejlesztői a használható mértékegységeket a következőképpen definiálták:

3.1. táblázat. XML-felületleírásban alkalmazható mértékegységek

Mértékegység	Jelentés
px	Pixels – a képernyőn megjelenő fizikai pixelben értendő
in	Inches – ugyancsak a fizikai képernyő méretére jellemző
mm	Millimeters – hasonlóképpen, mint az <i>in</i> esetében
pt	Point – egy inch 1/72-ed része
dp (jelölhető még: dip)	Density-independent Pixel – ahogy korábban említettük, absztrakt pixel, független a képernyő felbontásától. Definíció szerint 1 dp legyen egy 160 dpi felbontású képernyő egy pixele, tehát ettől eltérő fizikai felbontás esetén a dp-ben megadott távolság skálázódik.
sp	Scale-independent Pixel – a dp-hez hasonló, a felbontás mellett azonban figyelembe veszi még a felhasználó betűméret-preferenciáit is. Emiatt használata betűméretek megadásánál ajánlott.

Sokszor szükség lehet valamilyen felugró ablakos információ megjelenítésére. Ilyenkor többféle megoldást is biztosít a rendszer.

- **Activity megjelenítése felugró ablakban:** Egy teljes Activity jelenik meg egy felugró ablakban, míg a mögötte lévő Activity csak transzparensen lesz látható, de a rajta lévő vezérlőket nem irányíthatjuk. Használatához a *manifest* állományban kell a *theme* tulajdonságot beállítani a megfelelő Activityhez: *android:theme="@android:style/Theme.Dialog"*.
- **PopupWindow:** Felugró ablak teljes funkcionalitással, saját layoutot definiálhatunk hozzá.
- **AlertDialog:** Gomb lenyomásig megjelenő dialógusablak. Létrehozásához az *AlertDialog.Builder*re van szükség.
- **Toast:** Rövid szöveg megjelenítése kis ablakban, korlátozott ideig.

Nézzünk egy egyszerű példát az *AlertDialog* használatára. A következő függvény egy *AlertDialog*ot jelenít meg egy gombnyomás hatására:

```
private void showAlertMessage(final String aMessage) {
    AlertDialog.Builder alertbox =
        new AlertDialog.Builder(this);
    alertbox.setMessage(aMessage);
    alertbox.setNeutralButton("Ok",
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface arg0,
int arg1)
            {
                // Ok kiválasztva
            }
        });
    alertbox.show();
}
```

A példában figyeljük meg, hogy állítjuk be az *OK* gomb eseménykezelőjét.



3.13. ábra. AlertDialog példa

Következő példánkban nézzünk meg egy kicsit összetettebb komponenst, vizsgáljuk meg az *AutoCompleteTextView* használatát. A komponens feladata az, hogy a begépelte néhány betű alapján felajánljon egy listát a lehetséges tartalmakból, amelyek közül a felhasználó választhat, és nem kell begépelnie minden karaktert. Az alkalmazásunkat a következő ábra szemlélteti.



3.14. ábra. AutoCompleteTextView példa

A megvalósításhoz használt felhasználói felület XML-kódja a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
    <AutoCompleteTextView
        android:id="@+id/autoCompleteTextView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="" >
        <requestFocus />
    </AutoCompleteTextView>
</LinearLayout>
```

A logikát megvalósító Activity kódja a következőképpen néz ki:

```
public class AutoCompleteDemoActivity extends Activity
{
    static final String[] cityNames = new String[] {
        "Budapest", "Bukarest", "Krakkó", "Bécs" };

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        AutoCompleteTextView tv =
            (AutoCompleteTextView) findViewById(
                R.id.autoCompleteTextView1);
        ArrayAdapter<String> cityAdapter =
            new ArrayAdapter<String>(this,
                android.R.layout.simple_dropdown_
item_1line,
                cityNames);
        tv.setAdapter(cityAdapter);
    }
}
```

Az *AutoCompleteTextView* használatához létre kell hoznunk egy *ArrayAdapter* objektumot, amely példánkban egy *String* tömbből tölti fel magát. Ezt követően az *AutoCompleteTextView*-nek be kell állítani a létrehozott *ArrayAdapter* objektumot.

3.4. Menük készítése erőforrásból

Korábban már bemutattuk a menük használatát, az Android platformon azonban lehetőség van arra, hogy erőforrásból is menüket hozzunk létre, erre akkor van szükség, ha több/összetett menürendszerünk van. A következőkben nézzünk meg egy olyan példát, amelyben a menüt erőforrás-állományban készítettük el. A menüt leíró XML-t a */res/menu* könyvtárba kell elhelyezni.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
        android:title="@string/item1"/>
    <item android:id="@+id/item2"
        android:title="@string/item2"/>
</menu>
```

```

        <item android:id="@+id/submenu"
            android:title="@string/submenu_title">
            <menu>
                <item android:id="@+id/submenu_item1"
                    android:title="@string/submenu_
item1" />
                <item android:id="@+id/submenu_item2"
                    android:title="@string/submenu_
item2" />
            </menu>
        </item>
    </menu>

```

Figyeljük meg, hogy a fenti kódban olyan menüt definiáltunk, amely almenüt is tartalmaz. A menü betöltését a korábbi megoldáshoz hasonlóan az *onCreateOptionsMenu()* függvény felüldefiniálásban tehetjük meg, míg a menüválasztásra való reagálást pedig a jól ismert *onOptionsItemSelected()* függvényben.

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mymenu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.item1)
    {
        // TODO: item1 menü kezelése
    }
    return super.onOptionsItemSelected(item);
}

```

Figyeljük meg, hogy az almenü hogyan jelenik meg a készüléken.



3.15. ábra. Menükezelés erőforrás-állományból

3.5. Animációk készítése

Az Android kétféle animációtípust támogat. Az első az úgynevezett *tweened* animáció, amellyel a különböző UI-komponensek forgatását, mozgatását, nyújtását és halványítását lehet megoldani, a második pedig a *frame-by-frame* típusú animáció, amellyel tulajdonképpen képsorozatot lehet megvalósítani, hogy egy adott képelem bizonyos időközönként automatikusan cserélje a kép tartalmát.

Animációk összeállításához szintén készíthetünk erőforrás-állományokat XML-ben, amelyeket a */res/anim* könyvtárban kell elhelyezni.

Elsőként vizsgáljuk meg a *tweened* típusú animációkat. Az animációnak négy fő típusa lehet:

- *alpha*: átlátszóság,
- *scale*: méretezés,
- *translate*: mozgatás,
- *rotate*: forgatás.

Mindezek mellett lehetőségünk van az animációkat halmazokba (*set*) foglalni, és ezzel komplexebb animációkat alakíthatunk ki. Az animációknak különféle olyan paramétereket állíthatunk be, amelyek a viselkedésüket befolyásolják, ilyenek például a következők:

- *duration*: az animáció időtartama (ms),
- *startOffset*: animáció indításának eltolása (ms),
- *fillBefore*: az animáció indítása előtt alkalmazza-e a módosítást,
- *fillAfter*: az animáció lefutása után alkalmazza-e a módosítást,
- *interpolator*: sebességeírás.

Továbbá egy animációnak tipikusan meg lehet adni a neki megfelelő kezdő- és végértékeket, például forgatásnál a forgatás középpontját és a kezdő, illetve végső elforgatási szöget.

A következőkben nézzünk meg egy egyszerű példát, amelyben egy halmazba szervezett animációt alkalmazunk. A felhasználói felület egy *LinearLayout*-ból álljon, amelyen helyezünk el egymás alatt egy *TextView*-t és egy *Button*-t. A gomb lenyomásának hatására indítsuk el az animációt.

Az animáció leírását a *pushanim.xml* állomány tartalmazza, amelyet tehát a */res/anim* könyvtárban helyezünk el.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=
    "http://schemas.android.com/apk/res/android">
    <scale
        android:interpolator=
            "@android:anim/accelerate_interpolator"
        android:fromXScale="1.0"
        android:toXScale="0.2"
        android:fromYScale="1.0"
        android:toYScale="0.2"
        android:pivotX="50%"
        android:pivotY="50%"
        android:duration="2000"
    />
    <rotate
        android:startOffset="1000"
        android:fromDegrees="0"
        android:toDegrees="360"
        android:pivotX="10%"
        android:pivotY="20%"
        android:duration="2000"
    />
</set>
```

Az animációt tehát egy halmazban rakjuk össze, amely egy skálázást és egy elforgatást tartalmaz. Az elforgatás egy másodperccel az animáció indítása után indul, és az adott elemet a bal felső sarka környékén forgatja el.

Az animációt indító Activity a következőképpen néz ki:

```
public class TweenAnimDemoActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final LinearLayout layoutMain =
            (LinearLayout)findViewById(R.
id.layoutMain);
        final Button btnAnim =
            (Button)findViewById(R.id.btnAnim);
        btnAnim.setOnClickListener(new
OnClickListener() {
            @Override
            public void onClick(View v) {
                Animation pushAnim =
                    AnimationUtils.loadAnimation(
                        getApplicationContext(),
                        R.anim.pushanim);
                layoutMain.startAnimation(pushAnim);
                btnAnim.startAnimation(pushAnim);
            }
        });
    }
}
```

Tételezzük fel, hogy a felhasználói felületen létezik egy *layoutMain* és egy *btnAnim* komponens, amelyekre elsőként elkérjük a referenciát. Ezt követően a gomb eseménykezelőjében az *AnimationUtils* osztály segítségével betöltjük az animációt az erőforrásból, végül pedig ezt mind a *layoutMain*, mind pedig a *btnAnim* komponensre alkalmazzuk.



3.16. ábra. Tween animáció alkalmazása

Következő példánkban nézzük át a *frame* animáció használatát. Elsőként szükségünk lesz több képre, amelyek között az animáció során váltunk. Példánkban legyen ezenek a képfájloknak a neve *monster1..5*. A frame animáció *monstermove.xml*-ben megadott viselkedése a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="
    http://schemas.android.com/apk/res/android"
    id="selected" android:oneshot="false">
    <item android:drawable=
        "@drawable/monster1" android:duration="200" />
    <item android:drawable=
        "@drawable/monster2" android:duration="200" />
    <item android:drawable=
        "@drawable/monster3" android:duration="200" />
    <item android:drawable=
        "@drawable/monster4" android:duration="200" />
    <item android:drawable=
        "@drawable/monster5" android:duration="200" />
</animation-list>
```

Látható tehát, hogy *FrameAnimation* esetében valójában csak arról van szó, hogy felsoroljuk a különböző képeket, és megadjuk, hogy melyik kép mennyi ideig látszódjon.

Tételezzük fel, hogy a felhasználói felületünkön található egy *ivMonster* azonosítóval rendelkező *ImageView*. Az animáció indítása a következőképpen történik:

```
ivMonster = (ImageView)findViewById(R.id.ivMonster);
ivMonster.setBackgroundResource(R.anim.monstermove);
AnimationDrawable frameAnimation =
    (AnimationDrawable) ivMonster.getBackground();
frameAnimation.start();
```

3.6. Stílusok és témák

Az Androidhoz hasonló, XML-layout-tervezést biztosító grafikus platformok esetében (lásd .NET WPF) megszokott, hogy a fejlesztők különböző módszerekkel egyszerűsítik a felület leírásának a folyamatát. Például a widgetek tulajdonságainak egyenkénti beállítása helyett készíthetünk összetett sablonokat, amelyek a kinézetet befolyásoló propertyk egy csoportját állítják be. Ha a sablont egy-egy vezérlőre (esetleg egy teljes *Activity View*-hierarchiára) alkalmazzuk, hatékonyabban dolgozhatunk, mint az egyenkénti tulajdonságbeállítással. További előny az, hogy az így létrehozott felület könnyebben kezelhető, ha módosításra van szükség, ezt elegendő a sablon leírásában elvégezni, amely ezután érvényre jut a sablon teljes hatókörében.

Ilyen sablonokat az Android platform is támogat. Ha egyszerű widgetekre alkalmazott property halmazról van szó, elnevezése stílus (*Style*), míg az Activityre alkalmazott sablon elnevezése téma (*Theme*).

A következőkben megvizsgáljuk, hogyan hozhatunk létre egyszerű stílusokat, és hogyan használhatjuk fel őket a felülettervünkben. Ezután az Activitykre alkalmazható témák használatáról lesz szó. Bemutatjuk, hogyan alkalmazhatjuk a rendszerben előre definiált sablonokat, valamint összefoglaljuk, hogy milyen lehetőségünk van saját, egyedi témák készítésére.

3.6.1. Stílusok készítése

Saját stílusokat úgy készíthetünk, hogy a projektünk *res/values/styles.xml* erőforrás-állományában a megfelelő séma szerint definiáljuk őket. A gyökérelem kötelezően egy *<resources>* tag, amelynek gyermekelemeiként sorolhatjuk fel az alkalmazásunkban használandó *<style>* objektumokat. Egy stílusban tetszőleges *android:xyz* jellegű, megjelenést vezérlő tulajdonságnak adhatunk értéket. Hogy az elkészített stílusra az aktivitások felülettervéből

hivatkozni tudjunk, a *name* XML propertyvel kell hozzá szöveges azonosítót rendelnünk. A stílusok körében az öröklés is támogatott, a leszármazott stílus impliciten tartalmazza az őseiben elvégzett propertybeállításokat.

Az elmondottak alapján a stílusforrás-fájl általános felépítése a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name=stílus_neve [parent=szülő_stílus] >
    <item name=állítandó_property_neve>
      [beállítandó_érték]
    </item>
  </style>
</resources>
```

A beállított érték különféle típusú lehet, használhatunk szöveges értéket, betűméretet, színkódot, valamint tartalmazhat hivatkozást más erőforrás-elemekre (pl. képfájl). Az alkalmazott típus egyértelműen következik az állítandó property típusából. Például az *android:textColor* beállítása esetében a megadáshoz használható színkód (#AARRGGBB vagy egyéb elfogadott formátumban) vagy hivatkozás egy *Color* erőforrásra.

A létrehozott stílusokra a felülettervből a widget tulajdonságainak beállításánál hivatkozhatunk, a következőképpen:

```
<[Widget_típus]
  android:id="..."
  android:xxx="..."
  style="@style/[stílus_neve]" />
```

Készítsünk olyan stílust, amely úgy módosítja egy widget tulajdonságait, hogy a rajta szereplő szöveg 22 pontos, kék színű betűtípussal jelenjen meg.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="ExampleStyle">
  <item name="android:textSize">22sp</item>
  <item name="android:textColor">#0000EE</item>
</style>
</resources>
```

A kipróbáláshoz használjunk egy olyan felülettervet, amely a definiált stílust felhasználja. A felületen szerepeljen egy *TextView*, egy *Button* és egy *CheckBox*. Az XML-fájlban mindhárom widgetre alkalmazzuk az *ExampleStyle* stílust.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=
        "We applied the example style on this TextView."
    style="@style/ExampleStyle" />
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="And on this Button, too."
    style="@style/ExampleStyle" />
<CheckBox
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This CheckBox also uses it."
    style="@style/ExampleStyle" />
</LinearLayout>
```

Végül, amikor a felülettervet társítjuk egy aktivitáshoz, és futtatjuk a készüléken, a következőhöz hasonló eredményre jutunk:



3.17. ábra. Stílus alkalmazása

3.6.2. Témák készítése

A témák funkciója nagyon hasonlít a stílusokéhoz, az alapvető különbség az, hogy míg egy stílust minden esetben egy widgeten alkalmazunk, a témákat egy Activity-hez tartozó teljes *View*-hierarchiához rendelhetjük hozzá, megszabva a kijelzőn való megjelenési módját. A témák használatának az az előnye, hogy segítségükkel a teljes alkalmazás megjelenési stílusa könnyedén beállítható és módosítható.

A platformon elérhető előre definiált témák, amelyeket saját alkalmazásunkban is felhasználhatunk. Ha egy aktivításhoz nem rendelünk explicite témát, akkor az alapértelmezett *android.R.style.Theme*-ben leírt kinézetet társítja hozzá a rendszer. Ha ezen változtatni szeretnénk, megadhatjuk a felhasználandó témát a projekt *AndroidManifest.xml* állományában, például:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="hu.bute.daa.amorg.examples">
    <application ...>
        <activity
            android:name=".MyActivity"
            android:label="@string/app_name"
            android:theme="@android:style/Theme.Dialog"
        >
            ...
        </activity>
    </application>
</manifest>
```

A téma-hozzárendelés megadható ennél magasabb szinten is, ha az *android:theme* attribútumot nem az *<activity>*, hanem az *<application>* tagre alkalmazzuk. Ilyenkor a megadott téma az alkalmazáshoz tartozó minden aktivításon érvényben lesz.

Megfigyelhető még a fenti XML-részletből, hogy a témákat ugyancsak stílus-erőforrásként kezeli a rendszer, hiszen *@style/...* módon hivatkozhatunk rá. A példakódban a *style* előtt azért szerepel még az *android:* jelző is, mivel az erőforrásokra hivatkozás szabványos formája a *@[/package:]resource_type/resource_name* formátum. A *package* megadása opcionális. A *Theme.Dialog* alkalmazása hasznos lehet a saját programjainkban, hatására az Activity a képernyőn egy dialógusablakhoz hasonló kinézetet ölt. A dialógustémával felruházott Activity használata alternatív megoldás lehet a korábban bemutatott *AlertDialog* használata helyett. A következő ábra bemutatja a fenti példa futásának az eredményét egy egyszerű Activity-n.



3.18. ábra. Theme.Dialog alkalmazása

Az alkalmazott téma az XML-beli beállítás helyett megadható program-kódból is, ha szükséges. Ekkor gondoskodnunk kell arról, hogy a beállítást elvégző `setTheme()` metódus az aktivitás létrehozásakor, még a *View* elemek konstruálása előtt meghívódjon, hogy ezek renderelésénél már biztosan a megadott témát használja a rendszer.

```
public class MyActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        setTheme(android.R.style.Theme_Dialog);
        setContentView(R.layout.main);
    }
}
```

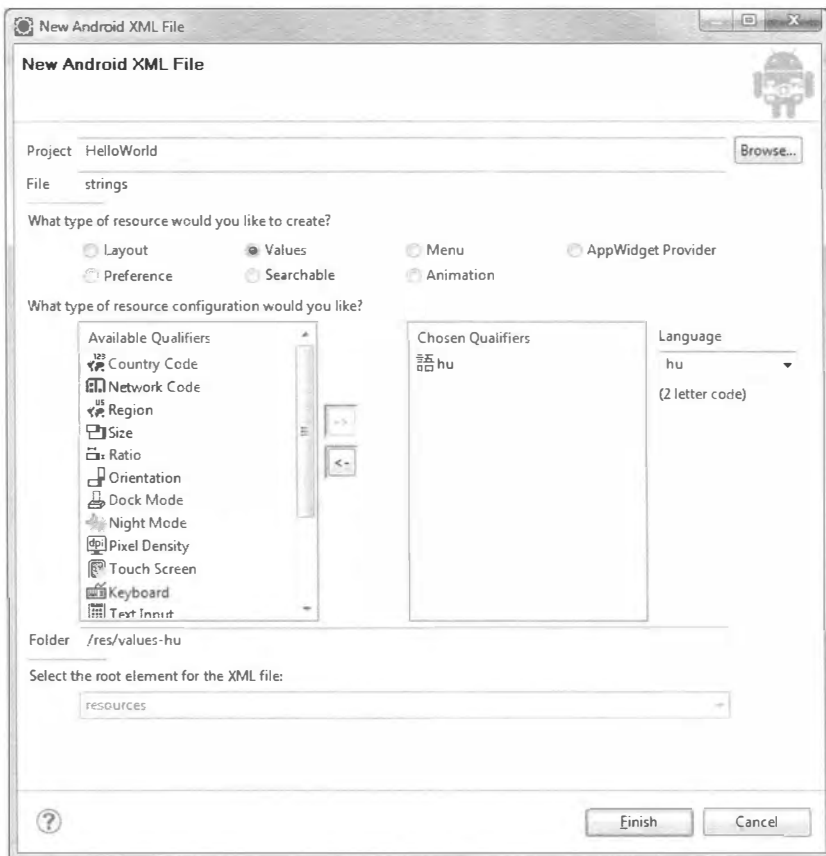
A rendszer nyújtotta beépített témák használata mellett arra is van lehetőségünk, hogy egyedi témákat definiáljunk, és felhasználjuk őket az alkalmazásunk felhasználói felületének testreszabására. A témák készítésénél a stílus megadásához nagyon hasonló módszert kell követnünk. Könyvünkben erre részletesen nem térünk ki, ám az Android hivatalos oldalán erre jó leírást találhatunk.¹²

¹² Témák kezelése: <http://developer.android.com/guide/topics/ui/themes.html>

3.7. Lokalizáció támogatása

Mobilalkalmazások fejlesztésekor sokszor szükség lehet arra, hogy egy alkalmazás felhasználói felülete többnyelvű legyen. Szerencsére az Android erre nagyon jó eszközt biztosít. Minden szöveges elem a `/res/values/strings.xml` állományban található. Lehetőségünk van a `values` könyvtárhoz is minősítőket adni, és a többnyelvűség támogatására a rendszer megengedi, hogy nyelvi minősítőket használjunk, így például létrehozhatunk `values-hu`, `values-pl` stb. könyvtárakat magyar, lengyel és bármilyen nyelv támogatására.

Érdekesség, hogy az Android nemcsak a `values` könyvtárhoz engedi meg a nyelvi minősítők hozzáadását, hanem bármilyen más erőforráskönyvtárhoz, így például megtehetjük azt is, hogy különböző nyelvű készülékeken különböző képeket vagy felületelrendezéseket alkalmazzunk. Új erőforrás létrehozásakor az Eclipse-ben egyszerűen be is állíthatjuk a nyelvi minősítőt.



3.19. ábra. Nyelvi minősítő választása erőforrás létrehozásakor

Ha emulátoron szeretnénk tesztelni a többnyelvűséget, ezt könnyedén megtehetjük, hiszen az emulátor lokalizációja is átállítható, valamint akár egyedi lokalizációs azonosítókat is felvehetünk.



3.20. ábra. Lokalizáció tesztelése emulátoron

3.8. További Android alkalmazáskomponensek

Az Android platform négyféle alkalmazáskomponens-típust különböztet meg. Két további speciális eset is létezik, amelyek az élő háttérképek (*live wallpaper*) és az úgynevezett asztalra elhelyezhető minialkalmazások (vagy widgetek, de ne keverjük össze a beépített *View* elemekkel). Mindkét speciális komponens működését érdemes megvizsgálni, hiszen sokszor hasznos eszközt jelenthetnek a fejlesztők kezében, illetve nem ritka, hogy egy alkalmazáshoz például egy egyszerűsített widgetet is kér a megrendelő. A következőkben ezek használatát ismertetjük és mutatjuk be egy-egy példán keresztül.

3.8.1. Élő háttérkép

Az élő háttérképek az Android 2.1-es verziójától érhetők el. Tipikusan egy animált, interaktív háttérképről van szó. Az élő háttérképekben rejlő lehetőségek azonban ennél sokkal nagyobbak, mivel el tudják érni az Android legtöbb funkcióját, például a 2D-s és 3D-s rajzolást, a GPS-t, a gyorsulásmérőt, a hálózati kommunikációt stb.

Az élő háttérkép leginkább egy *Service*-hez hasonlítható, ám rendelkezik felhasználói felülettel. Létrehozáskor az *onCreateEngine()* függvénye hívódik meg, amelyben vissza kell térnünk egy *WallpaperService.Engine* leszármazott osztállyal, amelyet mi implementáltunk. Ez az *Engine* felelős az élő háttérkép viselkedéséért. Az élő háttérképre való periodikus rajzoláskor ügyelnünk kell az optimális kirajzolási algoritmusra, ugyanis a magas CPU-használat gyorsabban lemeríti az akkumulátort, és ez elveheti a felhasználók kedvét attól, hogy használják. Ha élő háttérképet fejlesztünk, a *manifest* állományban ezt mindenképpen jelezni kell a következőképpen:

```
<uses-feature android:name=
    "android.software.live_wallpaper" />
```

Az *Engine* osztályban az alábbi események függvényeit definiálhatjuk felül:

- *onVisibilityChanged()*: A függvény akkor hívódik meg, amikor már nem látszik a háttérkép. Ebben az esetben függesszünk fel minden tevékenységet.
- *onOffsetsChanged()*: Több háttér esetén a „home screen”-ek közti váltás esetén hívódik meg a függvény.
- *onTouchEvent()*: Képernyő-érintési eseményt jelző függvény.
- *onCommand()*: Általa más alkalmazás küldhet utasítást a háttérképnek, ám jelenleg csak a beépített *Home* alkalmazás tudja ezt megtenni.

A következőkben bemutatjuk egy élő háttérkép elkészítésének a lépéseit. A példaalkalmazásunkban készítsünk egy háttérképet, amely fekete hátteret jelenít meg, és érintéskor szövegesen az érintés helyén kiírja az aktuális dátumot és az időt.

Az élő háttérkép *manifest* állományának tartalma a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="hu.bute.daa1.amorg.examples"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8"/>
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name">
        <service
            android:label="@string/app_name"
```

```

        android:name=".MyWallpaper"
        android:permission=
            "android.permission.BIND_WALLPAPER">
        <intent-filter>
            <action android:name=
                "android.service.wallpaper.
WallpaperService"/>
        </intent-filter>
        <meta-data android:name=
            "android.service.wallpaper"
            android:resource="@xml/mywall"/>
        </service>
    </application>
</manifest>

```

A *manifest* állomány alapján látható, hogy maga a háttérkép-szolgáltatás a *MyWallpaper* osztályban van megvalósítva, amelynek tartalma esetünkben a következő:

```

public class MyWallpaper extends WallpaperService {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    public Engine onCreateEngine() {
        return new CubeEngine();
    }

    class CubeEngine extends Engine {
        private final Paint mPaint = new Paint();
        private float mTouchX = 0;
        private float mTouchY = 0;

        CubeEngine() {
            final Paint paint = mPaint;
            paint.setColor(0xffffffff);
            paint.setTextSize(25);
        }
    }
}

```



```

        @Override
        public void onCreate(SurfaceHolder
surfaceHolder) {
            super.onCreate(surfaceHolder);
            // Érintés eseményre kezelés jelzése
            setTouchEventsEnabled(true);
        }

        @Override
        public void onTouchEvent(MotionEvent event) {
            super.onTouchEvent(event);
            if (event.getAction() == MotionEvent.ACTI-
ON_UP) {
                mTouchX = -1;
                mTouchY = -1;
            } else {
                mTouchX = event.getX();
                mTouchY = event.getY();
            }
            drawFrame();
        }

        void drawFrame() {
            final SurfaceHolder holder =
getSurfaceHolder();
            Canvas c = null;
            try {
                c = holder.lockCanvas();
                if (c != null) {
                    c.save();
                    c.drawColor(0xff000000);
                    c.drawText(new Date(
                        System.currentTimeMillis()).
                        toLocaleString(), mTouchX,
                        mTouchY, mPaint);
                    c.restore();
                }
            } finally {
                if (c != null)
                    holder.unlockCanvasAndPost(c);
            }
        }
    }
}

```

Példánkban a *CubeEngine* osztály valósítja meg a háttérkép működését. A *CubeEngine* osztály *onTouchEvent()* függvényében elmentjük az érintés helyét, a *drawFrame()* függvényben pedig megjelenítjük erre a helyre az aktuális időt. Az elkészült élő háttérképet a következő ábra szemlélteti.



3.21. ábra. Élő háttérképpélda

3.8.2. Widget

A *widget*ek tulajdonképpen olyan kis alkalmazások, amelyeket Android-alapú telefonokon a kezdő képernyőre helyezhetünk el kis méretben. Feladatuk az, hogy például friss információt nyújtsanak egy adott dologról, illetve a készülék vagy valamilyen alkalmazás funkcióit könnyen el lehessen érni általuk.

Widget készítésekor az alábbi komponenseket kell megvalósítanunk:

- *AppWidgetProviderInfo*: XML-ben adott metaállomány, amely a widgettel kapcsolatos különféle információkat tartalmazza (layout, frissítés gyakorisága, widget üzleti logikájáért felelős *AppWidgetProvider* osztály).
- *AppWidgetProvider*: A widget üzleti logikájáért felelős osztály, amely tulajdonképpen a programozói felületet biztosítja, továbbá értesítést kap különféle *Broadcast* eseményekről, ilyen például a frissítés, az engedélyezés, a tiltás, az eltávolítás.
- *View layout*: A widget felületét meghatározó XML-felület-leírás.

- Konfigurációs Activity: Opcionális Activity, amelynek segítségével a widgettel kapcsolatos beállításokat ejthetjük meg. A widget hozzáadásakor automatikusan elindul.

A következőkben nézzünk meg egy widgetpéldát. A widget feladata az aktuális dátum és idő megjelenítése és frissítése periodikusan, valamint érintéseménykor az azonnali frissítés.

A widgethez tartozó *manifest* állomány a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="hu.bute.daa.amorg.examples"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" />
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <receiver android:name=
            ".HelloWidget" android:label="@string/app_
name">
            <intent-filter>
                <action android:name=
                    "android.appwidget.action.APPWIDGET_
UPDATE" />
            </intent-filter>
            <meta-data android:name=
                "android.appwidget.provider"
                android:resource=
                    "@xml/hello_widget_provider" />
            </receiver>
        </application>
    </manifest>
```

A *manifest* állomány alapján látható, hogy a widgethez tartozó *AppWidgetProvider* XML a *hello_widget_provider* XML-állományban található.

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:updatePeriodMillis="300000"
    android:initialLayout="@layout/main"
/>
```

A *hello_widget_provider*ben láthatjuk, hogy a widgetfelület leírása a *main.xml*-ben található, amelynek tartalma a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="6dip"
    android:background="@drawable/myshape">
    <TextView
        android:id="@+id/tvStatus"
        style="@android:style/TextAppearance.Small"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center"
        android:gravity="center_horizontal|center_
vertical"
        android:layout_margin="4dip"
        android:text="" />
    </LinearLayout>
```

A *main.xml*-ben figyeljük meg, hogy a widget hátterét szintén egy erőforrás-ként adtuk meg, ez esetünkben nem egy kép lesz, hanem egy XML-ben összeállított, lekerekített sarkú, színátmenetes téglalap. A *myshape.xml* tartalma tehát a következő:

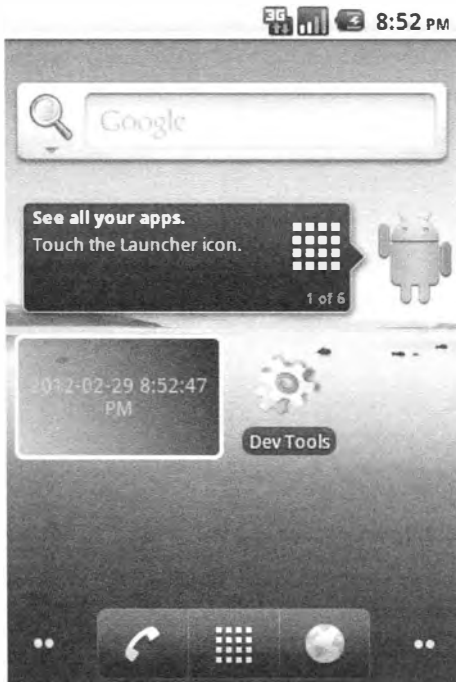
```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >
    <stroke
        android:width="3dp"
        android:color="#FFFFFF" />
    <gradient
        android:angle="225"
        android:endColor="#DDEEBB88"
        android:startColor="#DD000000" />
    <corners
        android:bottomLeftRadius="5dp"
        android:bottomRightRadius="5dp"
        android:topLeftRadius="5dp"
        android:topRightRadius="5dp" />
    </shape>
```

A *manifest* állományból ugyancsak látható volt, hogy a widget üzleti logikáját megvalósító osztály esetünkben a *HelloWidget* osztály, amelynek tartalma a következő:

```
public class HelloWidget extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context,
        AppWidgetManager appWidgetManager,
        int[] appWidgetIds)
    {
        // ID-k lekérdezése
        ComponentName thisWidget = new
        ComponentName(context,
            HelloWidget.class);
        int[] allWidgetIds = appWidgetManager.
        getAppWidgetIds(
            thisWidget);
        for (int widgetId : allWidgetIds) {
            RemoteViews remoteViews =
                new RemoteViews(context.
        getPackageName(),
            R.layout.main);
            // Szöveg beállítása
            remoteViews.setTextViewText(R.id.tvStatus,
                new Date(System.currentTimeMillis()
                ).toLocaleString());
            // Kattintás hatására frissül ismét a
        widget
            Intent intent =
                new Intent(context, HelloWidget.
        class);
            intent.setAction(
                AppWidgetManager.ACTION_APPWIDGET_UPDA-
        TE);
            intent.putExtra(
                AppWidgetManager.EXTRA_APPWIDGET_IDS,
                appWidgetIds);
            PendingIntent pendingIntent =
                PendingIntent.getBroadcast(context, 0,
        intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
            remoteViews.setOnClickPendingIntent(
                R.id.tvStatus, pendingIntent);
            appWidgetManager.updateAppWidget(
                widgetId, remoteViews);
        }
    }
}
```

Az osztályban az *onUpdate()* függvényt definiáltuk felül, amelyben egyrészt frissítettük a widgeten lévő *TextView* tartalmát az aktuális dátumra, másrészt összeállítottunk egy *PendingIntent* objektumot, amely a widgetre való kattintáskor fut le, és azonnal frissíti a widgetet. A kattintás eseményét egy *PendingIntent*tel valósítottuk meg, ez egy olyan *Intent*, amelynek elküldése csak a kattintásesemény bekövetkeztekor jön létre.

Az elkészült widgetet a következő ábra szemlélteti.



3.22. ábra. Widgetpélda

3.9. Összetettlista-alapú alkalmazás készítése

Az eddigiek összefoglalásaként tekintsük át egy összetett alkalmazás elkészítésének a lépéseit. Az alkalmazásban egy teljes képernyőlistát is használunk, amelynek megvalósításához a *ListActivity* osztály működését is bemutatjuk.

A *ListActivity* osztály tipikusan egy teljes képernyős lista megjelenítésekor használatos. Az *Activity*ktől megszokott képességeken kívül a *ListActivity* alapértelmezetten tartalmaz egy *ListView*-t, amelyet a *getListView()* függvényvel kérhetünk el, és egy *ListAdapter*t, amely a lista elemeinek tárolásáért és megjelenítésért felelős; a *ListActivity* működése így az adatkötési elveket követi. A *ListAdapter* tipikusan egy *BaseAdapter*ből leszármazó osztály, amely

tartalmazza az elemeket (objektumok listája), felelős az új elem hozzáadásáért, eléréséért és törléséért, valamint a *getView(...)* függvény felüldefiniálásával megadhatjuk, hogy a lista egy sorában egy elem hogyan renderlődjön ki. A *getView(...)* függvény paraméterül kap egy kirenderelendő objektumot (pl. egy *Todo* objektumot). A *getView(...)* függvényben tehát tipikusan kiválasztunk egy XML-ben leírt elrendezést a sorra, és beállítjuk az abban lévő felületi elemek tartalmát, képeket stb. Így megadhatjuk, hogy a lista elemei hogyan nézzenek ki, és nemcsak standard egysoros listákat hozhatunk létre, hanem gyakorlatilag korlátlan szabadsággal rendelkezünk.

Példánkban egy *Todo* (tennivalók) listaalapú alkalmazást hozunk létre. Az alkalmazás indításakor egy *ListActivity* jelenik meg, amely tartalmazza a *Todo* elemeket.

Egy *Todo*-ról az alábbiakat tároljuk:

- cím (title),
- prioritás (LOW, MEDIUM, HIGH),
- esedékesség dátuma (duedate),
- leírás (description).

Egy listaelem kinézete a következő: bal oldalt a prioritásnak megfelelő ikon látszik, jobb oldalt pedig felül a *Todo* címe, alatta pedig kisebb betűvel az esedékesség dátuma.

Egy listaelemre röviden kattintva (*getListView().setOnClickListener(...)*) a *Todo* elem leírása jelenik meg egy *Toast* ablakban.

Egy listaelemre hosszan kattintva (*registerForContextMenu(getListView());*) egy helyi menü jelenik meg *Delete* és *Back* gombokkal. A *Delete* gombot választva törlődik a kiválasztott listaelem.

A *ListActivity* emellett egy saját menüvel rendelkezik, amelyben egy „Create new *Todo*” menüpont található, ezt kiválasztva dialógus formában egy új *Activity* jelenik meg, ahol egy *TableLayout* elrendezésen a létrehozandó új *Todo* adatait adhatjuk meg. Az új *Todo* esedékességi dátumát egy dátumválasztó dialógussal (*DatePickerDialog*) oldjuk meg. A tervezendő alkalmazás felhasználói felületét a következő ábra szemlélteti.



3.23. ábra. Todo alkalmazás

Első lépésként hozzuk létre a szöveges elemeket tartalmazó erőforrás-állományt. Helyezzük el a *values/strings.xml*-be az alábbi tartalmat:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Todo list</string>

    <string name="itemCreateTodo">Create new Todo</string>

    <string name="lblTodoTitle">Title:</string>
    <string name="lblTodoPriority">Priority:</string>
    <string name="lblTodoDueDate">Due date:</string>
    <string name="lblTodoDescription">Description:</string>

    <string name="btnOk">Ok</string>
    <string name="btnCancel">Cancel</string>
</resources>
```


Ezt követően hozzuk létre a *Todo* osztályt, ezt példányosítva fogjuk *Java PoJo* (Plain Old Java Object) objektumokként tárolni a *Todo* elemeket. Figyeljük meg a prioritásokat leíró *Priority* enumerációt.

```
public class Todo {

    public enum Priority { LOW, MEDIUM, HIGH }
    private String title;
    private Priority priority;
    private String dueDate;
    private String description;
    public Todo(String aTitle, Priority aPriority,
        String aDueDate, String aDescription)
    {
        title = aTitle;
        priority = aPriority;
        dueDate = aDueDate;
        description = aDescription;
    }

    public String getTitle() {
        return title;
    }

    public Priority getPriority() {
        return priority;
    }

    public String getDueDate() {
        return dueDate;
    }

    public String getDescription() {
        return description;
    }
}
```

Következő lépésként hozzunk létre egy *TodoAdapter* osztályt, amely a *BaseAdapter*-ből öröklődik. Az osztály feladata a létrehozott *Todo* elemek tárolása, kezelése és a listában való megjelenítési módjuk megadása a *getView(...)* függvényben.

```
public class TodoAdapter extends BaseAdapter {

    private final List<Todo> todos;

    public TodoAdapter(final Context context,
        final ArrayList<Todo> aTodos) {
        todos = aTodos;
    }

    public void addItem(Todo aTodo)
    {
        todos.add(aTodo);
    }

    public int getCount() {
        return todos.size();
    }

    public Object getItem(int position) {
        return todos.get(position);
    }

    public long getItemId(int position) {
        return position;
    }

    // Sor megjelenítésének beállítása
    public View getView(int position, View
convertView,
        ViewGroup parent) {
        final Todo todo = todos.get(position);
        LayoutInflater inflater =
            (LayoutInflater) parent.getContext().
            getSystemService(
                Context.LAYOUT_INFLATER_SERVICE);
        View itemView = inflater.inflate(
            R.layout.todorow, null);
        ImageView imageViewIcon = (ImageView)
            itemView.findViewById(R.
id.imageViewPriority);
        switch (todo.getPriority()) {
            case LOW:
                imageViewIcon.setImageResource(
```

```

        R.drawable.low);
        break;
    case MEDIUM:
        imageViewIcon.setImageResource(
            R.drawable.medium);
        break;
    case HIGH:
        imageViewIcon.setImageResource(
            R.drawable.high);
        break;
    default:
        imageViewIcon.setImageResource(
            R.drawable.high);
        break;
    }
    TextView textViewTitle =
        (TextView) itemView.findViewById(R.
id.textViewTitle);
    textViewTitle.setText(todo.getTitle());
    TextView textViewDueDate =
        (TextView) itemView.findViewById(R.
id.textViewDueDate);
    textViewDueDate.setText(todo.getDueDate());
    return itemView;
}

public void deleteRow(Todo aTodo) {
    if(todos.contains(aTodo)) {
        todos.remove(aTodo);
    }
}
}

```

A *getView(...)* függvényben figyeljük meg, hogy állítjuk be az elrendezést a *LayoutInflater* használatával. Ebben a függvényben hivatkozunk a *R.layout.todorow* erőforrásra, amelynek tartalma (*layout/todorow.xml*) határozza meg a lista egy sorának az elrendezését.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <ImageView

```

```

        android:id="@+id/imageViewPriority"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/high"
        android:padding="5dp"/>
    <RelativeLayout
        android:layout_height="wrap_content"
        android:layout_width="fill_parent">
        <TextView
            android:id="@+id/textViewTitle"
            android:textSize="16dp"
            android:text="Title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"/>
        <TextView
            android:id="@+id/textViewDueDate"
            android:textSize="12dp"
            android:text="DueDate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/textViewTitle"
            android:layout_alignParentBottom="true"
            android:gravity="bottom"/>
    </RelativeLayout>
</LinearLayout>

```

Továbbá szintén a *getView(...)* függvényben hivatkozunk három képre (*R.drawable.low/medium/high*), amelyek beszerzése szabadon választható. Példánkban három egyszerű képet alkalmaztunk, ahol a különböző színek a különböző prioritásokat jelentették.

A listaelemek megjelenítéséhez készítsuk el a *ListActivity* osztályunkat, amely a korábban bemutatott *TodoAdapter*t használja fel az adatok kezeléséhez. A *ListActivity onCreate(...)* függvényében hozzuk létre az *Adapter*t, adjunk hozzá néhány példaelemet, majd pedig állítsuk be a *ListActivity*nek *Adapter*ként. Ezután a *ListView*-nak állítsunk be egy *OnItemClickListener*t, amelyben megvalósítjuk, hogy a *Todo description* mezője jelenjen meg egy *Toast*-ban.

```

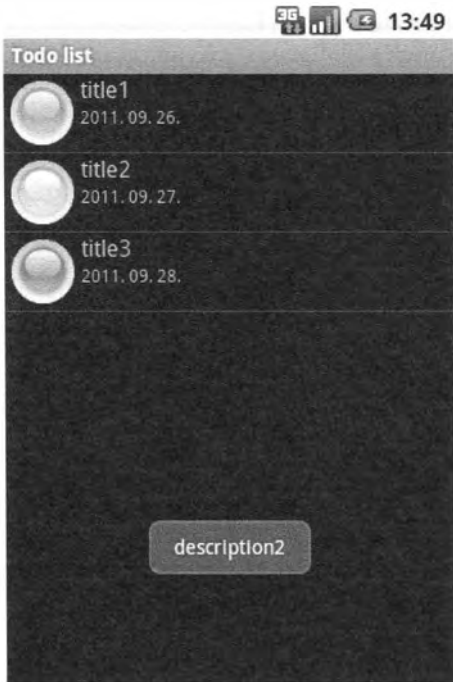
public class ActivityMain extends ListActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Adapter létrehozása és feltöltése néhány
        elemmel
        ArrayList<Todo> todos = new ArrayList<Todo>();
        todos.add(new Todo("title1", Priority.LOW,
            "2011. 09. 26.", "description1"));
        todos.add(new Todo("title2", Priority.MEDIUM,
            "2011. 09. 27.", "description2"));
        todos.add(new Todo("title3", Priority.HIGH,
            "2011. 09. 28.", "description2"));
        TodoAdapter todoAdapter = new TodoAdapter(
            getApplicationContext(), todos);
        setListAdapter(todoAdapter);

        // Elemre kattintás eseményre feliratkozás
        getListView().setOnItemClickListener(
            new.OnItemClickListener() {
                public void onItemClick(AdapterView
parent,
                    View v, int position, long id) {
                    Todo selectedTodo =
                        (Todo)getListAdapter().
getItem(position);
                    Toast.makeText(ActivityMain.this,
                        selectedTodo.getDescription(),
                        Toast.LENGTH_LONG).show();
                }
            });
    }
}

```

Figyeljük meg, hogy az *onCreate()* függvényben létrehozunk 3 kezdőelemet a listánkban.



3.24. ábra. Todo-lista

Következő feladatként valósítsuk meg a listaelem törlését. A konkrét feladat az, hogy egy listaelemre hosszan kattintva (*ListActivity*-ben: *registerForContextMenu(getListView());*) jelenjen meg egy helyi menü *Delete* és *Back* menüpontokkal, és a *Delete*-et választva törlődjön a kiválasztott elem.

Adjuk az alábbi sort a *ListActivity.onCreate(...)* függvényének végéhez:

```
registerForContextMenu(getListView());
```

Definiáljuk felül az *onCreateContextMenu(...)* és az *onContextItemSelected(...)* függvényeket.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View
v,
ContextMenuInfo menuInfo) {
    if (v.equals(getListView())) {
        AdapterView.AdapterContextMenuInfo info =
```

```

        (AdapterView.AdapterContextMenuInfo)menuInfo;
        menu.setHeaderTitle(((Todo)
            getListAdapter().getItem(info.position)).
getTitle());
        String[] menuItems = getResources().
getStringArray(
            R.array.todomenu);
        for (int i = 0; i<menuItems.length; i++) {
            menu.add(Menu.NONE, i, i, menuItems[i]);
        }
    }
}

@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterView.AdapterContextMenuInfo info =
        (AdapterView.AdapterContextMenuInfo)item.
getMenuInfo();
    int menuItemIndex = item.getItemId();
    if (menuItemIndex==0)
    {
        ((TodoAdapter)getListAdapter()).
deleteRow(((Todo)getListAdapter().getItem(
            info.position)));
        ((TodoAdapter)getListAdapter()).
notifyDataSetChanged();
    }
    return true;
}
}

```

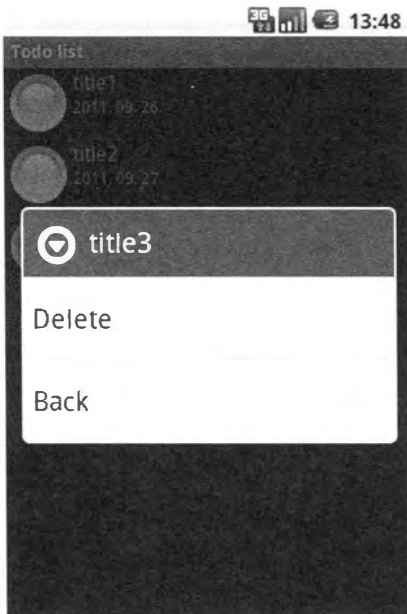
Az *onCreateContextMenu(...)* függvényben hivatkozunk a *R.array.todomenu* erőforrásra, amely egy szöveges tömböt ír le. Hozzunk létre egy *todomenu.xml* nevű állományt a *res/values* könyvtárba, amelynek tartalma legyen a következő:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array
        name="todomenu">
        <item>Delete</item>
        <item>Back</item>
    </string-array>
</resources>

```

Jól látható tehát, hogy az Android erőforrásrendszere rendkívül gazdag, és például tömböket is létrehozhatunk általa.



3.25. ábra. Todo elem törlése

Példánk következő lépéseként valósítsuk meg a fekvőnézet támogatását úgy, hogy fekvőnézetben a *Todo* listájában a tennivaló címe és dátuma ne egymás alatt, hanem egy sorban jelenjen meg. Ehhez hozzunk létre egy *layout-land* könyvtárat a *res* könyvtáron belül és benne egy új *todorow.xml*-t, amelynek tartalma a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/
  android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">
  <ImageView
    android:id="@+id/imageViewPriority"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:src="@drawable/high"
    android:padding="5dp"/>
  <LinearLayout
    android:layout_height="wrap_content"
```



```

        android:layout_width="fill_parent"
        android:orientation="horizontal">
        <TextView
            android:id="@+id/textViewTitle"
            android:textSize="16dp"
            android:text="Title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <TextView
            android:id="@+id/textViewDueDate"
            android:textSize="12dp"
            android:text="DueDate"
            android:paddingLeft="10dp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
    </LinearLayout>
</LinearLayout>

```



3.26. ábra. Fekvő nézet támogatása

Utolsó nagyobb lépésként valósítsuk meg az új *Todo* elem felvételét, hogy teljes legyen az alkalmazásunk. Hozzunk létre egy új *ActivityCreateTodo* Activityt, amely egy *TableLayout*-ban tartalmazza az új *Todo* létrehozásához szükséges elemeket. A *manifest* állományba vegyük fel az új Activityt.

```

<activity android:name=".ActivityCreateTodo"
    android:label="@string/itemCreateTodo"
    android:theme="@android:style/Theme.Dialog">
</activity>

```

Az új *ActivityCreateTodo* forrása a következőkben látható. Figyeljük meg, hogy működik a dátumkiválasztó dialógus. A dialógus megjelenítéséhez meghívjuk az *Activity showDialog(ID)* függvényét, amelyben átadunk egy egyedi dialógusazonosítót. Az *Activity* ezután az *onCreateDialog(...)* függvényt hívja meg, amelyben megadjuk, hogy az általunk megadott ID-hez milyen dialógust hozzon létre. Végül pedig az *mDateSetListener* objektum megadásával leírjuk, hogy mi történjen a dátum kiválasztásakor.

```
public class ActivityCreateTodo extends Activity {

    private static final int DATE_DUE_DIALOG_ID = 1;

    private Calendar calSelectedDate = Calendar.
getInstance();

    private EditText editTodoTitle;
    private Spinner spnrTodoPriority;
    private TextView txtDueDate;
    private EditText editTodoDescription;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.createtodo);

        // aktuális dátum beállítása
        calSelectedDate.setTime(
            new Date(System.currentTimeMillis()));
        // UI elem referenciák elkérése
        editTodoTitle =
            (EditText) this.findViewById(R.id.todoTitle);
        spnrTodoPriority =
            (Spinner) this.findViewById(R.id.todoPriority);
        String[] priorities=new String[3];
        priorities[0]="Low";
        priorities[1]="Medium";
        priorities[2]="High";
        spnrTodoPriority.setAdapter(new
        ArrayAdapter(this,
            android.R.layout.simple_spinner_item,
            priorities));
        txtDueDate =
            (TextView) this.findViewById(R.id.todoDueDate);
        refreshDateText();
        txtDueDate.setOnClickListener(new
        OnClickListener() {
            public void onClick(View v) {
```

```

        showDialog(DATE_DUE_DIALOG_ID);
    }
});
editTodoDescription =
    (EditText)this.findViewById(R.
id.todoDescription);
    // gomb eseménykezelők kezdeti beállítása
    Button btnOk =
        (Button)findViewById(R.id.btnCreateTodo);
    btnOk.setOnClickListener(new OnClickListener()
    {
        public void onClick(View v) {
            finish();
        }
    });
    Button btnCancel = (Button)findViewById(
        R.id.btnCancelCreateTodo);
    btnCancel.setOnClickListener(new
OnClickListener() {
        public void onClick(View v) {
            finish();
        }
    });
}

private void refreshDateText()
{
    StringBuilder dateString = new
StringBuilder();
    dateString.append(calSelectedDate.
get(Calendar.YEAR));
    dateString.append(". ");
    dateString.append(
        calSelectedDate.get(Calendar.MONTH)+1);
    dateString.append(". ");
    dateString.append(
        calSelectedDate.get(Calendar.DAY_OF_MONTH));
    dateString.append(".");
    txtDueDate.setText(dateString.toString());
}

@Override
protected Dialog onCreateDialog(int id)
{
    switch (id) {
        case DATE_DUE_DIALOG_ID:
        {
            return new DatePickerDialog(
                this, mDateSetListener,

```

```

        calSelectedDate.get(Calendar.YEAR),
        calSelectedDate.get(Calendar.MONTH),
        calSelectedDate.get(Calendar.DAY_OF_
MONTH));
    }
}
return null;
}

// mDateSetListener dátum kiválasztó esemény men-
tése
// attribútum formájában
private DatePickerDialog.OnDateSetListener
mDateSetListener =
    new DatePickerDialog.OnDateSetListener()
    {
        public void onDateSet(DatePicker view,
            int year, int monthOfYear, int
dayOfMonth)
        {
            // Új dátum beállítása
            calSelectedDate.set(Calendar.YEAR,
year);
            calSelectedDate.set(
                Calendar.MONTH, monthOfYear);
            calSelectedDate.set(
                Calendar.DAY_OF_MONTH, dayOfMonth);
            refreshDateText();
            removeDialog(DATE_DUE_DIALOG_ID);
        }
    };
}

```

A bemutatott *ActivityCreateTodo* layoutját a *createtodo.xml*-ben adjuk meg.

```

<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:stretchColumns="1">
    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="@string/lblTodoTitle"
            android:layout_width="wrap_content"
            android:gravity="right"/>
    </TableRow>
</TableLayout>

```

```

        <EditText
            android:id="@+id/todoTitle"
            android:width="200dp"/>
    </TableRow>
    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="@string/lblTodoPriority"
            android:layout_width="wrap_content"
            android:gravity="right"/>
        <Spinner
            android:id="@+id/todoPriority"
            android:width="200dp"/>
    </TableRow>
    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="@string/lblTodoDueDate"
            android:layout_width="wrap_content"
            android:gravity="right"/>
        <TextView
            android:id="@+id/todoDueDate"
            android:textSize="20dp"
            android:width="200dp"
            android:gravity="center"/>
    </TableRow>
    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="@string/lblTodoDescription"
            android:layout_width="wrap_content"
            android:gravity="right"/>
        <EditText
            android:id="@+id/todoDescription"
            android:width="200dp"/>
    </TableRow>
    <TableRow>
        <Button
            android:id="@+id/btnCreateTodo"
            android:layout_column="1"
            android:text="@string/btnOk"
            android:layout_width="wrap_content"
            android:gravity="right"/>
        <Button
            android:id="@+id/btnCancelCreateTodo"
            android:text="@string/btnCancel"
            android:layout_width="wrap_content"
            android:gravity="left"/>
    </TableRow>
</TableLayout>

```

Az elkészített, tennivalót létrehozó felületet a következő képernyőképek szemléltetik.



3.27. ábra. Új Todo létrehozása

A *Todo* elmentéséhez ebben a példában egy egyszerűsített, „csúnya” megoldást választunk, a helyes megoldást a perzisztenciafejezet elolvasása után tudjuk megvalósítani (ott kiderül, hogyan kell a telefon adatbázisába adatokat menteni és onnan kiolvasni).

A jelenlegi egyszerűsített megoldásban létrehozunk egy *DataPreferences* osztályt, amely tartalmaz egy *public static Todo todoToCreate = null;* objektumot. A *Todo*-t létrehozó Activity *OK* gombjának választásakor ebben az objektumban mentjük el a létrehozandó *Todo*-t, és az eredeti *ListActivity onResume()* függvényét felüldefiniálva megvizsgáljuk ennek a *todoToCreate* objektumnak a nullitását. Amennyiben nem *null*, felvesszünk egy új *Todo* elemet, és *nullra* állítjuk a *todoToCreate*-et. A *Todo*-t létrehozó Activity *Cancel* gombjára szintén *nullra* állítjuk a *todoToCreate* objektumot.

A *DataPreferences* osztály a következő:

```
public class DataPreferences {
    public static Todo todoToCreate = null;
}
```

Az *OK* és a *Cancel* gombok eseménykezelői a *Todo*-t létrehozó Activity *onCreate()* függvényében a következő:


```

// gomb eseménykezelők beállítása
Button btnOk = (Button)findViewById(R.
id.btnCreateTodo);
btnOk.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        Todo.Priority selectedPriority = Todo.
Priority.LOW;

        switch (spnrTodoPriority.
getSelectedItemPosition()) {
            case 0:
                selectedPriority = Todo.Priority.LOW;
                break;
            case 1:
                selectedPriority = Todo.Priority.MEDIUM;
                break;
            case 2:
                selectedPriority = Todo.Priority.HIGH;
                break;
            default:
                break;
        }

        // NAGYON RONDA MEGOLDÁS
        // A helyes működés megvalósításához az adat-
tárolásról
        // szóló fejezet anyagának elsajátítása szük-
séges
        DataPreferences.todoToCreate = new Todo(
            editTodoTitle.getText().toString(),
            selectedPriority,
            txtDueDate.getText().toString(),
            editTodoDescription.getText().
toString()
        );
        // befejezzük az Activity-t
        finish();
    }
});

Button btnCancel = (Button)findViewById(R.
id.btnCancelCreateTodo);
btnCancel.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        DataPreferences.todoToCreate = null;
        finish();
    }
});

```

Végül *Todo*-k listáját tartalmazó *ListActivity onResume(...)* függvénye, amely a tennivalót létrehozó Activityből való visszatéréskor hívódik meg az esetünkben, a következő:

```
@Override
protected void onResume() {
    super.onResume();

    // NAGYON RONDA MEGOLDÁS
    // A helyes működés megvalósításához az adattárolásról
    // szóló fejezet anyagának elsajátítása szükséges
    if (DataPreferences.todoToCreate != null)
    {
        ((TodoAdapter)getListAdapter()).
            addItem(DataPreferences.todoToCreate);
        DataPreferences.todoToCreate = null;
        ((TodoAdapter)getListAdapter()).
            notifyDataSetChanged();
    }
}
```

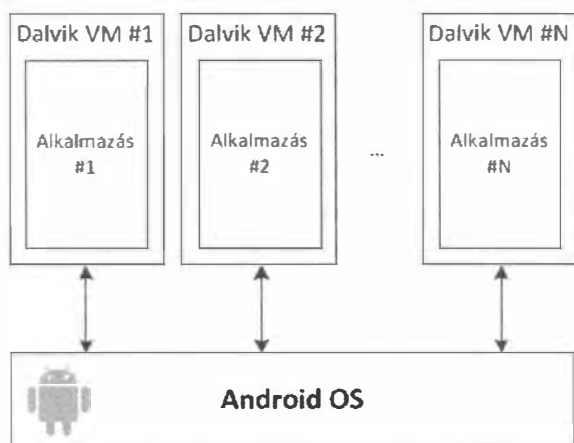


3.28. ábra. Új tennivaló elmentése

Mivel a *Todo* elemeket jelenleg nem mentjük el perzisztensen adatbázisba, ezért a képernyő elforgatásakor újra létrejön az Activity, és emiatt törölődnek az eddig létrehozott *Todo* elemek. Az adatkezeléssel foglalkozó fejezetben kitérünk az adatbáziskezelésre, és kibővítjük a példánkat.

Komponensek közti kommunikáció

A legtöbb mobilplatformon az alkalmazások egymástól jól elkülönítve, saját *sandbox*ukban futnak, szigorú korlátozásokkal egymás, az operációs rendszer natív komponensei és a hardvererőforrások elérésére. Ennek elsődleges célja a processzek védelme, egyrészt adatbiztonsági szempontból, másrészt pedig annak megelőzésére, hogy valamelyik alkalmazás abnormális leállása magával rántsa a többit vagy az operációs rendszert. A problémára az Android tervezésekor is gondoltak, a megoldás itt is hasonló, mint a többi platformon. Minden Android-alkalmazás a saját Dalvik virtuális gép (Dalvik Virtual Machine, módosított Java Virtual Machine) példányában fut, így a processz elkülönítése itt is teljesül (lásd a következő ábrát). A platform azonban versenytársainál sokkal több lehetőséget biztosít az alkalmazáskomponensek közti kommunikáció megvalósítására. Ezek a jelen fejezetben részletesen tárgyalt *Intent*- és *broadcast* üzenetek, valamint a későbbiekben kifejtett *ContentProvider*.



4.1. ábra. Alkalmazások elkülönítése az Androidon

A továbbiakban bemutatjuk az Android által bevezetett és a mobil-operációsrendszerek között egyedinek számító *Intent* fogalmát és működését, valamint megmutatjuk a rendszerszintű események kezelésének és a komponensek közti kommunikációra történő használatának a módját és legjobb gyakorlatait.

4.1. Az Intent fogalma

Android operációs rendszer esetén az alkalmazások háromféle komponensből állhatnak, ezek a felhasználói felülettel rendelkező *Activity*, a háttérbeli futásra képes *Service*, valamint a rendszerszintű események kezelésére képes *BroadcastReceiver*. Egy alkalmazás tipikusan több különböző típusú komponenset tartalmaz.

Nézzünk meg egy igen egyszerű, jegyzetkészítő Android-alkalmazást, amely lehetőséget biztosít a felhasználónak új feljegyzések létrehozására és a meglévők listázására, szerkesztésére, törlésére. Ennek legkézenfekvőbb implementációja két Activityt foglal magában, amelyek közül a nyitóképernyő-komponens a jegyzetek listázására, a másik pedig azok szerkesztésére vagy új létrehozására szolgál. Már az ilyen egyszerű esetekben is szükség van a komponensek közti kommunikációra, hiszen egy feljegyzés szerkesztésekor a nyitóképernyőn lévő listát megjelenítő Activitynek el kell indítania a módosításért felelős társát, és tudatnia kell vele, hogy a felhasználó melyik jegyzetet választotta ki.

Android platformon az alkalmazáskomponensek adatcseréjének elsődleges módszere az Intentek használata. Ez nem más, mint egy Java-osztály, amelyre passzív adatstruktúráként tekintünk. Késői, futásidejű kötést valósít meg az alkalmazáskomponensek között. Az adat, amelyet hordoz, és így maga az objektum is mindig valamilyen esemény absztrakt leírására szolgál. Ez lehet *elvárt esemény*, amelyet az Intent hatására szeretnénk előidézni (jellemzően Activity vagy Service indítása, BroadcastReceiver regisztrálása). Ha a rendszerben *broadcast* üzenet keletkezik, akár az operációs rendszer által (például kimenő hívás, akkumulátorfeszültség alacsony stb.), akár egy alkalmazás kezdeményezésére (például egy e-mail applikáció rendszerszintű értesítést küldhet új levél érkezésekor), akkor az üzenet szintén Intent-objektumként van jelen a rendszerben, és jut el az arra feliratkozott komponensekhez. Ekkor tehát egy *bekövetkezett esemény* attribútumait tartalmazza.

Bármilyen okból is jött létre, az Intent sohasem közvetlenül adódik át a komponensek között, minden esetben az operációs rendszeren keresztül történik a kézbesítése. Vagy maga az Android generálja valamilyen rendszeresemény hatására, vagy pedig a futtató környezet kapja egy komponensből, és gondoskodik a célba juttatásáról (lásd az alábbi ábrát).

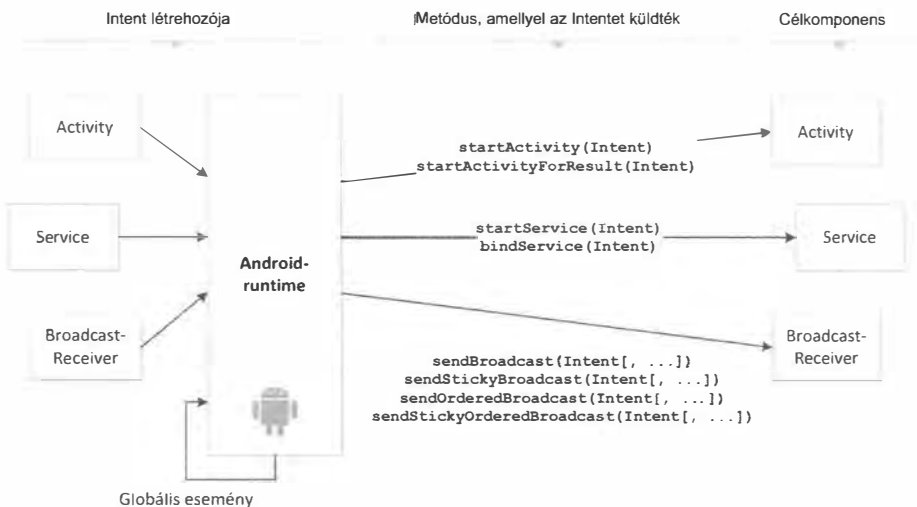


4.2. ábra. Intent kézbesítése az operációs rendszeren keresztül

Az Intent-objektum az Androidon a szabványos háromféle alkalmazáskomponens közti kommunikáció eszköze, továbbá a rendszer is ennek segítségével szolgáltat információkat a globális események bekövetkezéséről. Az Intent eredete lehet tehát *Activity*, *Service*, *BroadcastReceiver* vagy maga az operációs rendszer. A potenciális célpontok szintén ugyanezek a komponenstípusok, ám egy bizonyos módon a runtime-hoz került Intent mindig egy meghatározott osztály példányánál köthet ki, mégpedig a következőknél:

- Csak **Activity** lehet a fogadó objektum, ha bármilyen komponens *startActivity()* vagy *startActivityForResult()* metódussal küldte a rendszernek.
- **Service** lesz a célpont, hogyha az Intent célja szolgáltatás indítása a *startService()* metódussal vagy futásidejű kötés megvalósítása már futó szolgáltatáshoz a *bindService()* hívás hatására.
- A **BroadcastReceiver** az egyetlen osztály, amely képes fogadni és kezelni a *sendBroadcast()* metódus segítségével elküldött Intentet.

A helyzetet a következő ábra szemlélteti.



4.3. ábra. Az Intent lehetséges kiindulásai és céljai

Bárhol jött is létre az Intent-objektum, az Android megkeresi a megfelelő címzettet, szükség esetén példányosítja és elindítja, valamint átadja neki a híváshoz tartozó és az aktiválását kiváltó adatsomagot. Az Intent-konceptió egyik legnagyobb erőssége és egyben a platformok közti különlegessége is abban rejlik, hogy a megcélzott komponens nemcsak ugyanazon a processzen belül lehet, ahonnan az Intentet küldték, hanem más alkalmazás valamelyik

modulja is címezhető ilyen módon. Ebből természetesen az is következik, hogy saját applikációnkat is felkészíthetjük máshonnan küldött kérések fogadására és kiszolgálására, így az Android operációs rendszerre telepített alkalmazások egy nagy, lazán csatolt szolgáltatáshálózatot alkothatnak. A platform ezzel a mechanizmussal teremti meg a lehetőséget a gyári, előre telepített szoftverek lecserélhetőségére, hiszen egy harmadik féltől származó applikáció is képes lehet kiszolgálni bármilyen akciót igénylő Intentet. Nézzük meg az Intenek alapvető jellemzőit és lehetőségeit.

4.2. Intent felépítése

Az Intent nem más, mint egy olyan információcsomag, amely egy megtörtént vagy elvárt esemény leírását tartalmazza. Felhasználási módja sokrétű, így attribútumai is többféle kombinációban lehetnek kitöltve vagy üresen hagyva attól függően, hogy mit ír le, vagy éppen mire szeretnénk használni.

Intent-objektum					
Akció String	Adat Uri + típus-String	Komponens neve Osztálynév és Csomagnév	Kategóriák int konstansok	Extrák Bundle	Flagek int konstansok

4.4. ábra. Az Intent-objektum részei

Mezői között megtalálható a címzett komponens neve, a bekövetkezett vagy végrehajtandó akció azonosítója, az adat, amelyen ez értelmezve van, de tartalmazhat még egyedi kulcs-érték párokat (*extrák*), további megköteket a címzett komponenssel kapcsolatban vagy *flageket*, ha új Activity indítását kezdeményezzük az Intent segítségével. Ezeket az attribútumokat (lásd a következő ábrát) mindig olyan kombinációban kell kitöltenünk, amely az aktuális feladathoz vagy eseményhez szükséges, az API nem írja elő kötelezően egyik feltöltését sem. Nézzük meg a mezők részletes bemutatását.

Akció

Egy sztringkonstans az elvárt vagy *broadcast* hatására küldött Intent esetén a megtörtént eseményt jelzi. A platform rengeteg előre definiált, beépített akciót tartalmaz, de az alkalmazások egyedi konstansokat is definiálhatnak. Az operációs rendszer által használt akciók teljes listája folyamatosan bővül az új verziók kiadásával, a mindenkor legfrissebb kollekció mindig elérhető az API-referenciában, az Intent-osztály leírásánál.¹³ A fontosabb, leggyakrabban használt akciókat a következő táblázat mutatja be.

¹³ Intent-osztály dokumentációja: <http://developer.android.com/reference/android/content/Intent.html>

4.1. táblázat. Gyakori Intent-akciók

Konstans	Célpont típusa	Leírás
<code>ACTION_VIEW</code>	Activity	Az Intent adatmezőjében lévő entitás megnyitása olvasásra (például fájl, névjegy)
<code>ACTION_EDIT</code>	Activity	Az Intent adatmezőjében lévő entitás megnyitása szerkesztésre
<code>ACTION_PICK</code>	Activity	Választás az adatmezőben lévő URI által hivatkozott listából (például partner választása a névjegyzékből)
<code>ACTION_SEND</code>	Activity	Az adatmező tartalmának megosztása más alkalmazással
<code>ACTION_CALL</code>	Activity	Az adatmezőben átadott telefonszám felhívása
<code>ACTION_BATTERY_LOW</code>	Broadcast-Receiver	Akkufeszültség alacsony
<code>ACTION_BOOT_COMPLETED</code>	Broadcast-Receiver	A telefon bekapcsolt (ennek kezelésével lehet automatikusan indítani az alkalmazásunkat)
<code>ACTION_POWER_CONNECTED</code>	Broadcast-Receiver	Töltőre került az eszköz (ilyenkor érdemes például szinkronizálni a szerverrel, a cache-t karbantartani)

Fejlesztéskor mindig ellenőriznünk kell, hogy az alkalmazásunk által lekezelt akció az Android melyik verziójától számítva létezik. Ha ez magasabb, mint az alkalmazás által megkövetelt minimális API-szint, akkor progamozóként fel kell készülnünk arra, hogy nem minden eszközön lesz elérhető ez a bizonyos akció.

Hogyha például szeretnénk elérhetővé tenni alkalmazásunkat a könyv írásakor (2012. január) jó ár/érték aránya miatt népszerű, belépőszintű készülékre, a ZTE Blade-re, akkor az *app* minimális API-szintjét nem állíthatjuk 7 fölé (Android 2.1-eshez tartozó szint), ellenkező esetben a telefon megtagadná a telepítést. Ha ez a mobilapplikáció reagál a nyilvános tárhelyen erőforrásokkal rendelkező más alkalmazások elérhetetlenné válására (`ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE`), amelynek a telepítése csak a nyolcas szintnél (Android 2.2) épült be, akkor készülnie kell arra is, hogy ez az esemény soha nem következik be.

Az Intent-akciók elérhetősége az API-referenciában a konstansnévfejléc jobb szélén található (lásd az alábbi ábrát).

```
public static final String ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE
```

Sinon: API Level 2

Broadcast Action: Resources for a set of packages are currently unavailable since the media on which they exist is unavailable. The extra data `EXTRA_CHANGED_PACKAGE_LIST` contains a list of packages whose

4.5. ábra. API-verzió ellenőrzése

Az akciómező nemcsak az esemény azonosítására szolgál, hanem meghatározza a kérés feldolgozására képes komponens típusát (Activity vagy Broadcast-Receiver), valamint az Intent további – leginkább az adat és az extrák – attribútumait is erősen befolyásolja, mint ahogy egy metódus szignatúrája rögzíti annak nevén kívül a paramétereit és a visszatérési érték típusát is. Az akció konstans beállítására a `setAction()`, lekérésére a `getAction()` metódusok szolgálnak.

Adat

Ha az Intent által jelzett – megtörtént vagy igényelt – esemény önmagában nem értelmezhető, akkor a kapcsolódó adat URI-ja (Universal Resource Identifier, általános erőforrás-azonosító) és MIME-típusa¹⁴ szintén része az üzenetcsomagnak. Az akció meghatározhatja a mellette lévő adat típusát, hiszen például `ACTION_CALL` esetén csak „tel:/” kezdetű telefonszám, míg `ACTION_EDIT` beállításakor például dokumentum, névjegy vagy fénykép átadásának van értelme. A MIME-típus beállításának célja triviális, csak olyan komponens dolgozhat fel az Intentet, amely képes az URI által meghatározott típusú adat megfelelő kezelésére. Ez különösen igaz az általános jelentésű akciók esetében, hiszen egészen más feladatot jelent egy szöveges állományt és egy mp3-as fájlt szerkesztésre megnyitni. Persze az URI-ból gyakran egyértelműen következik a fájl típusa (pl. pdf, doc, jpg), a platform mégis biztosít lehetőséget annak a „kézi” beállítására, amelyet az Intent későbbi helyes kezelésének érdekében ajánlott is kihasználni.

Az adatmező beállítására a `setData()`, a `setType()` és a `setDataAndType()`, míg lekérdezésére a `getData()` és a `getType()` metódusok állnak rendelkezésre.

Komponensnév

Annak a komponensnek a neve, amelynek kezelnie kell az Intentet, az üzenet explicit címzettje. Típusát tekintve *ComponentName* objektum, ez egyrészt a célkomponens minősített osztálynevéből (*fully qualified class name*, például „*teszt.projekt.app.SzovegSzerkesztoActivity*”) és annak az alkalmazásnak a csomagnevéből (*package name*, például „*teszt.projekt*”) áll, amelyben a komponens megtalálható. Szükség van a csomagnév megadására, hiszen az nem feltétlenül egyezik meg a minősített osztálynév elejével.

¹⁴ Hivatalos MIME-típusok: <http://www.iana.org/assignments/media-types/index.html>

A komponens nevének megadása nem kötelező, ha üresen hagyva küldjük az Intentet a rendszernek, akkor annak többi attribútuma alapján az Android megkeresi számára a legjobb célpontot. Erről az Intent-feloldásnak (*Intent resolution*) nevezett mechanizmusról lásd bővebben a későbbiekben lesz szó.

A mező beállítására szolgáló metódusok a *setComponent()*, a *setClass()* és a *setClassName()*, míg olvasni a *getComponent()* segítségével lehet.

Kategória

További igényeket fogalmazhatunk meg az Intentet kiszolgáló komponenssel kapcsolatban, amelyeket az Android figyelembe vesz a célpont kiválasztásakor. Míg az eddig bemutatott attribútumok legfeljebb egy bejegyzést tartalmazhatnak, kategóriából többet is megadhatunk, amelyeket egyszerre kell teljesítenie a kiszolgáló komponensnek. Az egyes Android-verziókban bevezetett kategóriák teljes listája szintén az Intent dokumentációjában található, az alábbi táblázat csak a gyakran használtakat mutatja be.

4.2. táblázat. Gyakran használt Intent-kategóriák

Kategória	Leírás
CATEGORY_BROWSABLE	A webböngészőből biztonságosan elindítható Activityknek támogatniuk kell ezt a kategóriát. A gyári Android-böngésző (és a Marketben elérhetők is) beállítják ezt minden Intent küldésekor. Ez történik például az e-mail címre kattintás hatására, de így oldja meg a Market-alkalmazás is a <i>http://market.android.com</i> kezdetű webcímek megnyitását.
CATEGORY_LAUNCHER	Az Intentet csak olyan Activity szolgálhatja ki, amely megjelenik az alkalmazásmenüben (<i>Launcher</i>). Ezt közvetlenül nem használjuk valamilyen akció elvégzésére irányuló Intentek esetén, hanem a későbbiekben részletesen bemutatott szűrők beállításakor így jelezzük, hogy melyik Activityhez legyen ikon a Launcherben.
CATEGORY_HOME	Az Activity képes arra, hogy az operációs rendszer nyitóképernyője legyen. Szintén nem elküldött Intent esetében, hanem szűrőként beállítva használjuk saját <i>Home Screen</i> alkalmazásunkban.
CATEGORY_DEFAULT	Intent-szűrőben állítható be, jelzi az Activity képességét arra, hogy a beállított adat és akció esetében alapértelmezettként kiválasztható legyen.

Extrák

Gyakran szükség lehet olyan plusz információk elhelyezésére az Intentben, amelyek az eddig tárgyalt attribútumokba nem illenek. Ilyen lehet például a hibaüzenet szövege, amelyet egy elképzelt alkalmazásrendelés feladási képernyője átküld az általános hibamegjelenítőnek, de maga az Android-runtime is kihasználja ezt a lehetőséget, például, amikor a névjegyzékből kiválasztott kontaktnak elektronikus levelet szeretnénk írni, és az e-mail szerkesztő felületnek átadja a címzettet. Ez az információ nem kerülhet az Intent adatrészébe, hiszen nem az e-mail címen kell elvégezni az „új levél írása” műveletet. Az *Extrák* attribútum az Intent *Kategóriák* mezőjéhez hasonlóan több elemet tartalmazhat. Ez a tagváltozó nem más, mint egy *Bundle* objektum, amelyben tetszőleges mennyiségű kulcs-érték párt helyezhetünk el. A kulcs olyan *String*, amely nevéből adódóan kötelezően egyedi, az érték pedig csak olyan típusú változó lehet, amely megvalósítja a *Parcelable* vagy a *Serializable* interfészt. A platform közvetlenül használható beépített típusait a következő táblázat foglalja össze, de a *Bundle*-ben elhelyezhetjük saját osztályaink példányait is, ha azok implementálják valamelyik interfészt.

4.3. táblázat. Intent-extraként közvetlenül használható típusok

int	char	long	ArrayList<String>
double	String	byte	ArrayList<Integer>
float	CharSequence	Bundle	ArrayList<CharSequence>

Az operációs rendszer és az előre telepített alkalmazások által küldött extrák mindig specifikusak az Intent céljának megfelelően, pontos listájuk megtalálható az osztály API-referenciájában. Ha ilyen üzenetsomag kezelésére szeretnénk felkészíteni az alkalmazásunkat, itt kell utánanézni, hogy pontosan milyen adatok csomagolódnak be az Intentbe, és hogyan kell értelmezni, helyesen felhasználni őket.

Az extrákat tartalmazó *Bundle* egyes elemeinek írására és olvasására az adat típusának megfelelő *putExtra(String kulcs, Típus érték)* és *getTípusExtra(String kulcs)* metódusok használhatók, ahol a „Típus” a fenti táblázat valamelyik mezőjének értéke, például egész típusú érték esetén *putExtra(„UserID”, 23)* és *getIntExtra(„UserID”)*, az *ArrayList* extrák olvasására pedig a *getTípusArrayListExtra(String kulcs)* metódussalád szolgál. Az extrákat tartalmazó teljes *Bundle* példány a *getExtras()* és a *putExtras()* függvényekkel kérhető le és állítható be.

Flagek

Az Intent leggyakoribb használati esete az, amikor egy alkalmazás egyik képernyőjéről a másikra váltunk a *startActivity(Intent)* metódussal. Az így indított Activity alapértelmezetten a *Back Stack* tetejére jön létre, akármilyen

van éppen a veremben. Ha ez a mechanizmus nem felel meg, az Intent *Flags* részében írhatjuk elő az újonnan indított komponens elhelyezését a stacken, illetve kontrollálhatjuk az Intent rendszer általi kezelését. Ritkán van szükség az alapértelmezett működés változtatására, ekkor kellő körültekintéssel, valamint minden lehetséges eset korrekt tesztelésével kell ezt végeznünk, hiszen az alkalmazás viselkedése eltérhet attól, amit a felhasználó elvár. A használható flagek mindenkor aktuális listája a platform dokumentációjában az Intent osztály *setFlags()* metódusának leírásánál található, a fontosabbakat a következő táblázat mutatja be. A konstans nevében található *ACTIVITY* vagy *RECEIVER* részlet jelzi, hogy milyen típusú alkalmazáskomponensnek küldött Intent esetén használhatók az egyes flagek.

Flag	Leírás
FLAG_ACTIVITY_NEW_TASK	Az így indított Activity nem a hívó komponens taszkjában indul, hanem az Android indít újat. Az új Task vermében az ekkor induló Activity lesz az első. Általában akkor használjuk, ha nyitóképernyő-jellegű alkalmazást készítünk, amely olyan alternatívákat kínál a felhasználó számára, amelyek az indító Activitytól teljesen függetlenül kell, hogy fussanak. Ha az indítandó Activityhez tartozó taszk már létezik, akkor a rendszer nem készít újat, hanem előtérbe helyezi a meglévőt.
FLAG_ACTIVITY_CLEAR_TASK	Ha az ezzel a flaggel indítandó Activity már fut valamelyik taszkban, az operációs rendszer ki-dob mindent a taszk verméből, majd elindítja az Activityt.
FLAG_ACTIVITY_CLEAR_TOP	Ha az így indított Activity már szerepel a hívó komponens taszkjának a vermében, az Android eltakarít minden fölötte lévő elemet, és az így már legfelül lévő Activity példányának továbbítja az Intentet, amely ennek hatására folytatódik.
FLAG_ACTIVITY_NO_ANIMATION	Az így indított Activity az Android alapértelmezett képernyőváltó animációja nélkül indul el, és előtérbe kerül.
FLAG_RECEIVER_REPLACE_PENDING	Ezzel a flaggel ellátva egy <i>Broadcast Intent</i> et minden ugyanilyen <i>PendingIntent</i> et (lásd később) töröl a rendszer.

Az Intent tehát információkat tartalmaz egyrészt a kiszolgálója számára az akció-, az adat- és az extrák mezőkkel, másrészt igényeket közvetít az Android számára a megfelelő célpont kiválasztásával kapcsolatban a komponensnév, a kategória és a flagek segítségével.

4.3. Activity indítása

Az Intentet komponensek közti kötés megvalósítására használjuk, leggyakrabban az alkalmazás következő képernyőjére való ugrásra, azaz új Activity (vagy Service) indítására. Ezt a feladatot a `startActivity()` metódus látja el, amelynek hatására a rendszer elindítja vagy folytatja a paraméterként átadott Intent által meghatározott Activityt.

```
Intent intent = new Intent();
// intent tagváltozóinak beállítása

startActivity(intent);
```

A `startActivity()` metódus hatására a rendszer megkeresi a kapott Intent beállításainak leginkább megfelelő Activityt, és elindítja. Ha ezzel a módszerrel indítunk új komponenst, a hívó oldalon semmilyen értesítést vagy visszajelzést nem kapunk annak befejeződéséről. Hogyha szeretnénk információhoz jutni az indított Activity lefutásának az eredményéről, a `startActivityForResult()` metódust kell használnunk (lásd később).

A betöltendő komponens megadására két lehetőségünk van:

- *Explicit Intent*: Kitöltjük az Intent *Komponensnév* mezőjét. Ekkor a rendszer a *ComponentName* objektum osztálynév és csomagnév attribútumai alapján megkeresi az Activityt implementáló osztályt és az alkalmazást, amelyben szerepel, majd új példányt indít, vagy a memóriában lévő folytatja. Az Intent *Flags* mezőjének beállításával szükség szerint finomhangolható a folyamat utolsó lépése.
- *Implicit Intent*: Az Intent *Akció* mezőjét töltjük ki. Ezzel nem azt mondjuk meg az Androidnak, hogy pontosan melyik Activity induljon el (mint az előző esetben), hanem az elvárt eseményt írjuk le, és az operációs rendszerre bízunk a megfelelő komponens kiválasztását. Bizonyos akciók önmagukban is értelmezhetők (például fénykép, videó készítése), többnyire azonban az *Akción* kívül az *Adat* mező kitöltése is szükséges (például fénykép szerkesztése, zene lejátszása).

Mindkét esetben lehetőségünk van az indítandó komponenssel kapcsolatban további kritériumok megadására, az Intent *Kategória* mezőjének kitöltésével.

4.3.1. Explicit Intent

Ha az alkalmazásunkon belül szeretnénk a következő Activityre lépni, a lekézenfekvőbb megoldás az explicit Intent használata. Ekkor ugyanis ismerjük az újonnan indítandó Activity komponensnevét, és az operációs rendszernek nem szükséges a készülékre telepített összes Activityt megvizsgálnia a legjobb célpont kiválasztásához. Ekkor legegyszerűbb az Intent konstruktorában átadni az alkalmazáskontextust és az indítandó Activity osztálynevét, majd meghívni a `startActivity()` metódust.

```
Intent intent = new Intent(  
    getApplicationContext(), // ez csak Activity-ből hív-  
    ható!  
    TermekListaActivity.class // amit indítani akarunk  
);  
  
startActivity(intent);
```

A `startActivity` metódussal akár más alkalmazásban implementált Activityt is indíthatunk, ha a csomagkontextus-attribútumot nem a `getApplicationContext()`-tel állítjuk be, hanem a külső alkalmazás tényleges csomagnevét adjuk meg.

4.3.2. Implicit Intent

Abban az esetben, ha nem tudjuk vagy nem akarjuk pontosan meghatározni az indítandó komponenst, a `startActivity`-nek átadott Intent-objektumban nem kell kitöltenünk a *Komponensnév* mezőt. Helyette azt kell specifikálnunk, hogy milyen akciót szeretnénk végrehajtani, majd az operációs rendszer futásidőben megkeresi a kérésünk kiszolgálására legalkalmasabb Activityt. Ezzel a deklaratív jellegű célpontmegadással képesek vagyunk más alkalmazások képességeit igénybe venni, ha azok fel lettek készítve implicit Intentek kiszolgálására. Az Android platform egyik legnagyobb erőssége rejlik ebben a funkcióban, ugyanis leveszi a fejlesztők válláról azoknak a feladatoknak a megoldását, amelyeket egy másik alkalmazás már implementált.

Tipikusan olyan műveletek esetén használunk implicit Intentet, amelyeket valamelyik, Androidra előre telepített szoftver képes megoldani. Ilyen például a névjegy kiválasztása vagy szerkesztése, fénykép vagy videó készítése, zene lejátszása vagy weboldal megnyitása. Ám nem csak ezek az alkalmazások képesek szolgáltatásaik kiejánlására, bármelyik Android-szoftver felkészíthető arra, hogy máshonnan érkező implicit Intenteket szolgáljon ki. Így építhetjük be például PDF-dokumentumok olvasását az alkalmazásunkba két sor (!) kód segítségével. Ezeket a funkciókat implementálhatnánk ugyan saját magunk is, ám a készülékre telepített programok képesek elvé-

gezni a feladatot helyettünk, így esetenként komoly mennyiségű fejlesztési időt takaríthatunk meg. Az Android-fejlesztés egyik alapvető jótanácsa az előbbieket fényében a következő:

Amennyiben olyan funkcióra van szükségünk, amelyet egy másik alkalmazás valószínűleg képes nyújtani, akkor implementáció előtt tájékozódjunk erről, és ha lehetséges, használjunk *Intent*-et.

A tanács betartásához folyamatosan követnünk kell a platformra megjelenő alkalmazásokat, és tisztában kell lennünk azok képességeivel, ez mindennapos feladata egy felkészült Android-fejlesztőnek.

Implicit Intent használatakor tehát nem kell kitöltenünk a *Komponensnév* mezőt, ehelyett viszont mindenképpen meg kell adnunk az elvárt *Akció*-t. Ha ez önmagában nem értelmezhető (legtöbbször így van), az *Intent* *Adat* mezőjében adhatjuk meg az adatra mutató *Uri*-t. Példa adat nélkül indítható *implicit Intentre*:

```
Intent makePhoto = new Intent(
    MediaStore.ACTION_IMAGE_CAPTURE // fénykép készítése
);
startActivityForResult(makePhoto, 1);
```

Példa csak adattal értelmezhető *implicit Intentre*:

```
Intent phoneCall = new Intent(
    Intent.ACTION_CALL, // "hívás indítása" akció
    Uri.parse("tel:+36301234567") // telefonszám mint Uri
);
startActivity(phoneCall);
```

Ha az *Akció* mellé adatot is meg kell adnunk, ezt tehetjük *Uri*-ként vagy az *Intent* Extra *bundle* feltöltésével. *Implicit Intent* használatakor mindig az adott *Intent* dokumentációjában kell utánajárnunk, hogy milyen adatokat és hogyan kell átadnunk ahhoz, hogy a kiszolgálása biztosított legyen.

Előfordulhat olyan eset, hogy az általunk küldött *implicit Intent*-et egynél több vagy kevesebb komponens is képes kiszolgálni. Több lehetséges célpont esetén a választás a felhasználóra van bízva, ezt a kódból nem tudjuk befolyásolni. Ha a rendszer nem talál egyetlen célpontot sem, akkor a *startActivity* hívás *ActivityNotFoundException* típusú kivételt dob, amelyet a hívóoldalon kell elkapnunk és feldolgoznunk. Például PDF-fájl helyes megnyitása hibakezeléssel a következő:

```

// referencia a nyilvános tárhely gyökerében
// lévő "manual.pdf" fájlra
File pdfFile = new File(
    getExternalFilesDir(null) + "/manual.pdf"
);

// amennyiben létezik
if(pdfFile.isFile()){

    // Intent létrehozása, akció: megnyitás olvasásra
    Intent i = new Intent(Intent.ACTION_VIEW);

    // MIME típus meghatározása beépített metódussal
    // kiterjesztés alapján
    String pdfMimeType = MimeTypeMap.getSingleton().
        getMimeTypeFromExtension("pdf");

    // Intent adat és típus beállítása
    i.setDataAndType(Uri.fromFile(pdfFile),
        pdfMimeType);

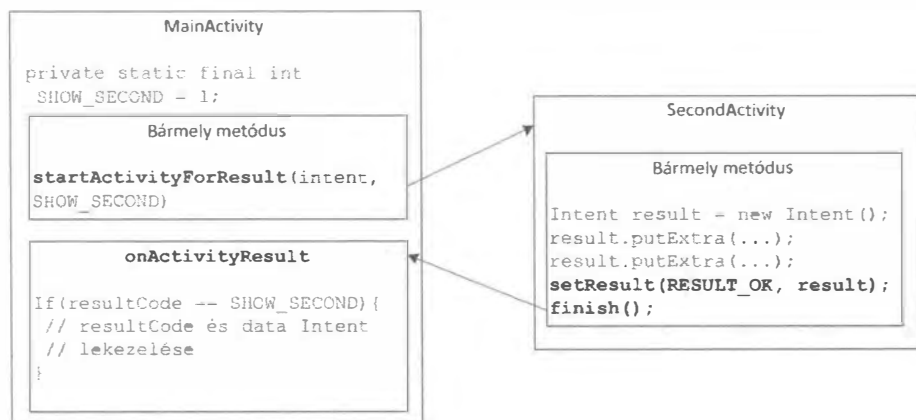
    try{
        // megpróbáljuk megnyitni a fájlt
        startActivity(i);
    }
    catch (ActivityNotFoundException e) {
        // amennyiben nincs az Intent kiszolgálására
        // alkalmas komponens, értesítjük a felhasználót
        Toast.makeText(
            getApplicationContext(),
            "A készülékre nincs telepítve
            PDF olvasó!",
            Toast.LENGTH_LONG).show();
    }
}

```

4.4. Activity visszatérési értéke

A *startActivity* metódussal történő indításkor a hívóoldalon semmilyen visszajelzést nem kapunk arról, hogy mi történt a hívott Activityben, hogyan fejeződött be, de még arról sem kapunk értesítést, hogy egyáltalán befejeződött. Ha szeretnénk kezelni egy Activity „visszatérését”, a *startActivityForResult* metódussal kell indítanunk. Ekkor az új komponens al-Activityként jön létre a veremben, és befejeződésekor a hívóoldalon lefut egy olyan eseménykezelő, amelyben lekezelhetjük a különböző befejeződési módokat, és feldolgozhatjuk

a visszaadott adatokat. Ez a módszer akkor a leghasznosabb, ha egy Activityt azért indítunk, hogy adatot szolgáltatson a hívónak (például felhasználónév és jelszó bekérése, névjegy kiválasztása, fényképezés). Ezek az al-Activityk semmiben nem különböznek hagyományos társaiktól, kivéve azt, hogy befejeződésük előidézi az *onActivityResult* eseménykezelő lefutását. A teljes folyamatot az alábbi ábra szemlélteti.



4.6. ábra. Activity visszatérésének lekezelése

Használatát a következő kódrészletek mutatják be.

```
// MainActivity.java

private static final int SHOW_SECONDACTIVITY = 1;

Intent intent = new Intent(
getApplicationContext(),
SecondActivity.class
);

startActivityForResult(intent, SHOW_SECONDACTIVITY);
```

A *startActivityForResult* második paramétere a *requestCode* nevű *int* azonosító, amely az Activityn belül egyedi. Minden Activityből korlátlan számú különböző al-Activityt indíthatunk, amelyek befejeződésükkor ugyanazt az eseménykezelőt triggerelik. Ez a paraméter különbözteti meg az egyes hívásokat, az eseménykezelő törzsében így tudjuk meghatározni, hogy honnan tért vissza a vezérlés.

A visszatérésiérték természetesen implicit *Intent* használatakor is fontos adatot tartalmazhat, ebben az esetben is használhatjuk a *startActivityForResult()* metódust. Például névjegy kiválasztására indított Activity a következő:

```
private static final int PICK_CONTACT = 2;

Intent intent = new Intent(
    Intent.ACTION_PICK,
    Contacts.CONTENT_URI
);

startActivityForResult(intent, PICK_CONTACT);
```

A hívott Activityben a befejeződést előidéző *finish()* hívás előtt a *setResult* metódus segítségével tudjuk beállítani azokat az értékeket, amelyeket a hívó visszakap. Itt lehetőségünk van megadni egy *int* visszatérési kódot, valamint opcionálisan egy egész Intent-objektumot.

A visszatérési kódra az Android biztosít számunkra néhány előre definiált konstanst, amelyek közül leggyakrabban a *RESULT_OK* és a *RESULT_CANCELLED* használatos. Ez utóbbi beállítása történik meg alapértelmezetten, tehát ha a kódból nem állítunk be semmit, vagy az Activity abnormalis módon fejeződött be, akkor a hívó ezt a kódot kapja meg visszatérésként.

A visszaadható Intent *Extrák* mezőjét tetszőleges számú és típusú adattal tölthetjük fel (egyedi típusra itt is érvényes, hogy implementálnia kell a *Parcelable* vagy a *Serializable* interfészt). Igény esetén használhatjuk az Intent további mezőit is, például az *Adat* mező kitöltésével Uri-t adhatunk vissza, ha az Activity olyan fájllistát jelenített meg, amelyből a felhasználó egyet választott.

```
// SecondActivity.java

pickContactBtn.setOnClickListener(new
OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent resultIntent = new Intent();
        resultIntent.setData(uriOfSelectedFile);
        setResult(RESULT_OK, resultIntent);
        finish();
    }
});

cancelBtn.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        setResult(RESULT_CANCELED);
        finish();
    }
});
```

A *finish()* hívás hatására az Activity befejeződik, törlődik a *Back Stack* tetejéről, és a vezérlés visszakerül a hívóhoz. Itt az *onActivityResult()* eseménykezelő hívódik, amelynek az Android paraméterként átadja az al-Activity azonosítására szolgáló *requestCode*-ot, valamint a visszatérési értéként *setResult* metódussal beállított *resultCode*-ot és adatot tartalmazó Intentet. Ha nem lett beállítva semmi a hívott Activityben, a *resultCode* értéke *RESULT_CANCELLED*, az Intent pedig *null*.

```
@Override
protected void onActivityResult(
    int requestCode, int resultCode, Intent
    data) {

    switch (requestCode) {
        case SHOW_SECOND:
            switch (resultCode) {
                case RESULT_OK:
                    Uri selectedFile = data.getData();
                    // kiválasztott fájl kezelése
                    break;
                case RESULT_CANCELED:
                    // a felhasználó nem választott semmit
                    break;
            }
        break;
    }
}
```

4.5. Intent-szűrők

Az Intent általánosan egy kérés egy akció elvégzésére valamilyen adaton. Ha implicit Intentet indítunk, azaz nem nevezzük meg a kérés kiszolgálására hivatott alkalmazáskomponenst, akkor az operációs rendszer feladata a legjobb jelölt kiválasztása. Hogyan végzi ezt az Android?

Az Activityk, a Service-ek és a BroadcastReceiverek úgynevezett Intent-szűrők (*Intent Filter*) segítségével tudják regisztrálni a képességeiket arra, hogy valamilyen akciót teljesítsenek egy bizonyos típusú adaton. Ugyancsak így történik a BroadcastReceiver feliratkozása a rendszerszintű eseményekre. Ezzel a komponens kinyilvánítja, hogy milyen kéréseket tud kiszolgálni az eszközre telepített bármelyik alkalmazás számára.

Az Intent-szűrők az *AndroidManifest.xml* leíróban jelennek meg *<intent-filter>* node-ként annak a komponensnek az XML-elemén belül, amely ki tudja szolgálni az adott kérést. Egy komponenshez akár több szűrő is tartozhat, mindegyik

egy-egy különböző képesség meghatározására szolgál. Az `<intent-filter>` elem kötelező és opcionális gyermekelemeket tartalmazhat, mindegyikből akár többet is.

1. **action:** Kötelező mező, minden szűrőben legalább egy ilyen node-nak kell szerepelnie. Azt az akciót határozza meg, amelyet a komponens képes kiszolgálni. Lehetséges értékei az Android platform beépített akciói (lásd Intent *Akció*) vagy globálisan egyedi *String*, utóbbi megadására legjobb gyakorlat a Java csomagnév-elnevezési konvencióinak a használata. Ha minden akciót le akarunk kezelni, üres `<action/>` taget kell felvennünk a szűrőhöz.
2. **category:** Opcionális mező, amely azt írja le, hogy a komponens milyen körülmények között képes kiszolgálni a bejövő implicit Intentet. Az Intent *Kategória* mezőjének értékeivel tölthető ki.
3. **data:** Opcionális mező, amely azt határozza meg, hogy az akció milyen adaton végezhető el. Az alábbi attribútumok tetszőleges kombinációjával adható meg:
 - **android:scheme** – Uri-séma megadása, például *http* (*http://*-rel kezdődő Uri), *content* (*content://*-rel kezdődő Uri, Content Provider-lekérés)
 - **android:host** – Uri-hosztnév, például *google.com*, *aut.bme.hu*
 - **android:port** – A hoszthoz tartozó érvényes port, például 80, 21
 - **android:path** – Uri elérési út a hoszton belül, például *„/media/images”*, *„/targyak”*
 - **android:mimeType** – Az adat MIME-típusa, például *application/pdf*

Ha írunk egy PDF-olvasó Activityt, és szeretnénk más alkalmazások számára is elérhetővé tenni ezt a funkciót, a következő szűrőt kell beállítanunk az *AndroidManifest.xml*-ben:

```
<activity android:name=".PDFViewerActivity"
  android:label="Display PDF">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"></action>
    <category android:name="android.intent.category.DEFAULT"/>
    <category
      android:name="android.intent.category.SELECTED_ALTERNATIVE"
    />
    <data android:mimeType="application/pdf"/>
  </intent-filter>
</activity>
```

A két kategóriabeállítás azt teszi lehetővé, hogy az alkalmazásunk megjelenjen a PDF-fájlok megnyitására használatos applikációk között, és a felhasználó kiválaszthassa alapértelmezettként.

4.5.1. Intent-feloldás

Az Intent-szűrőkkel azt állítottuk be, hogy az alkalmazásunk egyes komponensei milyen implicit Intentek kiszolgálására képesek. Ha az eszközre telepített bármelyik alkalmazás bármelyik komponense ilyen Intentet ad paraméterül a *startActivity* metódusnak, azt az Android dolgozza fel, és keresi meg a kiszolgálására leginkább alkalmas Activityt. Ezt a folyamatot nevezzük Intent-feloldásnak (*Intent Resolution*), amely a következő lépésekből áll.

1. A futtató környezet (*runtime*) összegyűjt minden Intent-szűrőt az összes telepített alkalmazásból.
2. Összehasonlítja a szűrőket a bejövő implicit Intent *Akció* és *Kategória* mezőjével.
 - a. Az akcióalapú ellenőrzés akkor sikeres, ha az Intent akciója megegyezik a szűrő valamelyik akciójával, vagy a szűrőben üres akció van beállítva.
 - b. A kategóriavizsgálat ennél szigorúbb: ha az Intent tartalmaz kategóriamegkötéseket, akkor a szűrőben az összes kategóriának szerepelnie kell ahhoz, hogy kiszolgálhassa a kérést.
3. A rendszer törli a listából azokat a szűrőket, amelyek nem feleltek meg valamelyik teszten.
4. Az adatalapú ellenőrzés az Intent Uri és MIME-típus beállítását veti össze a szűrők *data* beállításával. Bármely Uri-részlet vagy típusbeli eltérés törli a szűrőt a listából. Ha a szűrőben nincs *data* mező, az adatellenőrzés mindig sikeres lesz, bármilyen adatot képes befogadni a komponens.
5. Előálltak a kérés kiszolgálására alkalmas Intent-szűrők:
 - Ha egynél több alkalmazáshoz tartoznak, a rendszer listázza őket a felhasználónak, aki kiválaszthatja, hogy melyik induljon.
 - Ha nincs olyan komponens, amely képes lenne kiszolgálni a kérést, akkor *ActivityNotFoundException* generálódik a *startActivity* hívójánál.
 - Egy komponens esetén a rendszer ezt kérdés nélkül elindítja, és továbbítja az implicit Intent-objektumot.

4.6. További Intent-lehetőségek

Az Intent-mechanizmus leggyakrabban használt eseteit megvizsgáltuk, ám ennél még többre is képes. Ebben a részben megnézzük hogy milyen további lehetőségeink vannak az Intentek használatával kapcsolatban.

4.6.1. *PendingIntent*

Az Intent-mechanizmust eddig arra használtuk, hogy összeállítsuk a kérést, és ugyanebből a komponensből azonnal új Activityt indítsunk. A *PendingIntent* osztály segítségével lehetőségünk van olyan Intent definiálására, amelyet nem akarunk rögtön elindítani, és akár nem is mi küldjük el a rendszernek.

Tipikusan widget és értesítés (*Notification*) esetén használjuk ezt az osztályt, ahol előre be kell állítanunk, hogy milyen Intent süljön el a widget valamelyik UI-elemére történő kattintás vagy a *Notification* kiválasztása esetén.

```
// Activity indítása
Intent startActivityIntent = new Intent(
    getApplicationContext(),
    SecondActivity.class);
PendingIntent activityPI = PendingIntent.getActivity(
    getApplicationContext(), // kontextus
    0, // requestCode
    startActivityIntent, // Intent
    0); // flag-ek

// Broadcast esemény
// Intnet egyedi akcióval
Intent broadcastIntent = new Intent(
    "teszt.projekt.app.UJ_JEGYZET_KELETKEZETT");

PendingIntent broadcastPI = PendingIntent.
getBroadcast(
    getApplicationContext(), // kontextus
    0, // requestCode
    broadcastIntent, // Intent
    0); // flag-ek
```

4.6.2. Linkify

A *Linkify* segédosztálynak az a feladata, hogy *TextView* (és leszármazottai) tartalmát elemezve kattintható hivatkozásokat készítsen a szövegbe ágyazott linkekből. Az osztály reguláris kifejezések alapján képes meghatározni a linkeket, ezekre kattintva a következő kódrészletet futtatja:

```
startActivity(new Intent(Intent.ACTION_VIEW), uri);
```

A *Linkify* tehát implicit Intentet generál, amely megnyitásakciót hív a reguláris kifejezésre illeszkedő szövegrészletre.

Az egyedi mintákat kódból adhatjuk meg, de az osztály alapból tartalmazza a gyakori kifejezéstípusokat, amelyeket a kódból vagy a *TextView* XML-attribútumaként adhatunk meg legegyszerűbben. A beépített típusok a következők:

- ***Linkify.WEB_URLS***: böngésző által megnyitható webes hivatkozások (http://-rel kezdődő érvényes URL-ek). XML-beli megadása: „*web*”
- ***Linkify.EMAIL_ADDRESSES***: e-mail címek válnak kattinthatóvá; XML-beli megadása: „*email*”
- ***Linkify.MAP_ADDRESSES***: térképen megjeleníthető címek (~postacímek). XML-beli megadása: „*map*”
- ***Linkify.PHONE_NUMBERS***: telefonszámok XML-beli megadása: „*phone*”
- ***Linkify.ALL***: a fenti mintákra illeszkedő összes szövegrész kattinthatóvá válik, XML-beli megadása: „*all*”

Beállítása a layout-XML-fájlban történik, akár több is megadható egy *TextView*-hoz:

```
<TextView
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:text="@string/szoveg_hivatkozasokkal"
    android:autoLink="web|email|phone"
/>
```

4.6.3. Intent lekérése, delegálása, a feloldás előzetes eredménye

Ha olyan Intent-szűrőt állítottunk be az egyik komponensünkhöz, amely több *action* bejegyzést is tartalmaz, akkor a komponensnek tudnia kell, hogy a bejövő implicit Intent küldője melyik akciót kérte. Ennek meghatározására szolgál a *getIntent()* metódus, amely futásidőben visszaadja a teljes Intent-objektumot, és ennek hatására a komponensünk elindult. A lekért Intent egyes mezőinek olvasásához a *getAction()*, a *getData()*, a *getCategory()*... nevű metódusok használhatók.

```
@Override
public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Intent intent = getIntent();

    String action = intent.getAction();
    Uri data = intent.getData();
    ...
}
```

Ha az implicit Intent eljutott a mi komponensünkhöz, de mégsem szeretnénk lekezelni, a *startNextMatchingActivity(intent)* metódussal delegálhatjuk a feloldozását az Intent-feloldás során talált következő alkalmas komponensnek.

```
@Override
public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Intent intent = getIntent();
    if (nemSzolgalomKi)
        startNextMatchingActivity(intent);
}
```

Bizonyos esetekben jól jöhet, ha az implicit Intent elsütése előtt már tudjuk, hogy lesz-e olyan komponens, amelyik képes kiszolgálni a kérésünket. Ha tudjuk, hogy nincs telepítve erre alkalmas applikáció, akkor megspórolhatunk egy kivételelkapást, és a felhasználó elől elrejtethetjük azokat az opciókat, gombokat és menüelemeket, amelyek mögött nem tudjuk biztosítani a funkcionálisitást. Az alábbi rövid metódus ellenőrzi, hogy a kapott akciót képes-e bárki végrehajtani az eszközre telepített alkalmazások közül.

```

public static boolean isIntentAvailable(Context
context, String action) {
    final PackageManager packageManager =
context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
        packageManager.
queryIntentActivities(intent,
                        PackageManager.MATCH_DEFAULT_
ONLY);
    return list.size() > 0;
}

```

A vonalkód és a QR-kód olvasására irányuló, csak a *Barcode Scanner* alkalmazás által lekezelhető implicit Intent kiszolgálhatóságának vizsgálata a metódus segítségével a következő:

```

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    final boolean scanAvailable =
isIntentAvailable(this,
        "com.google.zxing.client.android.SCAN");

    MenuItem item;
    item = menu.findItem(R.id.menu_item_add);
    // ha nincs telepítve a Barcode Scanner,
    // akkor a menü elem letiltva jelenik meg
    item.setEnabled(scanAvailable);

    return super.onPrepareOptionsMenu(menu);
}

```

4.6.4. Google-alkalmazások implicit Intentjei

Az Android platform minden verziója számos előre telepített alkalmazással érkezik, amelyek jelenlétére és funkcionalitására számíthatunk fejlesztőként, és használhatjuk őket a saját alkalmazásunkból indított implicit Intentek lekezelésére. Az Android fejlesztői portálon a legtöbb ilyen Intent dokumentálva szerepel, érdemes a listát minden platformverzió kiadásakor újra tanulmányozni. Elérhetősége a következő: <http://developer.android.com/guide/appendix/g-app-intents.html>.

4.7. Rendszerszintű események

Eddig az Intent-mechanizmust arra használtuk, hogy alkalmazáskomponenseket indítsunk, ám mivel képesek átlépni a processzhatárokat, és strukturált adatot foglalhatnak magukban, az Intenek segítségével rendszerszintű (*broadcast*) üzeneteket is küldhetünk az arra feliratkozott BroadcastReceiver objektumainak. Így az alkalmazásunkban végbemenő eseményekről értesíthetjük a készülékre telepített applikációkat, eseményvezérelt programozási modellt valósítva meg. Az Android futtató környezet ugyanezt a mechanizmust használja a rendszerben keletkezett események, üzenetek továbbítására, így akár ezekre is reagálhatunk, ha szeretnénk. Jó gyakorlat például feliratkozni az „adatkapcsolat megszűnése” eseményre, és bekövetkezése esetén leállítani a webszerver kommunikációs kísérletét, így spórolva az energiahasználattal.

4.7.1. Broadcast esemény generálása

Egy Intentet *broadcast*ként küldeni majdnem ugyanúgy kell, ahogyan új Activityt indíthatunk a segítségével. Az *action*, a *data*, a *category* és az *extras* mezők (opcionális) feltöltése után a *sendBroadcast(intent)* metódushívás hatására az Android futtató környezet gondoskodik az eseményleíró Intent célbajuttatásáról minden érdekelt BroadcastReceiver számára.

```
Intent i = new Intent();
i.setAction("teszt.projekt.app.UJ_JEGYZET");
i.putExtra("name", "Jegyzet címe");
i.putExtra("body", "Jegyzet szövege");
sendBroadcast(i);
```

4.7.2. Feliratkozás *broadcast* eseményre

Android platformon a BroadcastReceiver az egyetlen alkalmazáskomponens, amely képes lefutni *broadcast* esemény bekövetkezésekor.

Saját *Receiver* osztály létrehozásához új osztályt kell írunk, és le kell származtatnunk az eseményt az *android.content.BroadcastReceiver* ősosztályból, majd implementálnunk kell az *onReceive()* metódusát.

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class UjJegyzetReceiver extends
BroadcastReceiver {
```

```

@Override
public void onReceive(Context context, Intent
intent) {
    // bejövő broadcast Intent kezelése
    String jegyzetNev = intent.
getStringExtra("name");
    String jegyzetTartalom =
        intent.getStringExtra("body");
}
}

```

Egy BroadcastReceiver-osztályt több különböző üzenet fogadására is beregisztrálhatunk, ekkor ugyanaz az *onReceive* metódus fut le bármelyik esemény bekövetkezésekor. Az itt futó kódnak öt másodpercen belül be kell fejeződnie, ellenkező esetben a rendszer ugyanúgy feldobja az „Application Not Responding” dialógusablakot, mintha a felhasználói felület szálában végeznénk blokkoló műveletet. Az *onReceive* törzsében végezhető műveletekkel kapcsolatban semmilyen korlátozást nem tartalmaz a rendszer, akár új szál, Activityt, Service-t is indíthatunk, ha szükséges.

4.7.3. BroadcastReceiver regisztrálása

A Receiver regisztrálása és érdekeltségeinek megadása Intent-szűrők megadásával történik, amelyeket az *AndroidManifest.xml*-ben vagy – kizárólag BroadcastReceiver esetén – futásidőben, kódból vehetünk fel. Utóbbi esetben a Receiver csak akkor fut le, ha a broadcast bekövetkezésének pillanatában az alkalmazás futó állapotban van, míg ha manifestben adjuk meg, a runtime képes elindítani az applikációt, és továbbítja a bejövő Intentet a Receiver-objektumnak.

A kódból történő regisztráció olyankor lehet hasznos, ha egy *broadcast* esemény hatására az alkalmazás felhasználói felületén szeretnénk változtatni valamit, például az internetkapcsolat megszakadásakor inaktívvá tesszük a szerverkommunikációt triggerelő gombokat, vezérlőket. Ilyenkor felesleges lenne a Receiver és az alkalmazás felébresztése vagy elindítása.

Az Intent-feloldás ugyanúgy történik, mint a nem broadcast implicit Intentek feldolgozásakor.

A regisztráció az *AndroidManifest.xml*-ből a következő:

```

<receiver android:name=".UjJegyzetReceiver">
  <intent-filter>
    <action android:name="teszt.projekt.app.UJ_JEGYZET"/>
  </intent-filter>
</receiver>

```


A regisztráció a kódból a következőképpen néz ki:

```
IntentFilter filter = new IntentFilter();
filter.setAction("teszt.projekt.app.UJ_JEGYZET");
UjJegyzetReceiver r = new UjJegyzetReceiver();
registerReceiver(r, filter);
```

4.7.4. Android *broadcast* eseményei

A platform sok natív *broadcast* Intentet generál, amelyekre feliratkozhatunk, és reagálhatunk igény szerint. A mindenkor teljes lista a hivatalos Android fejlesztői portálon található az Intent osztály dokumentációjában (<http://developer.android.com/reference/android/content/Intent.html>), a fontosabbakat az alábbi lista mutatja be.

- ***ACTION_BATTERY_LOW***: Az akkufeszültség alacsony, ekkor érdemes kikapcsolni vagy minimálisra csökkenteni a hálózati kommunikációt és a CPU-intenzív műveleteket.
- ***ACTION_POWER_CONNECTED***: A készülék hálózati töltőre lett kapcsolva. Ekkor érdemes a nagy terheléssel járó, elhalasztható műveleteket elvégezni (backup, teljes tartalomfrissítés, cache feltöltése stb.)
- ***ACTION_BOOT_COMPLETED***: A telefon bekapcsolása és az operációs rendszer felállása utáni esemény. Akkor érdemes feliratkozni rá, ha olyan Service-t készítünk, amelynek folyamatosan futnia kell anélkül is, hogy a felhasználónak kézzel el kéne indítania. Ekkor az *onReceive* metódusban indítjuk el a szolgáltatást.
- ***ACTION_NEW_OUTGOING_CALL***: Új kimenő hívás kezdődik. A hívásnapló alkalmazásnak fontos információ, az Intent *Extrá*ból kinyerhető a felhívott telefonszám.
- ***ACTION_INPUT_METHOD_CHANGED***: Megváltozott a szövegbeviteli mód. Tipikusan akkor történik, amikor a „kicsúsztható” fizikai billentyűzettel rendelkező telefonon a felhasználó váltott a virtuális és a valódi klaviatúra között. Érdemes ilyenkor eltüntetni vagy megjeleníteni a virtuális billentyűzetet.
- ***ACTION_MEDIA_MOUNTED***: Az eszközben lévő SD-kártya fel lett csatolva. Akkor következik be, amikor a felhasználó összeköti a telefont a számítógépével, és a PC látja a nyilvános tárhely tartalmát. Ekkor az Android-alkalmazások csak olvasni tudják a tárhelyet.
- ***ACTION_DATE_CHANGED***: A felhasználó átállította a dátumot.
- ***ACTION_TIME_CHANGED***: A felhasználó átállította a rendszeridőt.
- ***ACTION_TIME_TICK***: A rendszeridő változott (percenként sül el). Csak kódból regisztrált Receiverek kaphatják meg ezt az eseményt.

A perzisztens adattárolás eszközei

Az alkalmazások nagy részében a funkcionalitáshoz hozzátartozik, hogy az egyes futások között bizonyos adatok megmaradjanak. A memóriában tárolt objektumok ezt nem biztosítják, valamilyen állandó külső tárhoz kell fordulni segítségért. A számítástechnikában ezt ősidők óta a rendszer háttértáreleme vagy -elemei biztosítják, például egy merevlemez az asztali számítógépünkben.

Valószínűleg a programozás első lépései között tanultuk meg a fájlkezelést, amellyel a megőrzésre szánt byte-okat valamilyen általunk ismert struktúrában kiírtuk, illetve később visszaolvastuk. A korszerű osztálykönyvtárak a nyers adatfájl elérésénél magasabb szintű funkcionalitást is nyújtanak, hogy a programozónak ne kelljen a folyton ismételt alacsony szintű utasításokkal törődnie. A lokális adatkezelés jelenlegi egyik legösszetettebb módja a helyi relációs adatbázis használata. Jó hír azoknak, akik Android-programozására adják a fejüket, hogy a két véglet – nyers fájlkezelés és relációs adatbázis – és köztük néhány, adott célra kidolgozott programozói felület mind elérhető ezen a platformon. Ebben a fejezetben főként a fájlkezeléssel és az egyszerű adatmentéssel foglalkozunk, a következőben a relációs adatbázis-kezelést mutatjuk be.

5.1. Alacsonyszintű fájlkezelés

A Java nyelv mindennapi felhasználóiként joggal várhatjuk, hogy Android alatt a Java által biztosított és megszokott módon legyen lehetőségünk fájlok írására és olvasására. Ebben nem is kell csalódnunk, hiszen az itt vizsgált kódok jelentős része szabványos Java API-k használatát jelenti.

Egy koncepciót azonban meg kell ismernünk, mielőtt továbblépünk. Android alatt két „lemezterületet” különböztethetünk meg: az egyik az *Internal Storage*, vagyis a *belső tároló*, a másik – logikusan – az *External Storage*, azaz a *külső tároló*. Kezdő Android-programozók ezt gyakran úgy fordítják le maguknak, hogy van a beépített memória, illetve a cserélhető memóriakártya. Ez azonban nagy tévedés. A két koncepció valójában az adatok elérhetőségére vonatkozik. Az Internal Storage az a védett tárhely, amelyhez kizárólag a tulajdonos alkalmazás fér hozzá, más programok, felhasználók elviekben nem láthatják a tartalmukat. Az alkalmazásunk eltávolításával az operációs rendszer gondoskodik ezeknek az adatoknak a törléséről is. A külső tárolás ezzel szemben gyakorlatilag egy publikusan hozzáférhető fájlrendszer elérését jelenti, azaz egy mindenki által írható-olvasható területet. A készülék beépített memóriája is tartalmaz(hat) External Storage területet.

Fontos beszélnünk a fájlkezelés megismerésének legelején arról a tényről, hogy az *input/output* műveletek elvégzéséhez szükséges idő a memória sebessége, illetve a tipikus adatmennyiségek miatt nagyságrendekkel lassabb lehet az operatív memóriával végzett műveleteknél. Emiatt éles felhasználói alkalmazásoknál különösen figyeljünk arra, hogy ezek a műveletek ne a felhasználói felület megjelenítéséért is felelős fő szálon fussanak, hanem hozzunk létre külön szálat a részükre. Ez biztosítja, hogy alkalmazásunk reszponzív lesz, így jó felhasználói élményt nyújthat.

5.1.1. A privát tárterület használata

5.1.1.1. Fájlok írása és olvasása

A definíciójából adódóan ezt a tárolóhelyet privát módon szoktuk legtöbbször használni, bár opcionális beállítással a hozzáférés finomhangolható. Az íráshoz az *openFileOutput()* függvényt hívjuk egy vagy két paraméterrel: egyrészt a megnyitandó/létrehozandó fájl nevét adjuk meg, másrészt a létrehozandó állományhoz való hozzáférés módját. Ez utóbbi alapértelmezetten a *MODE_PRIVATE* konstanssal jelzett változat, amely mindenki elől elrejtja a fájlunkat (de használható még a *MODE_WORLD_READABLE* és a *MODE_WORLD_WRITEABLE* módosító is, amelyek a publikus olvasási és írási jogot jelzik, illetve a *MODE_APPEND*, amellyel hozzáfűzhetünk a meglévő állományhoz). A függvény egy *FileOutputStream*-et ad meg, amelyet ettől kezdve a szokásos módon használhatunk a *write()* függvénnyel. Használat után a *close()* függvénnyel zárhatjuk le a fájlunkat. Olvasáshoz analóg módon az *openFileInput()* függvényt használhatjuk, a visszaadott *FileInputStream*-et pedig a *read()* függvénnyel olvashatjuk.

```
String FILENAME = "storagefile.txt";
String string = "Titkos megorizando informaciok";
FileOutputStream out = null;
try {
    out = openFileOutput(FILENAME, Context.MODE_PRIVATE);
    out.write(string.getBytes());
} catch (Exception e) {
    Log.e("MyActivity", "Error! ", e)
}
finally {
    out.close();
}
```

Tipikus gyakorlat, hogy a privát fájloknak nem adunk kiterjesztést, hiszen a saját alkalmazásunkon kívül más nem fogja értelmezni.

Vajon meg lehet-e tudni, konkrétan hova íródik ki mindez az adat? A *getFilesDir()* egy *File* objektumban visszaadja a könyvtár elérési útját. Ha ezen belül alkönyvtárakat szeretnénk létrehozni, illetve elérni, a *getDir()* függvényt kell használnunk, amely visszaadja a paraméterül kapott elérési útvonalat a könyvtárunkon belül (illetve létrehozza, ha még nem létezik).

Fájlolvasáskor az *openFileInput()* függvényre van szükség, amely értelem-szerűen *FileInputStream*mel tér vissza (illetve *FileNotFoundException* kivételt dob, ha a fájl nem létezik).

5.1.1.2. Gyorsítótárazás

A belső tár egyik gyakori felhasználási területe az, amikor gyorsítótárazni (cache-elni) szeretnénk bizonyos állományokat, például egy lista elemeihez rendelt letöltött képeket vagy egyéb, a weben át elérhető erőforrásokat. Itt egyrészt figyelniünk kell arra, hogy a tárolt fájlok mérete ne foglaljon aránytalanul nagy területet a háttértárunkon, másrészt pedig ne legyenek benne elévült adatok. Ha nagyon gondosan akarunk eljárni, akkor az alkalmazásunknak a szabad lemezterület függvényében is „eszébe juthatna” kiüríteni ezt a könyvtárat, sőt a legjobb az lenne, ha ehhez az alkalmazásunknak nem is kellene éppen futnia, hanem az operációs rendszer maga végezné el a műveletet. Ez nem is nagy ördögösség, mindössze egy egyezményes nevű alkönyvtárban kellene tárolnunk ezeket az adatokat. Az Android biztosít minden alkalmazáshoz egy ilyen könyvtárat. Ennek elérési útját nem nekünk kell kitalálni, egyszerűen meghívhatjuk a *getCacheDir()* függvényt, amely visszaadja a gyorsítótárazásra szánt könyvtárat egy *File* objektum képében. Érdekes szem előtt tartani, hogy ennek mérete ne legyen túl nagy, 1–2 megabyte-nál többet tipikusan nem illik tárolni benne. A *deleteFile()* használható egy állomány eltávolítására, illetve a *fileList()* segítségével egy *String* tömbbe le is kérdezhetjük az állományok listáját.

5.1.1.3. Feltelepített, csak olvasható nyers adatfájlok elérése

Időnként előfordulhat, hogy az alkalmazásunk olyan nagyobb méretű külső adathalmazt használna, amely már telepítéskor a rendelkezésre áll, ám a tipikus erőforrások, például képek (*drawable*) vagy értékek (*values*) közé nehezen sorolhatók be. Ilyen lehet például az alkalmazásunk alapértelmezett vagy demonstrációs adatbázisa.

Fejlesztési időben ezeket az állományokat a */res/raw* könyvtárba kell másolnunk, és futás közben a belső tároló részeként érhetjük el. Az *InputStream* elérése kicsit eltér az előzőektől, például a „*myfile*” megnyitása nyers erőforrásként a következő:

```
Resources resources = getResources();
InputStream inStream =
    resources.openRawResource(R.raw.myfile);
```

Ezek az állományok azonban csak olvashatók lesznek. Mint minden más erőforrás, ezek is konfigurációtól függően változhatnak, amelyeket a *res* könyvtár konvenció szerinti elnevezésű alkönyvtáraiba kell másolnunk (pl.: *res/raw-hu_rHU*).

5.1.2. A nyilvános lemezterület használata

A másik fájlátrolási terület az, amikor mindenki által írható-olvasható területen szeretnénk adatokat kezelni. Az External Storage-ként megjelölt terület valamilyen gyors és a legtöbb operációs rendszer által kezelt fájlrendszert használ (pl. SD-kártyán tipikusan FAT, illetve valamelyik *yaffs* verzió).

Figyelni kell azonban arra, hogy például Mass Storage USB-kapcsolat esetén vagy az SD-kártya eltávolításakor ezek a területek leválasztódhatnak (*unmount*), és így ezek az adatok az alkalmazásunk számára elérhetetlenné válhatnak. Ezért hozzáférés előtt a platform által biztosított függvényekkel ellenőrizni kell, hogy rendelkezésre áll-e a kívánt terület. Ehhez az *Environment* osztály statikus *getExternalStorageState()* függvényére van szükségünk, amely *String* konstansok segítségével mutatja meg, hogy mi az adott tároló állapota. Ha ez a *MEDIA_MOUNTED* vagy a *MEDIA_MOUNTED_READ_ONLY* konstanssal egyenlő, írható vagy csak olvasható módon férhetünk hozzá az adott médiumhoz.

```
String state = Environment.getExternalStorageState();
// sokféle állapotban lehet, nekünk kettő fontos:
if (state.equals(Environment.MEDIA_MOUNTED)) {
    // Olvashatjuk és írhatjuk a külső tárat
} else if (state.equals(Environment.MEDIA_MOUNTED_
READ_ONLY)) {
    // Csak olvasni tudjuk
} else {
    // Valami más állapotban van, se olvasni,
    // se írni nem tudjuk
}
```

Mindemellett azonban egy éppen elérhető média is bármikor leválasztódhat. Az ebből fakadó hibák kivédésére futásidőben létrehozhatunk egy *intent* fílt, amellyel a média eltávolításáról a rendszer értesítést küld.

```
IntentFilter filter = new IntentFilter();
filter.addAction(Intent.ACTION_MEDIA_REMOVED);
//beregisztráljuk a saját broadcast receiverünket
registerReceiver(myExternalStorageReceiver, filter);
```

Az Android 2.2-es verziója előtt a nyilvános táron kaotikusan keveredtek az alkalmazások különböző célú felhasználásra szánt fájljai. Az operációs rendszer beépített fájlskenner-alkalmazása, amely a különböző médiumtípusokat azonosította, nehezen igazodott ki ezeken a vegyes könyvtárakon. Ezen az alapon a 2.2-es verzió az egyes fájl típusoknak az alkalmazás dedikált könyvtárán belül különböző alapértelmezett alkönyvtárakat javasol, amelyekhez konstansok segítségével férhetünk hozzá:

```
File f = GetExternalFilesDir(DIRECTORY_MUSIC);
```

Külön könyvtára (és így természetesen *String* konstansa) van a zenéknek, a csengőhangoknak, a képeknek, a filmeknek, a letöltött állományoknak stb. Ha kimondottan más alkalmazásoknak is szánjuk a kiírandó fájlokat, akkor a *getExternalStoragePublicDirectory(DIRECTORY_MUSIC)* stb. hívással kapjuk meg a média gyökérkönyvtárán belül az egyes típusok könyvtárait.

A 2.2 előtti verziókon kövessük az ajánlott konvenciót, miszerint a *getExternalStorageDirectory()* függvény segítségével elérhető gyökérkönyvtárán belül képezzük a saját alkalmazásunkhoz köthető könyvtárstruktúrát.

```
/Android/data/<package_name>/files/
```

A 2.2-es vagy későbbi Android operációs rendszerek ezeket az alkönyvtárakat az alkalmazás eltávolításakor letörlik.

Nyilvános meghajtón is van lehetőségünk cache-adatokat tárolni, ezekhez a *getExternalCacheDir()* függvénnyel férünk hozzá, illetve (2.2-es előtti rendszereken) a következő könyvtárba helyezzük a konvenció szerint:

```
/Android/data/<package_name>/cache/
```

Mivel a fentiek értelmében az egyes médiumtípusoknak külön alkönyvtárai vannak, az ezeket lejátszó vagy kezelő alkalmazásoknak egyszerű ezeket a fájlokat megkeresni és beindexelni. Ezt a szolgáltatást a *MediaScanner* osztály végzi a teljes nyilvános terület átnézésével. Ha az általa megtalált médiaállományok nem az alapértelmezett könyvtárunkban vannak, akkor a MIME-típusuk

alapján megpróbálja kitalálni a formátumot. Ha nem szeretnénk, hogy a nyilvános táron elhelyezett állományaink megjelenjenek a médiaalkalmazásokban, például az alkalmazásunk grafikus elemei a fényképalbumban, akkor egy üres, *.nomedia* nevű állományt kell elhelyeznünk az adott könyvtárba, amelyet így a szkennerek kihagy.

Nézzünk meg egy egyszerű példát, ahol a nyilvános könyvtárunk gyökerében szeretnénk egy rövid szövegállományt elhelyezni.

```
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // Írható a média
    File file =
        new File(getExternalFilesDir(null), "szoveg.
txt");
    try {
        file.createNewFile();
        Writer out = new BufferedFileWriter(new
FileWriter(file),
            1024);
        out.write("Hello!");
        out.close();
    }
    catch (IOException e)
    {
        //Nem sikerült írni
        Log.w("FileTest", "Nem tudtam kiírni: " + file, e);
    }
}
```

5.2. Beállítások tárolása a *SharedPreferences* segítségével

5.2.1. Kulcs-érték párok tárolása

Az előzőekben megismert fájlkezelési alapok, néhány Android-specifikus könyvtárat és függvényt leszámítva, nagyrészt ismerős lehetett a Java-programozóknak. Tipikus mobilalkalmazások esetén azonban gyakran szükség van a nyers állományoknál strukturáltabb adattárolásra, illetve a tárolt adatok változókra való egyszerű leképezésére (és fordítva). Ilyen szituáció például egy alkalmazás beállításainak, konfigurációjának vagy egy képernyő adott állapotának az elmentése.

A gyakorlatban a fenti feladatokra leginkább a kulcs-érték párok perzisztálása bizonyult a legidőállóbbnak. A legtöbb mobilplatformon találunk ennek

megfelelő osztályokat (a PythonS60 perzisztens szótáraitól a Windows Phone 7, illetve .NET izolált tárolójáig bezárólag). Androidon a *SharedPreferences* osztály nyújt ehhez segítséget, a háttérben nyers adatfájlokkal dolgozik. A platform készítői azonban észrevették, hogy ezeket a kulcs-érték párokat például konfigurációs adatok esetén a felhasználó valamilyen beállításdialóguson adja meg, így a programozó munkájának megkönnyítésére egy erre épülő, úgynevezett *Preferences* keretrendszert is kialakítottak (lásd alább).

A *SharedPreferences* keretrendszer tehát primitív adattípusokat (*int*, *long*, *float*, *String*, *boolean*) tud tárolni *String* kulcsok alatt úgy, hogy ezek az adatok túléljék az alkalmazás leállítását is. Ezek az adatok az Activity-hez tartozó privát területen tárolódnak el, így ugyanolyan hozzáférés-módosítók rendelkeznek hozzájuk (*MODE_PRIVATE*, *MODE_WORLD_READABLE*, *MODE_WORLD_WRITEABLE*). Ha egyetlen, alapértelmezett *SharedPreferences* szeretnénk kezelni, még név megadására sincs szükség (*getPreferences(int mode)* függvény). Nevesített állomány létrehozására, illetve megnyitására a *getSharedPreferences(String name, int mode)* függvény szolgál.

Közvetlenül nem írhatunk a *SharedPreferences* állományokba, csak egy *Editor* objektumon keresztül, amelyet az *edit()* függvényhívással kapunk meg. Ezek után elhelyezhetjük a tárolóban a kulcsokhoz rendelt értékeket a *put<típus>(String key, <típus> value)* függvénycsalád valamelyik tagjával, például *putString(String key, String value)* vagy *putBoolean(String key, Boolean value)*. *String* halmaz tárolására is van lehetőség a *putStringSet(String key, Set<String> value)* hívással. A *remove(String key)* hívás egyetlen kulcsot töröl a hozzárendelt értékkel, míg a *clear()* a teljes tartalmat.

A változtatások érvényre juttatására meg kell hívni a *commit()* vagy a vele teljesen ekvivalens, csak aszinkron módon működő *apply()* függvényt. A *SharedPreferences* objektumokat az Activity állapotváltásainál gyakran használjuk a felhasználói felület állapotának a rögzítésére. A keretrendszer gondoskodik arról, hogy az aszinkron *apply()* függvény végrehajtsódjon, mielőtt az állapotváltás megtörténne.

Az adatok visszaolvasásához nincs szükség semmilyen különleges objektumra, a *set()* függvények *get()* párja használható, az első paraméter a kulcs, a második paraméternek pedig megadhatunk egy alapértelmezett értéket, amelyet akkor szeretnénk visszakapni, ha a *SharedPreferences* objektum nem tartalmazza az adott kulcsot.

Egy egyszerű példán bemutatjuk a technika használatát. Ebben az alkalmazásban az adatok utolsó szinkronizációjának az idejét akarjuk elmenteni, majd visszatölteni, feltéve, hogy az alkalmazást nem először futtatjuk. Nevesített állományt használunk (*MySettings*), amelybe a *lastSyncTimestamp* kulcshoz rendeljük az aktuális időbélyeget, illetve a *firstRun* kulcshoz elmentjük a logikai *hamis* értéket. Visszaolvasásnál is ezeket az értékeket keressük, viszont a *firstRun* kulcshoz beállítunk egy alapértelmezett *igaz* értéket, hiszen akkor nem lesz érték elmentve ehhez a kulcshoz, ha még nem futott az alkalmazásunk.

Az írás kódrészlete a következő:

```
private final String PREF_NAME = "MySettings";
SharedPreferences sp =
    getSharedPreferences(PREF_NAME, MODE_PRIVATE);
Editor editor = sp.edit();
editor.putLong("lastSyncTimestamp",
    Calendar.getInstance().getTimeInMillis());
editor.putBoolean("firstRun", false);
editor.commit();
```

A visszaolvasáshoz tartozó kód a következő:

```
private final String PREF_NAME = "MySettings";
SharedPreferences sp =
    getSharedPreferences(PREF_NAME, MODE_PRIVATE);
long lastSaved = sp.getLong("lastSaved");
Boolean isFirstRun = sp.getBoolean("firstRun", true);
```

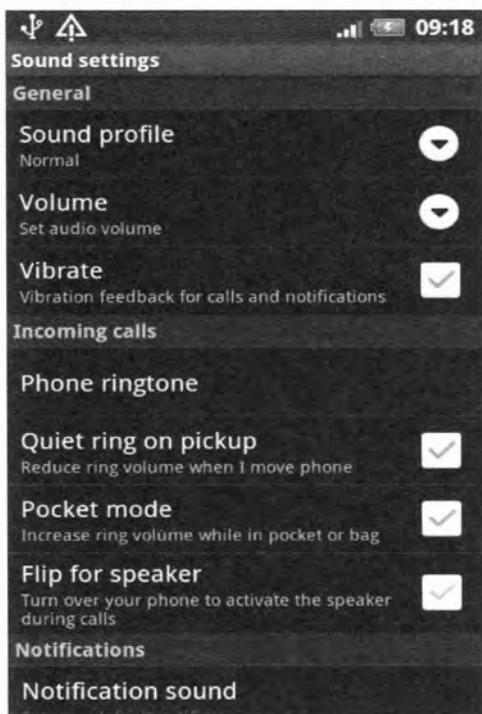
A *getAll()* metódus időnként nagy hasznunkra lehet, egy *Map* objektumban visszaad minden kulcs-érték párt.

5.2.2. A *Preferences* keretrendszer

5.2.2.1. A keretrendszer célja

Az alkalmazások beállításainak tárolására kifejezetten alkalmasnak tűnik a *SharedPreferences* technika. Ezeket a beállításokat vagy nagy részüket gyakran a felhasználó adhatja meg egy beállítás-képernyőn. Gondoljunk csak az operációs rendszer saját *Beállítások (Settings)* alkalmazására. Célszerűnek és a programozási idő szempontjából produktívnak tűnik tehát egy olyan megoldás, ahol az egyes beállításokat tartalmazó űrlapot automatikusan elő lehet állítani. A *Preferences* keretrendszer pontosan ezt a szolgáltatást nyújtja egy XML-alapú leíró és a *PreferenceActivity* segítségével.

Hogy az ebben a szakaszban leírt technikák segítségével milyen eredményt kapunk, jól mutatja a következő képernyőkép, amely a *Beállítások* alkalmazásról készült, és ez a program is a *Preferences* keretrendszerre épül.



5.1. ábra. PreferenceActivity alkalmazása a gyári beállításpanelen

5.2.2.2. A beállításokat leíró XML-erőforrás

Ahhoz tehát, hogy megvalósítsuk saját beállításnézetünket az Android által biztosított egyszerű módszerrel, két dologgal kell foglalkoznunk. Először is el kell készíteni egy erőforrás-XML-t, amely a szokásos nézeteink layoutleírását váltja fel. Ezt az állományt a *res/xml* mappában kell elhelyeznünk. Gyökereleme a *PreferenceScreen*, amely egy beállítás-képernyőt ír le. Ez az elem azonban nemcsak a gyökérben alkalmazható, hanem bárhol máshol is, ahol azt szeretnénk, hogy egy új képernyő nyíljon meg. A képernyőn belüli csoportosításhoz a *PreferenceCategory* XML-csomópontokat használhatjuk, amelynek a *title* tulajdonsága jelenik meg vizuális elválasztóként.

A kategórián belül következnek a különböző beállítások, amelyek egy-egy sornak felelnek meg, például: *CheckBoxPreference*, *EditTextPreference*, *ListPreference*, *RingTonePreference*, *DialogReference* stb. Ezek mindegyike tartalmaz egy címkét (*title* tulajdonság), egy rövid leírást (*summary*), amelyet akár a beállítástól függően dinamikusan változtathatunk is, valamint egy kulcsot (*key*) és egy értéket (*value*) a mögöttes *SharedPreferences* állományba történő mentéshez. Alapértelmezett értéket a *defaultValue* tulajdonsággal adhatunk meg. A *Preference* osztályból leszármaztatva lehetőségünk van saját beállítást is készíteni.

Nézzünk meg ezek után egy példát egy saját alkalmazásbeállítás XML-jére. Ebben egy jelölőnégyzetet helyezünk el, amely az automatikus beléptetés engedélyezését kéri a felhasználótól.

```
<?xml version="1.0" encoding="UTF-8"?>
<PreferenceScreen
    xmlns:android=http://schemas.android.com/apk/res/
    android">
    <PreferenceCategory android:title="Beléptetés">
        <CheckBoxPreference android:title="Automatikus be-
        lépés"
            android:key="autologin"
            android:defaultValue="false" />
    </PreferenceCategory>
</PreferenceScreen>
```

Természetesen a felületen megjelenő szövegelemeket sokkal inkább szöveges erőforrásokból (*strings.xml*) kellene hivatkozni, ebben a kódrészletben csak az átláthatóság kedvéért „drótoztuk be” a feliratokat.

Az Android készítői nem szeretik kétszer dolgoztatni a programozókat. A rendszer modulárisan épül fel, és minden elem, amelyet több helyen használhatunk, valamilyen módon futásidőben újrahasznosítható. Emiatt nem kell például egy kontakt adat kiválasztásához saját Activityt készítenünk, hanem használhatjuk a beépítettet (vagy azt, amire ezt a felhasználó lecserélte). Az egyes lazán csatolt elemek között Intentekkel kommunikálunk.

Ugyanezen elgondolások alapján kézenfekvőnek tűnik, hogy bizonyos beállítások halmazát, képernyőjét újrahasznosítsuk. Így nem kell sokat gépelni ahhoz, hogy újra megírjuk például a kijelző beállításait, ha már egyszer ezt a panelt az operációs rendszer is szállítja. A *Preferences* keretrendszer XML-jében elhelyezhetünk olyan képernyőket, amelyek tartalmát más alkalmazások beállításáiból szerzünk meg. Ehhez egy Intentet kell beraknunk, amelynek *action* elemét a másik alkalmazás beállításainak *intent filtere* el tudja kapni. Az operációs rendszer saját beállításaihoz tartozó *action*ök az *android.provider.Settings* osztályban vannak felsorolva. A következő részlet a képernyő-beállításokat helyezi el a saját beállításunk XML-jében:

```
<PreferenceScreen android:title="GPS engedélyezése">
    <intent
        android:action="android.settings.ACTION_LOCATION_
        SOURCE_SETTINGS"/>
</PreferenceScreen>
```

Hasonlóan a saját beállításainkat is kijáánhatjuk egy megfelelő *intent filter*-nek az *AndroidManifest* állományban történő elhelyezésével.

```
<activity android:name=".UserPreferences"
    android:label="Beállítások">
    <intent-filter>
        <action android:name="amorg.teszt.ACTION_USER_
PREFERENCE" />
    </intent-filter>
</activity>
```

5.2.2.3. A beállításokhoz tartozó Activity

A beállítások képernyőjéhez tartozó Activityt is előkészítették a programozóknak, így ilyenkor ezt a *PreferenceActivity*-ből kell leszármaztatnunk, és természetesen az *AndroidManifest* állományba ugyanúgy be kell vezetnünk az Activityt. Az eddigiektől eltérően viszont nem egy layoutot kell a nézetünkhöz rendelni, hanem a *Preferences* XML-t kell megadnunk az *onCreate()* függvényben, például: *addPreferencesFromResource(R.xml.prefs)*.

A 3.0-s (Honeycomb) verzió óta nemcsak egy, hanem több különböző összefüggő beállításkészlet is kezelhető, ezekhez *PreferenceFragment*-leszármazott osztályokat kell létrehozni, és ehhez kell a hozzájuk tartozó erőforrást betölteni, szintén az *addPreferencesFromResource(R.xml.prefs)* függvénnyel. Ebben az esetben a *PreferenceActivity* feladata az lesz, hogy az egyes beállításkészletekhez tartozó címkéket (fejléceket) visszaadja a kötelezően megvalósítandó *onBuildHeaders(List)* függvényben. Kis képernyők esetén a beállítások első nézete a fejléceket felsoroló lista, és az elem kiválasztása után kerül a vezérlés a beállítások űrlapjára. Nagy képernyőn egy osztott nézetben jelennek meg bal oldalt a fejlécek, jobb oldalt pedig a kiválasztott elemhez tartozó űrlap.

A *Preferences* keretrendszer, illetve a hozzá tartozó Activity által csak az alapértelmezett *SharedPreferences* fájljába menthetünk értékeket, maximum egy ilyen fájl lehet alkalmazásonként. Ennek elérése a következő:

```
SharedPreferences defaultPrefs =
    PreferenceManager.getDefaultSharedPreferences(
        getApplicationContext());
```

Előfordulhat, hogy szeretnénk értesítést kapni arról, ha valamelyik beállítás megváltozik (hogy érvényre juttassuk vagy megváltoztassuk a leírását a nézetben). Ehhez a *PreferenceActivity*-nknek implementálnia kell az *OnSharedPreferenceChangeListener* interfészt, amely az *onSharedPreferenceChanged()* metódus megírását jelenti. Ez a függvény megkapja a szóban forgó *SharedPreferences* objektumot, illetve a változtatott értékkulcsát tartalmazó

Stringet. Ahhoz, hogy ez a függvény meg is hívódjon, a *SharedPreferences* objektumon keresztül be kell állítanunk a *listener* osztályt, ahogy a lenti példakód is mutatja.

Az eddigieket szemléltető kódrészlet a következő:

```
public class MyActivity extends PreferenceActivity
    implements
        OnSharedPreferencesChangeListener {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate();
        AddPreferencesFromResource(R.xml.preferences);
        SharedPreferences settings =
            PreferenceManager.getDefaultSharedPreferences(this);
        settings.registerOnSharedPreferencesChangeListener(
            this);
    }

    @Override
    public void onSharedPreferencesChanged(
        SharedPreferences prefs, String key) {
        // prefs-ben lévő, key kulcsú beállítás megváltozásának
        // lekezelése
    }
}
```

5.3. Példányszintű adatok elmentése

Nem minden esetben célravezető a *SharedPreferences* (vagy akár nyers állományok) használata az adatok elmentéséhez. Akkor, amikor egy futó alkalmazás példányszintű adatait szeretnénk megőrizni, az Android Activity-életciklus modelljének függvényei adnak választ. Vegyük például azt az esetet, amikor a felhasználó, aki álló képernyővel tartja a telefont, hirtelen egy űrlapot tartalmazó képernyővel szembesül. Elkezd gépelni, majd rájön, hogy a készüléket elforgatva kényelmesebb a karakterek bevitele. Ám a telefon elforgatása konfigurációmódosítást jelent, ez pedig az adott Activity újraindítását eredményezi. Ilyenkor elvesznének a már beírt adatok, ez pedig nem lenne felhasználóbarát viselkedés. Ezen segíthetünk a példányszintű állapot lementésével. Amikor az alkalmazást, illetve Activityt „normál” módon zárjuk le, ezeknek az adatoknak az elmentésére nincs szükség. Például, amikor a felhasználó a *Vissza* gombot lenyomja, az állapotmentés nem fut le.

A fent vázolt esetre az Android az *onSaveInstanceState()*, illetve az *onRestoreInstanceState()* függvényt páros bevetésével segít a programozóknak. A *Paused*, illetve *Stopped* állapotban az alkalmazás állapotát leíró változók a memóriában tárolódnak. Az *onSaveInstanceState()* függvény (illetve természetesen az általunk felülírt megvalósítása) meghívódik minden esetben, mielőtt még az Activitynket bármilyen okból kilőné a rendszer (pl. háttérben van, és nincs elég memória az előtérben levő alkalmazás futtatásához, konfigurációváltáshoz stb.). A függvény futtatása az *onStop()* eseménykezelő előtt történik meg, az azonban nem egyértelmű, hogy az *onPause()* előtt, avagy után.

Az *onSaveInstanceState()* függvény egy *Bundle*-típusú változót kap, amelybe elmentheti a szükséges állapotváltozókat. Ezt a *Bundle* változót adja át az operációs rendszer az Activity újraindulásakor egyrészt az *onCreate()*, másrészt az *onRestoreInstanceState()* függvénynek. Ha az *onCreate()* függvényen belül meghívjuk az ősz osztály eredeti függvényét (*super.onCreate(savedInstanceState)*), ez néhány dolgot visszaállít. Például a legtöbb, *id*-vel rendelkező felhasználófelület-widget állapotát (*EditText*, *CheckBox*), viszont például egy *TextView* nem őrzi meg a szövegét, vagy a *Spinner* a kiválasztást. Ezt az automatikus funkcionalitást letilthatjuk, ha az adott elemre beállítjuk az *android:saveEnabled="false"* tulajdonságot.

A tipikus használat tehát a következőképpen néz ki. Az *onSaveInstanceState()* paramétereként kapott *Bundle*-be beleírunk mindent, ami a felhasználói felületen megjelenik, és nem mentődik automatikusan, vagy egyéb okból szükséges lehet visszaállítanunk. Ilyenek lehetnek a listák aktuálisan kiválasztott elemei vagy azok a tagváltozók, amelyek futás közben változhattak stb. Az *onCreate()* függvényben pedig (vagy sok adat esetén az *onRestoreInstanceState()* függvényben) megvizsgáljuk, hogy a kapott paraméter nem null-e (történt-e állapotmentés, vagy friss indításról beszélünk), és ha tartalmaz adatokat, akkor azokból helyreállítjuk az állapotot.

A következő példakódrészlet azt feltételezi, hogy létezik egy „*text*” *id*-vel ellátott szöveges mező, amelynek külön el szeretnénk menteni az állapotát valamiért (pl. változik), illetve egy *String* tagváltozó, amelynek az értékét szintén helyre szeretnénk állítani a futó alkalmazás visszaállításakor.

```
public class MainActivity extends Activity {

    private String myString;
    private TextView text;
    private final String KEY_MYSTRING = "str";
    private final String KEY_TEXT     = "text";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
text = (TextView)findViewById(R.id.text);

myString = "str kezdeti erteke, valtozhat futas
soran";
if(savedInstanceState != null) {
    if(savedInstanceState.containsKey(KEY_MYSTRING))
        myString = savedInstanceState.getString(KEY_
MYSTRING);
    if(savedInstanceState.containsKey(KEY_TEXT))
        text.setText(savedInstanceState.getString(KEY_
TEXT));
}
}

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putString(KEY_MYSTRING, myString);
    outState.putString(KEY_TEXT, (String)text.
getText());
}
}
```

Strukturált adatok tárolása

6.1. Az SQLite-adatbázismotor

Strukturált adatok tárolására máig a leginkább elfogadott módszer a relációs adatbázisok használata. Természetesen nem ez az egyetlen lehetőség arra, hogy az adatainkat valamilyen rendezőelv szerint elmentsük, illetve biztosítsuk a manipulációjukat és lekérdezésüket, viszont tagadhatatlanul ez a leginkább bevált és közismert megoldás.

A mobileszközök hálózatra kapcsolásának természetessé válása azt is jelenti, hogy komoly alternatívaként merül fel még egyszerűbb rendszereknél is, hogy az akár csak egy felhasználóhoz kötődő adatokat is szerveroldalon tároljuk, és ezeket a mobileszköz szinkronizálja. Ám a „pusztán online” működésnek vannak hátulütői. Ha az adatforgalom költségét nem is nézzük, azt el kell ismernünk, hogy a hordozható eszközeink hálózati kapcsolata nem stabil: ha nincs hálózati lefedettség, és épp WLAN-hálózathoz sem tud a mobil kapcsolódni, akkor az adatokat a csak szerveroldalon tároló alkalmazás működésképtelenné válik. Elkerülhetetlen tehát a strukturált üzleti, felhasználói adatok lokális tárolása. A lokális tárolásnak az okostelefonok hardverkapacitási növekedésével semmilyen akadálya nincs.

Az Android platform alapvetően tartalmaz egy teljes értékű relációs adatbáziskezelő komponenst, mégpedig a – többek között – más mobilplatformokról ismerős SQLite-ot, egy C nyelven írt, nyílt forráskódú, kisméretű osztálykönyvtárat, amelynek egyik jellemzője az, hogy egy adatbázisszerver nem külön processzként fut, hanem a szervert használó alkalmazáshoz linkelve, annak részeként. Maga az adatbázis pedig egy fájlban van eltárolva (ez a fájl hordozható, vagyis nem függ a konkrét futtató platformtól). Android alatt az adatbázist a neve alapján tudjuk az alkalmazásunkból azonosítani. Maga az adatbázis más alkalmazások számára nem érhető el, ehhez *Content Providert* kell készítenünk. Az SQLite további jellegzetessége, hogy *gyengén típusos*, vagyis az egyes oszlopoknak nincs szigorúan meghatározott adattípusa (csak a konkrét eltárolt értékeknek).

Az Android nem biztosít alpból objektumrelációs réteget az SQLite-adatbázis fölé, tehát saját adataink eltárolásához nekünk kell az SQL-lekérdezéseket megírunk.

6.2. Az adatbázis-elérés általános menete

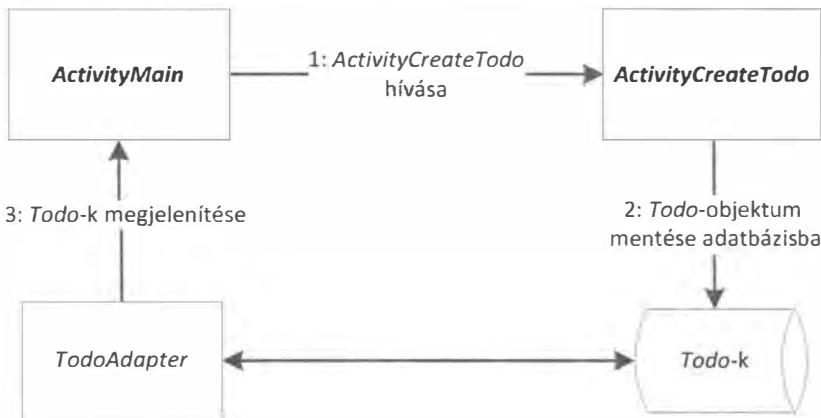
Adatbázisok használatához alapvetően a következő műveleteket kell megvalósítani:

- Adatbázisok létrehozása, illetve megnyitása. Ehhez a művelethez az Android az *SQLiteOpenHelper* osztály felüldefiniálását javasolja, amelyben az *onCreate()* metódus felüldefiniálásával megírhatjuk és futtathatjuk az adatbázis sémáját létrehozó SQL-utasítást. Emellett az *onUpgrade()* függvényt kell még feltétlenül felüldefiniálni, amely egy régebbi adatbázis-verzióról az újbóli migrációt végző utasításokat tartalmazza.
- Az adatbázis konkrét elérése az így létrehozott segédosztály *getWritableDatabase()*, illetve *getReadableDatabase()* függvényeivel történik, írásra vagy csak olvasásra.
- Az adatbázis használata tipikusan az adatbázis-objektumon hívott *execSQL()*, illetve *query()* függvények valamelyik variánsának a segítségével történik. Ha összetettebb lekérdezést szeretnénk összeállítani, ehhez az *SQLiteQueryBuilder* segédosztályt használhatjuk.
- A lekérdezések eredményein, jól ismert módon, egy *Cursor* objektummal tudunk végigmenni. Ez az eredményhalmazon rekordról rekordra tud lépni, illetve az egyes oszlopokat, azok metaadatait lekérdezni.
- Az adatok beszúrása (INSERT), illetve módosítása (UPDATE) pedig a *ContentValues* osztályok segítségével történik, ez egy *Map*-típusú adatszerkezet, amelyet a *ContentResolver* fel tud dolgozni.

6.3. Az adatbáziskezelő használata

Az SQLite képességeinek megismeréséhez ismét megnézzük a felhasználói felületet bemutató fejezetben elkezdett teendőlista-alkalmazásunkat. Ott tartottunk benne, hogy az alkalmazás újraindításakor a *Todo* elemek elvesztek, mivel még nem tároltuk el őket a perzisztens tárbá. Adatbázis használatához három feladatot kell elvégeznünk:

- a *Todo* objektumok tárolása adatbázisban,
- listával dolgozó adapter átalakítása adatbázissal (*Cursorral*) működővé,
- a vezérlőlogika átalakítása.



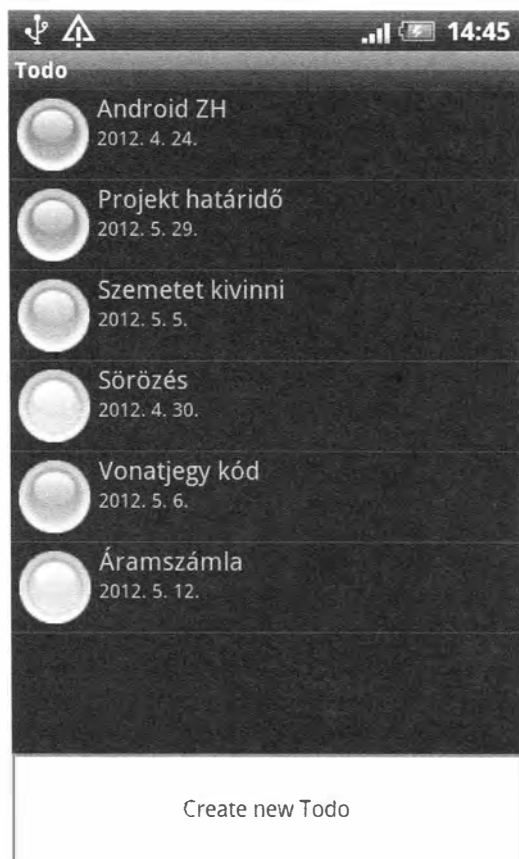
6.1. ábra. Új Todo elem létrehozása a továbbfejlesztett alkalmazásban

6.4. Teendőelemek tárolása adatbázisban

Első lépésként kihasználjuk az *SQLiteOpenHelper* osztály által nyújtott segítséget. Ebből származtatva olyan saját osztályt hozunk létre, amely referenciát szolgáltat az egész általunk használt adatbázisra, így tehát több entitásosztály és adatbázistábla esetén is elegendő egy ilyen segédosztály. Hozzuk létre az osztályt *DatabaseHelper* néven egy új package-be.

Az *SQLiteOpenHelper* osztályból való származtatás a konstruktoron kívül az *onCreate()* és az *onUpgrade()* absztrakt metódusok felüldefiniálását írja elő. Az *onCreate()*-ben futtatjuk a sémalétrehozó SQL-szkriptet, míg az *onUpgrade()*-ben kezelhetjük le az adatbázis verzióváltásával kapcsolatos feladatait. Legegyszerűbb esetben itt töröljük, majd újra létrehozuk az egész sémát, ám ekkor az adatokat is elveszítjük.

Az adatbáziskezelés során sok konstans jellegű változóval kell dolgoznunk, például a táblákban lévő oszlopok neveivel, a táblák nevével, az adatbázisfájl nevével, a sémalétrehozó és -törölő szkriptekkel stb. Ezeket érdemes egy közös helyen tárolni, így szerkesztéskor vagy új entitás bevezetésekor nem kell a forrásfájlok között ugrálni, valamint egyszerűbb a teljes adatbázist létrehozó és törölő szkripteket generálni. Hozzuk létre egy új osztályt *DbConstants* néven, és statikus tagváltozóként vegyünk fel minden szükséges konstanst. A sorok elsődleges kulcsára nincs kötelező előírás az SQLite részéről, viszont ahhoz, hogy a későbbiekben az adatbázisunkat más Activityk számára is elérhetővé tegyük a *ContentProvider* mechanizmuson keresztül, előrelátóan az *_id* névvel illetjük. Ez az oszlopnév jelenti a *ContentProvider* számára a rekordok egyedi azonosítóját. Mivel ezt az oszlopot egyedinek szeretnénk tudni, olyan integerértéket rendelünk hozzá, amely még nem szerepel más sornál. Ezt legegyszerűbben az SQLite-ra bízva tehetjük meg, mégpedig úgy, hogy *autoincrement* kulcsszóval látjuk el az oszlopot.



6.2. ábra. A Todo-lista felülete

```
public class DbConstants {  
  
    // fajlnev, amiben az adatbázis lesz  
    public static final String DATABASE_NAME = "data.db";  
    // verziószám  
    public static final int DATABASE_VERSION = 1;  
    // összes belső osztály DATABASE_CREATE szkriptje  
    // összefüztve  
    public static String DATABASE_CREATE_ALL =  
        Todo.DATABASE_CREATE;  
    // összes belső osztály DATABASE_DROP szkriptje  
    // összefüztve  
    public static String DATABASE_DROP_ALL = Todo.  
        DATABASE_DROP;  
}
```

```

/* Todo osztály DB konstansai */
public static class Todo{
    // tabla neve
    public static final String DATABASE_TABLE = "todo";
    // oszlopnevek
    public static final String KEY_ROWID = "_id";
    public static final String KEY_TITLE = "title";
    public static final String KEY_PRIORITY =
"priority";
    public static final String KEY_DUEDATE = "dueDate";
    public static final String KEY_DESCRIPTION =
        "description";
    // sema létrehozó szkript
    public static final String DATABASE_CREATE =
        "create table if not exists "+DATABASE_TABLE+"
("
    + KEY_ROWID +" integer primary key
autoincrement,"
    + KEY_TITLE + " text not null, "
    + KEY_PRIORITY + " text, "
    + KEY_DUEDATE +" text, "
    + KEY_DESCRIPTION +" text"
    + "); ";
    // sema törölő szkript
    public static final String DATABASE_DROP =
        "drop table if exists " + DATABASE_TABLE + "; ";
}
}

```

Ha megvannak a konstansok, töltsük ki a *DatabaseHelper* osztály metódusait. A konstruktor paraméterei közül törölhetjük a *CursorFactory*-t és a verziószámot, majd az *Ősosztály* konstruktorának hívásakor a megfelelő helyeken adjunk át *nullt* (így a default *CursorFactory*-t használja), valamint a *DbConstants.DATABASE_VERSION* *String*-et verzióként.

```

public class DatabaseHelper extends SQLiteOpenHelper {

    public DatabaseHelper(Context context, String name)
    {
        super(context, name, null, DbConstants.DATABASE_
VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DbConstants.DATABASE_CREATE_ALL);
    }
}

```

```

    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int
oldVersion,
        int newVersion) {
        db.execSQL(DbConstants.DATABASE_DROP_ALL);
        db.execSQL(DbConstants.DATABASE_CREATE_ALL);
    }
}

```

A séma létrehozásához és megnyitásához szükséges osztályok rendelkezésünkre állnak, így a következő feladatunk az entitásosztályok felkészítése lesz arra, hogy az adatbázisból használjuk őket. Ehhez meg kell írunk azokat a kódrészleteket, amelyek egy memóriában lévő *Todo* objektumot képesek az adatbázisba írni, onnan visszaolvasni, módosítani, valamint törölni (természetesen más funkciók is szükségesek lehetnek). Ezt a kódot az entitásosztály (*Todo*) helyett érdemes egy külön osztályban megvalósítani (*TodoDbLoader*):

```

public class TodoDbLoader {

    private Context ctx;
    private DatabaseHelper dbHelper;
    private SQLiteDatabase mDb;

    public TodoDbLoader(Context ctx) {
        this.ctx = ctx;
    }

    public void open() throws SQLException{
        // DatabaseHelper objektum
        dbHelper = new DatabaseHelper(
            ctx, DbConstants.TODO.DATABASE_NAME);
        // adatbázis objektum
        mDb = dbHelper.getWritableDatabase();
        // ha nincs még séma, akkor létrehozuk
        dbHelper.onCreate(mDb);
    }

    public void close(){
        dbHelper.close();
    }
    // CRUD és egyéb metódusok (azonnal folytatjuk)
}

```

Az adat létrehozását, módosítását, törlését végző tagfüggvényeket összefoglaló néven, az SQL-kulcsszavak kezdőbetűiből képzett szóval CRUD (CReate, Update, Delete) metódusoknak hívjuk.

```
// INSERT
public long createTodo(Todo todo){
    ContentValues values = new ContentValues();
    values.put(DbConstants.TODO.KEY_TITLE, todo.
getTitle());
    values.put(DbConstants.TODO.KEY_DUE_DATE, todo.
getDueDate());
    values.put(DbConstants.TODO.KEY_DESCRIPTION,
        todo.getDescription());
    values.put(DbConstants.TODO.KEY_PRIORITY,
        todo.getPriority().name());

    return mDb.insert(DbConstants.TODO.DATABASE_TABLE,
        null, values);
}

// DELETE
public boolean deleteTodo(long rowId){
    return mDb.delete(
        DbConstants.TODO.DATABASE_TABLE,
        DbConstants.TODO.KEY_ROWID + "=" + rowId, null) >
0;
}

// UPDATE
public boolean updateProduct(long rowId, Todo newTodo)
{
    ContentValues values = new ContentValues();
    values.put(DbConstants.TODO.KEY_TITLE, newTodo.
getTitle());
    values.put(DbConstants.TODO.KEY_DUE_DATE,
        newTodo.getDueDate());
    values.put(DbConstants.TODO.KEY_DESCRIPTION,
        newTodo.getDescription());
    values.put(DbConstants.TODO.KEY_PRIORITY,
        newTodo.getPriority().name());
    return mDb.update(
        DbConstants.TODO.DATABASE_TABLE,
        values,
        DbConstants.TODO.KEY_ROWID + "=" + rowId ,
        null) > 0;
}
```

Az adatbázis leképezését végző osztályoknál az adatok lekérdezésére az esetek túlnyomó többségében legalább két lekérdezőfüggvényt szoktak megvalósítani. Az egyik függvény az összes adat elérését lehetővé teszi (tipikusan *fetchAll()* néven): ez egy *Cursor* objektummal tér vissza. Akik az egyes szoftverrétegek szétválasztását fontosnak tartják, azok gyakran kritikával illetik ezt az eljárást, ugyanis a függvény egy adatréteghez kapcsolódó objektumot ad vissza egy felsőbb rétegnek, például egy általánosabb adatrepresentálást jelentő kollekció helyett. Ám a *Cursor* objektum kikerülésével éppen az adatok elérésének optimalizálását veszítenénk el (például amiatt, hogy az összes adatot a memóriában kellene tartani, és erre gyakran fizikai korlátok miatt nincs lehetőség). Másik megoldásként az osztályunk megvalósíthatja az adatok eredményhalmazán való navigálást végző interfészt, ezzel gyakorlatilag a *Cursor* interfészét nyújtjuk kifelé, esetleg más néven. Arra az eredményre jutunk tehát, hogy még mindig jobban járunk, ha ténylegesen a *Cursor*-t, mint az adatrétegbeli osztályt exponáljuk a felsőbb rétegek számára.

A másik függvény pedig egyetlen bejegyzést kérdez le az egyedi azonosítója (tipikusan *_id*) alapján. Ennek a két függvénynek a megírását a *ContentProvider* működési mechanizmusa miatt, annak előírásai okán (összes sor, illetve *id*-vel azonosított rekord elérése előírt *URI*-val) gyakran nem is kerülhetjük el.

```
// minden Todo lekérése
public Cursor fetchAll() {
    // cursor minden rekordra (where = null)
    return mDb.query(
        DbConstants.TODO.DATABASE_TABLE,
        new String[]{
            DbConstants.TODO.KEY_ROWID,
            DbConstants.TODO.KEY_TITLE,
            DbConstants.TODO.KEY_DESCRIPTION,
            DbConstants.TODO.KEY_DUEDATE,
            DbConstants.TODO.KEY_PRIORITY
        }, null, null, null, null, DbConstants.TODO.KEY_
TITLE);
}

// egy Todo lekérése
public Todo fetchTodo(long rowId) {
    // a Todo-ra mutató cursor
    Cursor c = mDb.query(
        DbConstants.TODO.DATABASE_TABLE,
        new String[]{
            DbConstants.TODO.KEY_ROWID,
            DbConstants.TODO.KEY_TITLE,
            DbConstants.TODO.KEY_DESCRIPTION,
            DbConstants.TODO.KEY_DUEDATE,
            DbConstants.TODO.KEY_PRIORITY
        }, null, null, null, null, DbConstants.TODO.KEY_
TITLE);
    if (c != null) {
        Todo t = new Todo(c);
        c.close();
        return t;
    }
    return null;
}
```

```

    }, DbConstants.TODO.KEY_ROWID + "=" + rowId,
    null, null, null, DbConstants.TODO.KEY_TITLE);
// ha van rekord amire a Cursor mutat
if(c.moveToFirst())
    //ezt később megírjuk
    return getTodoByCursor(c);
// egyébként null-al térünk vissza
return null;
}

```

6.5.A TodoAdapter átalakítása

Ha az *Adapter* osztályunkat adatbázisból szeretnénk feltölteni, akkor a *BaseAdapter* helyett a *CursorAdapter* őssztályból kell származtatnunk. Ez a konstruktor megírásán kívül két metódus implementációját írja elő kötelezően. A *newView(Context context, Cursor cursor, ViewGroup parent)* létrehoz egy sornak megfelelő *View*-t, amelyet adatokkal tölt fel, és visszatér vele. A takarékos erőforrás-felhasználás érdekében az adatfeltöltést érdemes a másik kötelező metódusban, a *bindView(View view, Context context, Cursor cursor)*-ban végezni. A forráskód a következő:

```

public class TodoAdapter extends CursorAdapter {

    public TodoAdapter(Context context, Cursor c) {
        super(context, c);
    }

    @Override
    public View newView(Context context, Cursor cursor,
        ViewGroup parent)
    {
        final LayoutInflater inflater =
            LayoutInflater.from(context);
        View row = inflater.inflate(R.layout.todorow, null);
        bindView(row, context, cursor);
        return row;
    }

    // UI elemek feltöltése
    @Override
    public void bindView(View view, Context context,
        Cursor cursor)
    {

```



```

// referencia a UI elemekre
TextView titleTV = (TextView) view.findViewById(
    R.id.textViewTitle);
TextView dueDateTV = (TextView) view.findViewById(
    R.id.textViewDueDate);
ImageView priorityIV = (ImageView) view.
findViewById(
    R.id.imageViewPriority);

// Todo példányosítás Cursorból
Todo todo = TodoDbLoader.getTodoByCursor(cursor);

// UI elemek
titleTV.setText(todo.getTitle());
dueDateTV.setText(todo.getDueDate());
switch (todo.getPriority()) {
    case HIGH:
        priorityIV.setImageResource(R.drawable.high);
        break;
    case MEDIUM:
        priorityIV.setImageResource(R.drawable.
medium);
        break;
    case LOW:
        priorityIV.setImageResource(R.drawable.low);
        break;
}
}
}
}

```

Az implementáció hivatkozik a *TodoDbLoader* egyik statikus metódusára (*getTodoByCursor()*), amely egy rekordra állított *Cursor*-t kap paraméterként, és egy *Todo* objektummal tér vissza. Ennek a kódja triviális, viszont ugyanez a funkcionalitás szerepel a *fetchTodo()* metódusban is, így érdemes ott is ezt az új függvényt használni.

```

// egy Todo lekérése
public Todo fetchTodo(long rowId) {
    // a Todo-ra mutató cursor
    Cursor c = mDb.query(
        DbConstants.TODO.DATABASE_TABLE,
        new String[]{
            DbConstants.TODO.KEY_ROWID,
            DbConstants.TODO.KEY_TITLE,
            DbConstants.TODO.KEY_DESCRIPTION,
            DbConstants.TODO.KEY_DUEDATE,

```

```

        DbConstants.TODO.KEY_PRIORITY
    }, DbConstants.TODO.KEY_ROWID + "=" + rowId,
    null, null, null, DbConstants.TODO.KEY_TITLE);
// ha van rekord amire a Cursor mutat
if(c.moveToFirst())
    return getTodoByCursor(c);
// egyebkent null-al terunk vissza
return null;
}

public static Todo getTodoByCursor(Cursor c){
    return new Todo(
        //title:
        c.getString(c.getColumnIndex(DbConstants.TODO.KEY_TITLE)),
        //priority:
        Priority.valueOf(c.getString(c.getColumnIndex(
            DbConstants.TODO.KEY_PRIORITY))),
        //dueDate:
        c.getString(c.getColumnIndex(DbConstants.TODO.KEY_DUEDATE)),
        //description:
        c.getString(c.getColumnIndex(
            DbConstants.TODO.KEY_DESCRIPTION)) // description
    );
}

```

6.6. A vezérlőlogika átalakítása

A fentiek megírásával gyakorlatilag készen vagyunk ahhoz, hogy a *Todo* elemeket adatbázisban tároljuk. Eddig azonban az *ActivityMain* osztályban listákat használtunk, hogy a teendőket gyűjteménybe foglaljuk. Meg kell írunk tehát még azt, hogy az adatok az újonnan létrehozott adatbázisból származzanak.

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    TodoDbLoader dbLoader = new
        TodoDbLoader(getApplicationContext());
    dbLoader.open();
    // kurzor minden rekordra
    Cursor c = dbLoader.fetchAll();
    // pozícionáljuk a kurzort

```

```

c.moveToFirst();
// példányosítjuk és beállítjuk az adaptert
todoAdapter = new TodoAdapter(getApplicationConte
xt(), c);
setListAdapter(todoAdapter);
}

```

Ha ezen a ponton kipróbáljuk az alkalmazást, azt látjuk, hogy nincsenek adatok, és ez teljesen jogos, tekintve, hogy csak az adatbázis sémáját hoztuk létre, ám adatokkal nem töltöttük fel. Ha szeretnénk a teszteléshez „rövidíteni”, egészítsük ki például a fenti kódunkat úgy, hogy üres tábla esetén néhány tesztrekordot helyezünk el a táblában.

```

// ha üres a tábla, akkor létrehozunk pár rekordot
if(c.getCount() == 0){
    // kezdeti elemek létrehozása
    dbLoader.createTodo(new Todo("title1", Priority.
LOW,
        "2012. 09. 26.", "description1"));
    dbLoader.createTodo(new Todo("title2", Priority.
MEDIUM,
        "2012. 09. 27.", "description2"));
    dbLoader.createTodo(new Todo("title3", Priority.
HIGH,
        "2011. 09. 28.", "description3"));
    // elavult a kurzor, ismét lekerjük az összes
    rekordot
    c = dbLoader.fetchAll();
}

```

Ha minden kódrészlet a megfelelő helyen van, akkor az alkalmazás indulásakor ugyanaz a felhasználói felület jelenik meg, mint a kiinduló projekt esetén, ám az elemek már adatbázisban tárolódnak.

Pozíciómeghatározás és térképkezelés

Mobilalkalmazás fejlesztésekor gyakran szembesülünk azzal a feladattal, hogy meg kell határoznunk a készülék pozícióját. Ilyenkor értelemszerűen a készülékbe épített GPS-vevő jut az eszünkbe, ám a mobiltelefonok esetében ennél sokkal több lehetőségünk is van, elegendő csupán a mobilhálózat-alapú helymeghatározásra vagy a cellaazonosítók lekérdezésére gondolnunk.

Általában azonban az ilyen feladatok több kérdést is felvetnek, amelyekkel mobilalkalmazás-fejlesztőként mindenképpen foglalkoznunk kell. Az első és legfontosabb kérdés, hogy milyen pontosságú helymeghatározásra van szükség, illetve hogy milyen sűrűn kellene a friss pozícióadatok. Ezek a tényezők nagyban befolyásolják az alkalmazásunk struktúráját, továbbá az alkalmazás energiaigényét is jelentősen megszabják.

Ebben a fejezetben elsőként megvizsgáljuk a helymeghatározási módszereket mobil eszközökön, majd bemutatjuk a hálózati és cellainformációk lekérdezésének módszerét, ismertetjük a pozíciókezelés lehetőséget Android platformon, valamint kitérünk a térképnézet használatára.

7.1. A helymeghatározás módszerei mobil eszközökön

A helymeghatározás mobil eszközökön is egyre népszerűbb, és egyre több alkalmazásban valósítják meg. Ilyenkor azt használjuk ki, hogy a készülék tulajdonképpen állandóan velünk van, és így az aktuális pozíciónk lekérdezhető, ez pedig számos alkalmazásnak sajátos színezetet ad.

Ha tehát ez az információ a rendelkezésünkre áll, számos érdekes funkciót valósíthatunk meg, például:

- közeli helyek listázása,
- navigáció,
- flottakövetés,
- geocaching,
- találkozásszervezés.

Ha ilyen alkalmazásokat fejlesztünk fontos azonban, hogy a technológia jellegéből eredő korlátokat folyamatosan szem előtt tartsuk. Ilyen korlátok a pontosság és az energiafogyasztás.

A pontosság esetében fontos, hogy egy új pozíció kiértékelésekor mindig kérdezzük le a pozíció pontosságát, és ne feledkezzünk el az időbélyegről sem, hiszen nem biztos, hogy az teljesen friss érték. Az optimális energiafogyasztás kialakításához fontos, hogy alaposan átgondoljuk a következőket:

- a pozíciófrissítés gyakorisága,
- a GPS-jel elvesztése esetén az újrapróbálkozás stratégiája,
- az új pozíció érkezésekor végrehajtandó műveletek,
- a feleslegesen bekapcsolt helymeghatározás kerülése.

A leggyakoribb, mobilkörnyezetben használt helymeghatározási módszerek a következők:

- GPS,
- mobilhálózat-alapú helymeghatározás,
- GPS és mobilhálózat együttműködése (Assisted-GPS) a műholdak becsült helyzetének felhasználása alapján,
- wifialapú helymeghatározás (a Google saját adatbázisa).

A következőkben tekintsük át ezeknek a módszereknek a lényegét.

7.1.1. Wifialapú helymeghatározás

A wifialapú helymeghatározás lényege az, hogy egy központi adatbázisban tárolódnak a wifihozzáférési pontok nevei és azok koordinátái, a telefon pedig az éppen látott wifi-hálózatok alapján a központi adatbázist felhasználva határozza meg a közelítőleges pozícióját. Az adatbázisok általában a BSSID, a MAC-cím és a jelerősség alapján határozzák meg a közelítő pozíciókat.

Számos adatbázis létezik, amelyek a wifi-hozzáférési pont nevei alapján közelítőleges koordinátákat szolgáltatnak. Az egyik ilyen rendszer az *openBmap*,¹⁵ amely egy kisebb adatbázis, de nyílt és ingyenes, illetve a közönsége folyamatosan bővíthető. Egy másik, nagyobb adatbázis a Google¹⁶ gondozásában található, amely egy megbízható, sok minta alapján működő rendszer, de használata licenchez kötött.

¹⁵ *openBmap*: <http://openbmap.org/>

¹⁶ Google wifi-location adatbázis: <http://code.google.com/p/gears/wiki/GeolocationAPI>

7.1.2. Cellaalapú helymeghatározás

Mobilkörnyezetben a helymeghatározás egyik tipikus módja, ha figyelembe vesszük a készülék által látott cellákat, és ez alapján határozzuk meg a pozíciót. A legtöbb esetben az adott mobilplatform és hálózat ezt automatikusan elvégzi, és egy megfelelő API segítségével a hálózat által megállapított pozíciót és pontosságot elérhetővé teszi, ám kerülhetünk olyan helyzetbe, hogy csak a cellaazonosító áll rendelkezésre, és az alapján kell becslést adnunk a pozícióra.

Cellaalapú helymeghatározás esetén általában a készülék által aktuálisan használt cella, a jelerősség mérése és a környező cellák alapján történik a helymeghatározás. Tipikusan valamilyen háromszögelési technológiát alkalmazunk, amely a jelidőt is figyelembe véve határozza meg a pozíciót.

Ha csupán a cellaazonosító áll rendelkezésre, a wifihez hasonlóan használhatunk különböző adatbázisokat, amelyek a cellák földrajzi pozícióit tárolják.

Ilyen adatbázisok például a következők:

- *OpenCellId*,¹⁷
- *openBmap*,
- Google-cella adatbázisa.¹⁸

7.1.3. GPS-alapú helymeghatározás

Mobilkészülék helymeghatározásához leggyakrabban a beépített GPS-vevőt használjuk. A GPS-technológiában eredetileg 24 MEO- (Medium Earth Orbit) műholdat használtak fel, ám a rendszer még hatékonyabb működéséhez 2008 márciusától már 32 aktív műhold áll rendelkezésre.

A GPS-vevő kellően kis mérete lehetővé tette, hogy mobil eszközökbe is beépítsék, így napjainkra tulajdonképpen a legtöbb mobil eszköz támogatja a technológiát. A GPS-műholdak által sugárzott információk az alábbiak:

- pontos idő,
- műholdpozíció,
- általános rendszerállapot.

A GPS-technológia esetén a műholdakról érkező jel közel fénysebességgel halad, a vevő pedig az érkezési idő felhasználásával számolja ki a pozíciót. Elviekben már három műholdtól érkező jel alapján ki lehetne számítani a pozíciót, de ehhez nanosecundum pontosságú időbélyegre lenne szükség. Négy műholdtól érkező jel alapján viszont már pontosan meg lehet adni a megfelelő, három dimenziót leíró értékeket (szélesség, hosszúság, magasság) és az időt laboratóriumi pontosságú óra nélkül.

¹⁷ *OpenCellId*: <http://www.opencellid.org/>

¹⁸ Google-cella adatbázis: <https://www.google.com/loc/json>

7.2. Cella- és hálózati információk lekérdezése

A hálózati és a cellainformációk lekérdezésének tárgyalása előtt fontos kitérünk az Android rendszerszolgáltatásainak a használatára. Általában minden rendszerközeli funkciót el kell kérnünk használat előtt a rendszertől a `getService()` függvény segítségével.

Például a telefon és a rádióegység eléréséhez használható *TelephonyManager* a következőképpen érhető el egy *Activity*-n belül:

```
TelephonyManager telephonyManager =
    ((TelephonyManager) getSystemService(
        Context.TELEPHONY_SERVICE));
```

A későbbi fejezetekben gyakran használjuk majd a rendszerszolgáltatást, ezért a teljesség igénye nélkül nézzünk meg néhányat:

- *Context.TELEPHONY_SERVICE*: telefónia-rendszerszolgáltatások,
- *Context.LOCATION_SERVICE*: helymeghatározást segítő szolgáltatás,
- *Context.CLIPBOARD_SERVICE*: vágólap-szolgáltatás,
- *Context.NOTIFICATION_SERVICE*: értesítésszolgáltatás,
- *Context.KEYGUARD_SERVICE*: képernyőzár-szolgáltatás.

A cellainformációk eléréséhez a korábban említett *TelephonyManager* referenciára lesz szükségünk, amelytől olyan általános információkat kérdezhetünk le, mint a hívás állapota, a cellaazonosító, az operátornév, a készülék IMEI-száma stb. A *TelephonyManager* segítségével továbbá feliratkozhatunk különféle telefoneseményekre is.

A mobilhálózat általában nagyobb területi egységekre van felosztva, amelyek egyedi LAC- (Location Area Code) azonosítóval rendelkeznek, és egy ilyen területen belül található több cella. A következőkben nézzünk meg egy egyszerű példát, amelyben a cellaazonosítót és a LAC-értékeket kérdezzük le.

```
TelephonyManager tm = (TelephonyManager)
getSystemService(
    Context.TELEPHONY_SERVICE);
GsmCellLocation loc = (GsmCellLocation)
tm.getCellLocation();
int cellid = loc.getCid();
int lac = loc.getLac();
```

Ahhoz, hogy az előző kódrész működhessen, a *manifest* állományban szükségünk van az *ACCESS_COARSE_LOCATION* engedélyre.

```
<uses-permission android:name=
    "android.permission.ACCESS_COARSE_LOCATION"/>
```

A következő példában kérdezzük le a szomszédos cellaazonosítókat és a hozzájuk tartozó jel-zaj viszonyt:

```
TelephonyManager tm = (TelephonyManager)
getSystemService(
    Context TELEPHONY_SERVICE);
List<NeighboringCellInfo> cellinfo =
    tm.getNeighboringCellInfo();
for(NeighboringCellInfo info: cellinfo){
    cellid = info.getCid();
    rssi = info.getRssi(); // jel-zaj
}
```

Sok telefonon a fenti kódrész nem szolgáltat adatot, ugyanis a készülék gyakran letiltja a szomszédos cellaadatok lekérdezését. Általános érvényű állítás, hogy minden telefonazonosításhoz és hálózathoz tartozó információt felhasználó alkalmazást alaposan teszteljünk le a célkészülékeken, hiszen nem garantált, hogy az összes információ lekérdezhető.

Nézzünk további példákat néhány telefon- és hálózatspecifikus adat lekérdezésére a teljesség igénye nélkül:

```
TelephonyManager tm = (TelephonyManager)
getSystemService(
    TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
String IMSI = tm.getSubscriberId();
// ország azonosító hálózat alapján
String networkCountryISO = tm.getNetworkCountryIso();
String operator = tm.getNetworkOperatorName();
String simID = tm.getSimSerialNumber();
```

A készülék- és hálózatspecifikus adatok lekérdezéséhez a *READ_PHONE_STATE* engedélyre lesz szükségünk.

Ezek az adatok sok egyéb olyan járulékos információt is tartalmazhatnak, amelyekre szükségünk lehet az alkalmazásfejlesztés során. Például az IMEI-szám első nyolc karaktere a TAC- (Type Allocation Code) érték, amely egyértel-

műen azonosítja a telefon típusát. A *networkCountryISO* érték lekérdezésével például megállapíthatjuk, hogy a készülék éppen milyen országban tartózkodik, és ezt az információt akár közösségi alkalmazásokban is felhasználhatjuk.

Hasonlóan fontos függvények a *TelephonyManager* *getNetworkType()* és a *getPhoneType()* függvényei, amelyekkel megállapíthatjuk a hálózat típusát (2G, 2.5G, 3G stb.), illetve a készülék típusát is, hogy az európai GSM vagy az amerikai CDMA hálózatnak felel-e meg.

A *getNetworkType()* függvény lehetséges visszatérési értékei a következők:

- *TelephonyManager.NETWORK_TYPE_UNKNOWN*,
- *TelephonyManager.NETWORK_TYPE_GPRS*,
- *TelephonyManager.NETWORK_TYPE_EDGE*,
- *TelephonyManager.NETWORK_TYPE_UMTS*.

A *getPhoneType()* lehetséges visszatérési értékei a következők:

- *TelephonyManager.PHONE_TYPE_NONE*,
- *TelephonyManager.PHONE_TYPE_GSM*,
- *TelephonyManager.PHONE_TYPE_CDMA*.

A *TelephonyManager* lehetővé teszi, hogy különféle hálózati eseményekre is feliratkozzunk, és azokról értesítést kapjunk. Ehhez a *listen(listener,event)* függvényt kell használnunk, amelynek első paraméterében azt az objektumot kell megadni, amely a *PhoneStateListener* osztályból származik le, míg a második paraméterében azt az eseményt kell megadnunk, amelyre fel szeretnénk iratkozni. Ha több eseményre szeretnénk feliratkozni, a konstans értékeket vagy ('|') szimbólummal összekötve adhatjuk meg (bitenkénti vagy).

A támogatott események a következők:

- *PhoneStateListener.LISTEN_SIGNAL_STRENGTHS*: jelerősség, jel-zaj arány,
- *PhoneStateListener.LISTEN_DATA_ACTIVITY*: adatforgalom iránya,
- *PhoneStateListener.LISTEN_CELL_LOCATION*: cellaváltás,
- *PhoneStateListener.LISTEN_CALL_STATE*: hívásállapot,
- *PhoneStateListener.LISTEN_CALL_FORWARDING_INDICATOR*: hívásátirányítás állapota,
- *PhoneStateListener.LISTEN_DATA_CONNECTION_STATE*: adatforgalom állapota,
- *PhoneStateListener.LISTEN_MESSAGE_WAITING_INDICATOR*: üzenetvárakoztatás állapota,
- *PhoneStateListener.LISTEN_SERVICE_STATE*: hálózati szolgáltatás állapota.

Minden eseménynek van egy megfelelő callback függvénye, amely meghívódik az adott esemény bekövetkezésekor, és paraméterként megkapja az eseményhez kapcsolódó információkat. Nézzünk meg erre egy példát, ahol a jelerősséget monitorozzuk.

Elsőként szükségünk lesz egy osztályra, amely a *PhoneStateListener*-ből származik le, és implementálja a szükséges callback függvényt:

```
private class MyPhoneStateListener extends
PhoneStateListener{
    public void onSignalStrengthsChanged(
        SignalStrength signalStrength) {
        super.onSignalStrengthsChanged(signalStrength);
        Toast.makeText(getApplicationContext(),
            "GSM Cínr = "+String.valueOf(
                signalStrength.getGsmSignalStrength()),
            Toast.LENGTH_SHORT).show();
    }
};
```

Látható, hogy jelerősség-változáskor egy *Toast*-ban megtörténik az új érték kiírása. A következő feladatunk az eseményre való feliratkozás:

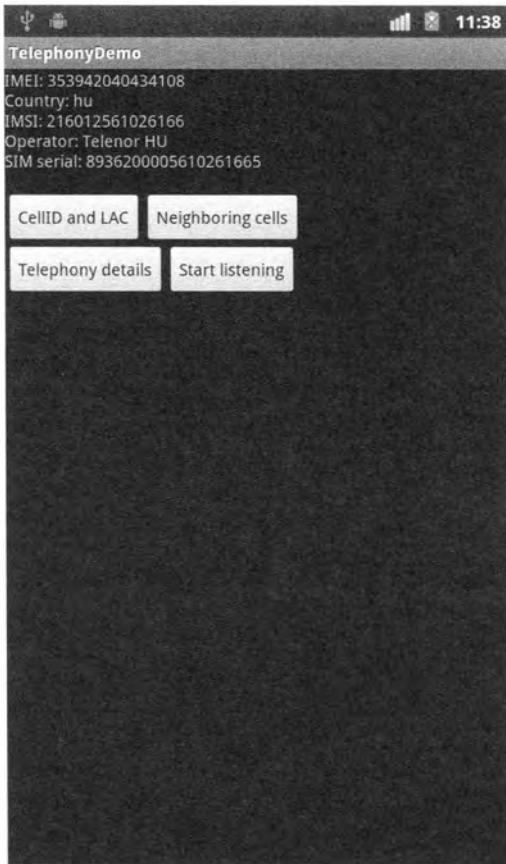
```
MyPhoneStateListener myListener = new
MyPhoneStateListener();
TelephonyManager tm = (TelephonyManager)
    getSystemService(Context.TELEPHONY_SERVICE);
tm.listen(myListener,
    PhoneStateListener.LISTEN_SIGNAL_STRENGTHS);
```

Végül le kell iratkoznunk az eseményről, ehhez szintén a *TelephonyManager* *listen()* függvényét kell használnunk. Első paraméterként a korábban használt *PhoneStateListener*-ből leszármazó objektumot adjuk meg, második paraméterként pedig a *PhoneStateListener.LISTEN_NONE* konstanst. Ezt például az Activity *onStop()* függvényében gyakran megteesszük.

```
if (myPhoneStateListener != null && tm != null)
    tm.listen(myPhoneStateListener,
        PhoneStateListener.LISTEN_NONE);
```

Az eddigiek összefoglalásaként vegyünk egy egyszerű példát, ahol négy gombot helyezünk el, amelyekkel a CellaID-t, a szomszédos cellainformációkat, a telefon- és hálózata adatokat tudjuk lekérdezni, valamint fel tudunk iratkozni a jelerősség- és az adatforgalom-irányváltozási eseményekre. A példánkban

a lekérdezések és a változások értékeit egy egyszerű *TextView*-n jelenítjük meg. A megoldás az eddig bemutatott elemekből építkezik, ezért a forráskódot nem részletezzük külön.



7.1. ábra. Hálózati és cellainformációk megjelenítése

Az alkalmazás felhasználói felületét a következő XML-erőforrás valósítja meg:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/tvStatus"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/tvStatus" />
```

```

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <Button android:id="@+id/btnCIDLAC"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnCIDLAC" />
    <Button android:id="@+id/btnNCIDs"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnNCIDs" />
</LinearLayout>
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <Button android:id="@+id/btnTelephony"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnTelephony" />
    <Button android:id="@+id/btnStartListening"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/btnStartListening"
    />
</LinearLayout>
</LinearLayout>

```

A felhasználói felülettel kapcsolatban figyeljük meg, hogyan ágyaztunk egymásba több *Layout*-ot.

Az alkalmazást megvalósító Activity a következő:

```

public class TelephonyDemoActivity extends Activity {
    private class MyPhoneStateListener extends
        PhoneStateListener {
        public void onSignalStrengthsChanged(
            SignalStrength signalStrength) {
            super.onSignalStrengthsChanged(signalStrength);
            tvStatus.setText("GSM Csinr = " +
                String.valueOf(
                    signalStrength.getGsmSignalStrength()));
        }
    }

    @Override
    public void onDataActivity(int direction) {

```

```

        super.onDataActivity(direction);
        tvStatus.setText("Data activity direction = "+
            direction);
    }
};

private TelephonyManager tm;
private MyPhoneStateListener myPhoneStateListener =
null;
private TextView tvStatus;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    tm = (TelephonyManager) getSystemService(
        TELEPHONY_SERVICE);
    tvStatus = (TextView) findViewById(R.id.tvStatus);
    Button btnCIDLAC = (Button) findViewById(
        R.id.btnCIDLAC);
    btnCIDLAC.setOnClickListener(new OnClickListener()
    {
        public void onClick(View v) {
            showCIDLAC();
        }
    });
    Button btnNCIDs = (Button) findViewById(
        R.id.btnNCIDs);
    btnNCIDs.setOnClickListener(new OnClickListener()
    {
        public void onClick(View v) {
            showNeightBoringCells();
        }
    });
    Button btnTelephony = (Button) findViewById(
        R.id.btnTelephony);
    btnTelephony.setOnClickListener(new
    OnClickListener() {
        public void onClick(View v) {
            showTelephonyInfo();
        }
    });

    Button btnStartListening = (Button) findViewById(
        R.id.btnStartListening);
    btnStartListening.setOnClickListener(
    new OnClickListener() {
        public void onClick(View v) {

```

```

        startListening();
    }
});
}

@Override
protected void onStop() {
    super.onStop();
    if (myPhoneStateListener != null && tm != null)
        tm.listen(myPhoneStateListener,
            PhoneStateListener.LISTEN_NONE);
}

protected void startListening() {
    MyPhoneStateListener myListener =
        new MyPhoneStateListener();
    tm.listen(myListener,
        PhoneStateListener.LISTEN_DATA_ACTIVITY |
        PhoneStateListener.LISTEN_SIGNAL_STRENGTHS);
}

private void showCIDLAC() {
    GsmCellLocation loc =
        (GsmCellLocation) tm.getCellLocation();
    int cellId = loc.getCid();
    int lac = loc.getLac();
    tvStatus.setText("LAC: " + lac + ", " +
        " CellID: " + cellId);
}

private void showNeightBoringCells() {
    List<NeighboringCellInfo> cellinfo =
        tm.getNeighboringCellInfo();
    StringBuilder sb = new StringBuilder();
    for (NeighboringCellInfo info : cellinfo) {
        int cellId = info.getCid();
        int rssi = info.getRssi();
        sb.append("CellID: ");
        sb.append(cellId);
        sb.append(", ");
        sb.append("RSSI: ");
        sb.append(rssi);
        sb.append("\n");
    }
    tvStatus.setText(sb.toString());
}

private void showTelephonyInfo() {

```

```

        StringBuilder sb = new StringBuilder();
        sb.append("IMEI: ");
        sb.append(tm.getDeviceId());
        sb.append("\n");
        sb.append("Country: ");
        sb.append(tm.getNetworkCountryIso());
        sb.append("\n");
        sb.append("IMSI: ");
        sb.append(tm.getSubscriberId());
        sb.append("\n");
        sb.append("Operator: ");
        sb.append(tm.getNetworkOperatorName());
        sb.append("\n");
        sb.append("SIM serial: ");
        sb.append(tm.getSimSerialNumber());
        sb.append("\n");
        tvStatus.setText(sb.toString());
    }
}

```

7.3. Pozíciókezelés Android platformon

Az Android kétféle lehetőséget ad a fejlesztők kezébe a pozíció meghatározására. Az egyik a GPS-alapú helymeghatározás, a másik pedig a hálózatalapú, amely magában foglalja a cellaalapú és a wifialapú helymeghatározást. Mindkét módszer számos előnyt és hátrányt jelent, ezekkel tisztában kell lennünk az alkalmazások fejlesztésekor. A GPS-alapú helymeghatározás például az egyik legpontosabb technológia, ám komoly hátránya, hogy csak kültéren működik. Ezzel szemben a hálózatalapú helymeghatározás ugyan pontatlan, de gyorsan talál közelítőleges pozíciót, amely sokszor elegendő.

A következőkben megvizsgáljuk, hogyan használhatók a különféle helymeghatározási technológiák Android platformon, és hogyan használhatók ki az egyes technológiák előnyei.

7.3.1. Pozíciómeghatározás

Android platformon a pozíció meghatározásához a *LocationManager* objektumra lesz szükségünk, amelyet ugyancsak a *getSystemService()* függvénnyel tudunk elérni, mint rendszerszolgáltatást, például egy Activity-n belül:

```

LocationManager locationManager =
    (LocationManager) getSystemService(Context.LOCATION_
SERVICE);

```

Miután megszereztük a referenciát a *LocationManager* objektumra, a *requestLocationUpdates()* függvény segítségével elindíthatjuk a pozíció folyamatos monitorozását. Ezt követően minden új pozícióértékről és pozíciómeghatározáshoz kapcsolódó eseményről a rendszer egy interfészen keresztül tájékoztat.

```
// szolgáltató, Pozíciókezelés Android platformon
minimális idő, minimális távolság, listener
locationManager.requestLocationUpdates(
    LocationManager.NETWORK_PROVIDER, 0, 0,
    locationManager);
```

A *requestLocation Updates()* függvény paraméterei a következők:

- providertípus: GPS- vagy hálózatalapú, de egyszerre mindkettőt is használhatjuk,
- minimumidő két frissítés között (0 – a lehető leggyakrabban),
- minimumtávolság két frissítés között (0 – a lehető leggyakrabban),
- *LocationListener* implementáció.

A függvény negyedik paramétere tehát egy olyan objektum, amely implementálja a *LocationListener* interfészt. Ez az interfész négy absztrakt függvény megvalósítását teszi kötelezővé, amelyeken keresztül értesítést kapunk az új pozícióértékről és egyéb, helymeghatározással kapcsolatos eseményekről.

A négy függvény a következő:

- *onLocationChanged(Location location)*: új pozícióérték;
- *onStatusChanged(String provider, int status, Bundle extras)*: állapotváltozás a helymeghatározás során, például a GPS-vétel megszűnt egy alagút miatt;
- *onProviderEnabled(String provider)*: a paraméterül kapott helymeghatározó módot engedélyezték;
- *onProviderDisabled(String provider)*: a paraméterül kapott helymeghatározó módot letiltották (például a felhasználó kikapcsolta a GPS-t).

Pozíciómeghatározáshoz a *manifest* állományban két engedélyre lehet szükségünk:

- *android.permission.ACCESS_COARSE_LOCATION*: csak hálózatalapú helymeghatározás,
- *android.permission.ACCESS_FINE_LOCATION*: GPS- és hálózatalapú helymeghatározási engedély.

Ha már nincs szükségünk további helymeghatározásra, a *LocationManager* objektumon keresztül leállíthatjuk a pozíciófigyelést a *removeUpdates()* függvény segítségével, amelynek meg kell adni az általunk használt, *LocationListener* interfészt implementáló objektumot.

```
locationManager.removeUpdates(locationListener);
```

Nagyon fontos, hogy egy helymeghatározást használó alkalmazásnál sose feledkezzünk meg a helymeghatározás leállításáról, hiszen ha nem tartjuk feleslegesen bekapcsolva a helymeghatározásra használt eszközöket, rengeteg energiát takaríthatunk meg. Ha például csak egy Activityben használjuk a helymeghatározást, az *onStop()* függvény ideális hely a pozíciómeghatározás leállítására.

Nézzünk meg néhány további javaslatot a helymeghatározáshoz:

- Ellenőrizzük, hogy a kapott pozíció jelentősen újabb-e, mint amelyről korábban tudtunk.
- Ellenőrizzük a kapott pozíció pontosságát (accuracy).
- Az előző pozícióból és a hozzátartozó sebességből ellenőrizhető az új pozíció realitása.
- Ellenőrizzük melyik technológiától (GPS vagy hálózatalapú) származik az új pozíció.
- 60 másodpercnél gyakoribb pozíciókérés sokszor felesleges.
- Általában nem kell GPS-t és hálózati módszert egyszerre használni, így ezt kerüljük.

Gyakorlásként nézzünk meg egy egyszerű alkalmazást, amely az aktuális pozíciót folyamatosan megjeleníti a kijelzőn. A pozíció folyamatos figyelése egy *Start* gomb lenyomása után kezdődik.

A felhasználói felület forrása a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <Button
        android:id="@+id/btnStart"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnStart" />
```

```

        <TextView
            android:id="@+id/tvStatus"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"/>
    </LinearLayout>

```

Az alkalmazás forráskódja az alábbi:

```

public class MyLocationActivity extends Activity
    implements LocationListener {
    private TextView tvStatus;
    private LocationManager lm;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        tvStatus = (TextView) findViewById(R.id.tvStatus);
        Button btnStart =
            (Button) findViewById(R.id.btnStart);
        btnStart.setOnClickListener(new OnClickListener()
        {
            public void onClick(View arg0) {
                startLocationUpdate();
            }
        });
    }

    @Override
    protected void onStop() {
        super.onStop();
        if (lm != null) {
            lm.removeUpdates(this);
        }
    }

    protected void startLocationUpdate() {
        LocationManager lm = (LocationManager)
            getSystemService(Context.LOCATION_SERVICE);
        lm.requestLocationUpdates(
            LocationManager.GPS_PROVIDER, 0, 0, this);
        lm.requestLocationUpdates(
            LocationManager.NETWORK_PROVIDER, 0, 0,
this);
    }

```

```
public void onLocationChanged(Location location) {
    if (location != null) {
        tvStatus.append(
            new Date(location.getTime()).
toString()+", " +
            location.getLatitude() + ", "
            + location.getLongitude()+"\n");
    }
}

public void onProviderDisabled(String provider) {
    // TODO esemény jelzése
}

public void onProviderEnabled(String provider) {
    // TODO esemény jelzése
}

public void onStatusChanged(String provider,
    int status, Bundle extras) {
    // TODO esemény jelzése
}
}
```

A fenti példában egyszerre alkalmaztunk GPS- és hálózatalapú helymeghatározást. Az elkészült alkalmazást a következő ábra szemlélteti.



7.2. ábra. Pozíció folyamatos kijelzése

Ha helymeghatározási alapú alkalmazást fejlesztünk, az emulátor is rendkívül jól használható, hiszen akár a DDMS-nézetten, akár telneten keresztül is beállíthatjuk, hogy milyen pozíción legyen aktuálisan az eszköz, valamint egy időbélyeg-alapú útvonalat is megadhatunk a készüléknek.

Telnet használatakor az alábbi kóddal állíthatjuk be az emulátor által használt pozíciót (NMEA-formátum¹⁹ is megadható):

```
telnet localhost <console-port> (pl. 5554)
geo fix 19.134 47.51119 210 (lat/lng/altitude)
```

¹⁹ NMEA-formátum: http://en.wikipedia.org/wiki/NMEA_0183

A *requestLocationUpdates()* első paramétere a helymeghatározásra használandó eszköz (pl. GPS vagy hálózat). Ha nem tudjuk pontosan, hogy melyikre van szükségünk, egy *Criteria* objektum definiálásával a *getBestProvider()* függvényen keresztül kérhetjük a rendszert, hogy állapítsa meg a feltételeknek leginkább megfelelőt.

A következő kódrészben például egy alacsony energiaigényű, magassági és sebességértékeket nem használó, adatforgalmat engedélyező technológiára kérünk ajánlást:

```
Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_COARSE);
criteria.setPowerRequirement(Criteria.POWER_LOW);
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setSpeedRequired(false);
criteria.setCostAllowed(true);
// második paraméter: csak az engedélyezett technológiák
String bestProvider =
    locationManager.getBestProvider(criteria, true);
```

Sokszor nincs szükségünk folyamatos pozíciófrissítésre, elegendő csupán egy érték. Ilyenkor a *LocationManager getLastKnownLocation()* függvényével lekérdezhethetjük a készülék által utoljára ismert pozíciót *Location* objektum formájában. Az objektum egyben tartalmazza a rögzítés időbélyegét is, így könnyedén eldönthetjük, hogy megfelelő-e számunkra, vagy muszáj új frissítést indítanunk.

```
LocationManager locationManager = (LocationManager)
    getSystemService(Context.LOCATION_SERVICE);
Location location =
    locationManager.getLastKnownLocation(provider);
if (location != null) {
    double lat = location.getLatitude();
    double lng = location.getLongitude();
}
```

7.3.2. Közelségi riasztások kezelése

Helyalapú alkalmazások esetén gyakran szükség van olyan funkcionalitásra, amikor valamilyen reakciót kell kiváltanunk, ha egy bizonyos helyszínt megközelítettünk, vagy eltávolodtunk tőle egy meghatározott mértékben.

Ilyenkor használhatnánk az előzőleg bemutatott módszert, és manuálisan megpróbálhatnánk megállapítani a megközelítés és az eltávolítás eseményét, ám ez nagy pazarlás lenne. Helyette az Android egy úgynevezett *ProximityAlert* mechanizmust biztosít. A megoldás lényege az, hogy definiálnunk kell egy *PendingIntent* (előkészített szándék) objektumot, amelynek a kibocsátása az esemény bekövetkezésekor jön létre. A *ProximityAlert* mechanizmus előnye az, hogy intelligensen választja ki a helymeghatározási technológiákat, és optimálisan bánt az erőforrásokkal.

A *ProximityAlert* használatához meg kell adnunk egy koordinátát, illetve egy hozzátartozó sugarat, valamint egy olyan időintervallumot, ameddig az adott esemény értékes lehet a számunkra (-1 érték esetén korlátlan ideig figyel a rendszer).

Az emulátoron való tesztelés előtt ügyeljünk arra, hogy ha közelségriasztás-eseményre iratkoztunk fel, akkor, ha az emulátor számára elsőként küldött koordináta a sugáron belül van, nem jelez a rendszer, csak ha valóban „megközelítési” esemény történt.

A következőkben nézzünk meg egy egyszerű példát a *ProximityAlert* használatára. A közelségi esemény elküldése egy *Intent* formájában történik meg, ezt legegyszerűbben egy *BroadcastReceiver*-rel tudjuk érzékelni. Ehhez valósítsunk meg egy megfelelő osztályt:

```
public class ProximityIntentReceiver
    extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent
    intent) {
        String key = LocationManager.KEY_PROXIMITY_ENTER-
    ING;
        // alapértelmezetten false érték
        Boolean entering =
            intent.getBooleanExtra(key, false);
        textViewStatus.append("\nEntering: "+entering);
    }
}
```

Az alábbi függvény segítségével fel tudunk iratkozni *ProximityAlert*-re:

```
private void setProximityAlert(double lat, double lon,
    float radius, long expiration, int requestCode){
    LocationManager locationManager =
        (LocationManager) this.getSystemService(
            Context.LOCATION_SERVICE);
    Intent intent = new Intent(PROX_ALERT_INTENT);
    PendingIntent pendingIntent =
        PendingIntent.getBroadcast(
```

```
        getApplicationContext(), requestCode, intent,
        PendingIntent.FLAG_CANCEL_CURRENT);
    locationManager.addProximityAlert(
        lat, lon, radius, expiration, pendingIntent);
}
```

Ezt követően szükségünk lesz egy *ProximityIntentReceiver* osztálypéldányra, valamint egy egyedi azonosítóra a *PendingIntent* számára.

Végül az előbbi függvény segítségével az eseményre való feliratkozás a következőképpen történhet:

```
// szükséges tagváltozók a felhasználó osztályban
private static final String PROX_ALERT_INTENT =
    "hu.bute.daa1.amorg.examples.PROXIMITY_ALERT";
private ProximityIntentReceiver pxr =
    new ProximityIntentReceiver();
// ...
// BroadcastReceiver beállítása és
// ProximityAlert inicializálása
IntentFilter intentFilter =
    new IntentFilter(PROX_ALERT_INTENT);
registerReceiver(pxr, intentFilter);
setProximityAlert(47.001, 19.001, -1, 0);
```

Ha nincs már szükség a *ProximityAlert*re, a következő két dolgot kell tennünk a leállításához:

- *LocationManager removeProximityAlert()* függvényének meghívása a releváns *PendingIntent* objektummal,
- *BroadcastReceiver*-ről való leiratkozás.

Soha sem feledkezzünk el a *ProximityAlert*ek leállításáról, hiszen ezzel rengeteg erőforrást takaríthatunk meg. Továbbá vegyük figyelembe, hogy egy alkalmazásban több *ProximityAlert*et is használhatunk, így azokat egyenként kell leállítani.

7.3.3. Átalakítás földrajzi koordináta és postacím között

Korábbi példáinkban csak földrajzi koordinátákkal dolgoztunk, az Android azonban kifinomult módszert biztosít a földrajzi koordináta és a postacím közötti átváltásra. A postacímről földrajzi koordinátára való átalakítást a szakirodalomban *Geocoding*nek, a fordított irányt (koordinátából postacím) *Reverse Geocoding*nek nevezik.

Android-környezetben ennek megvalósítására a *Geocoder* osztály használatos. Az átalakítás szervertámogatással történik, így használatához mindenképpen szükség van az internetengedély (*android.permission.INTERNET*) jelzésére a *manifest* állományban.

A következőkben nézzünk meg egy egyszerű példát a Geocodingra. A cím szöveges értéke lehet irányítószám, város, utca, házszám és esetleg középület (múzeum, vasútállomás stb.). A válasz egy *Address* lista formájában érkezik meg, amelynek maximalizálni tudjuk a méretét egy megfelelő paraméter megadásával.

```
// Locale.ENGLISH: Angol formátumban adjuk meg a címet
Geocoder geocoder = new Geocoder(this, Locale.ENGLISH);
String streetAddress = "Magyar tudósok körútja 2, Budapest";
List<Address> locations = null;
// maximum 5 lehetőség érdekel bennünket
int maxNumOfResults = 5;
locations = geocoder.getFromLocationName(streetAddress,
    maxNumOfResults);
```

A *Geocoder* objektumnak létezik egy másik függvénye is, amellyel egy adott területen belül kereshetünk, ha megadjuk a területet határoló téglalap bal alsó és jobb felső koordinátáját.

```
getFromLocationName (String locationName, int
    maxResults,
    double lowerLeftLatitude, double lowerLeftLongitude,
    double upperRightLatitude, double
    upperRightLongitude)
```

Nézzünk meg egy olyan példát, ahol földrajzi koordinátaérték alapján kérdezzük le a postacímet.

```
double latitude = 47.0;
double longitude = 19.0;
Geocoder gc = new Geocoder(this, Locale.getDefault());
List<Address> addresses = null;
// maximum 5 lehetőség érdekel bennünket
int maxNumOfResults = 5;
addresses = gc.getFromLocation(latitude, longitude,
    maxNumOfResults);
```

Mindkét esetben figyeljük meg, hogy a postacím nyelvi megjelenését a *Locale* osztály használatával tudjuk megadni.

7.4. Térképnézet

Végül, de nem utolsósorban vizsgáljuk meg a térképnézet használatát. Az Android platform egy nagyon jól átgondolt, *Google Maps*-alapú térképnézetet biztosít a fejlesztőknek, amellyel könnyedén integrálhatunk térképalapú megoldásokat az alkalmazásunkba. A következőkben áttekintjük a térképnézet (*MapView*) nyújtotta lehetőségeket.

A térképnézetet megvalósító *MapView* komponens nem található meg az alap-Android-osztálykönyvtárban, használatához a Google APIs osztálykönyvtárat kell importálnunk (*com.google.android.maps*). Ha *MapView*-t szeretnénk használni az alkalmazásunkban, akkor az Android SDK segítségével a *Google APIs* könyvtár megfelelő verzióját is le kell töltenünk, és egy ezzel kompatibilis emulátort (AVD) kell létrehoznunk. Mielőtt a térképnézetet használnánk, szükségünk van még egy térképkulcsra is, amelyet a Google hivatalos oldaláról²⁰ tudunk beszerezni. A későbbiekben, ha ki szeretnénk adni az alkalmazást, ezt alá kell írunk, és az aláíráshoz külön Maps API key-t kell igényelnünk.

A *MapView* egy olyan Android-UI-komponens, amellyel térképet tudunk megjeleníteni. A *MapView* használatához nem standard Android-projektet kell létrehozni, hanem a *Google APIs*-t kell *Build Target*ként kiválasztani. Ezt követően a *manifest* állományban jelezni kell, hogy használjuk a *Maps API*-t:

```
<uses-library android:name="com.google.android.maps" />
```

A térkép használatához internetelérésre is szükség van, amelyre az engedélyt szintén a *manifest* állományban kell jeleznünk az *</application>* záró-Tagen kívül.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Ekkor már használhatjuk is a *MapView* komponenst akár erőforrásból is, például:

```
<com.google.android.maps.MapView
    xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:id="@+id/mapview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:clickable="true"
    android:apiKey="0DmytYWSGb87LXU1UaecgEDbZ1Ph97FhTubmrlg"
/>
```

²⁰ Google Maps API Key Android-regisztráció: <http://code.google.com/intl/hu-HU/android/maps-api-signup.html>

Az erőforrásban az *android:apiKey* attribútumnak kell megadnunk a regisztráció után kapott kulcsot, tehát a fejlesztés során a fenti kulccsal nem, csak a magunk által regisztrált kulccsal jelenik meg helyesen a térképnézet. A *MapView* térképet jelenít meg, ám bármilyen elemet, illetve bármilyen objektumokat elhelyezhetünk rá, úgynevezett *Overlay*-eket is rá tud rajzolni a térképre. Tipikusan különféle jelölőket vagy útvonalakat szokás a térképre rajzolni. Egy *MapView* több *Overlay*-t is tartalmazhat.

A *MapView* használatának elsajátításához készítsünk egy olyan alkalmazást, amely egy térképet jelenít meg, és egy gomb segítségével a műholdnézetet ki-be kapcsolhatjuk.

Miután sikeresen megigényeltük a kulcsot, hozzunk létre egy új Android-projektet úgy, hogy a *build target Google APIs* legyen (7-es verzióval), és a *manifest* állományban jelezzük a *Google Maps API* használatát.

```
<uses-library android:name="com.google.android.maps" />
```

Ezt követően a *manifest* állományba vegyük fel a szükséges engedélyeket az *</application>* záró-Tag után (ezekre a fejlesztés során mind szükség lesz):

```
<uses-permission android:name="android.permission.INTERNET" />
```

Majd a felhasználói felületet leíró XML-t alakítsuk át a következőre, továbbá az *apiKey* értéket állítsuk be arra, amit kaptunk a regisztrációkor:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/mainlayout"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <ToggleButton
        android:id="@+id/toggleButtonSatellite"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textOn="@string/tbSatelliteOn"
        android:textOff="@string/tbSatelliteOff" />
    <com.google.android.maps.MapView
        android:id="@+id/mapview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:clickable="true"
```

```

        android:apiKey=
            "0DmytYWSGb87LXU1UaecgEDbZ1Ph97FhTubmrlg"
    />
</LinearLayout>

```

Az alkalmazáslogika megvalósítását egy Activity segítségével hozzuk létre, térkép alkalmazásakor azonban nem Activity, hanem *MapActivity* osztályból kell leszármaznunk.

```

public class MapDemoActivity extends MapActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // térkép beállítások és középpont meghatározása
        final MapView mapView =
            (MapView) findViewById(R.id.mapview);
        mapView.setBuiltInZoomControls(true);
        MapController mapController = mapView.
getController();
        mapController.setCenter(
            new GeoPoint(47000000, 19000000));

        // gomb eseménykezelő megvalósítása
        final ToggleButton tbSatellite = (ToggleButton)
            findViewById(R.id.toggleButtonSatellite);
        tbSatellite.setOnClickListener(new
OnClickListener() {
            public void onClick(View v) {
                if (tbSatellite.isChecked())
                    mapView.setSatellite(true);
                else
                    mapView.setSatellite(false);
            }
        });
    }

    @Override
    protected boolean isRouteDisplayed() {
        return false;
    }
}

```

A *MapActivity* osztályból való leszármaztatás miatt az *isRouteDisplayed()* függvényt kötelező felüldefiniálni. A függvény visszatérési értékeként meg kell adnunk, hogy a térképen megjelenítünk-e útvonalat. Ügyeljünk erre, mivel *true*-val való visszatérés esetén más üzleti szabályzat vonatkozik az alkalmazásunkra.

Az Activity *onCreate()* függvényében elsőként elkérjük a *MapView* referenciát, amelynek *setBuiltInZoomControls()* függvényével megjelenítjük a beépített közelítő/távolító vezérlőket. A *MapView* további néhány érdekes függvénye a következő:

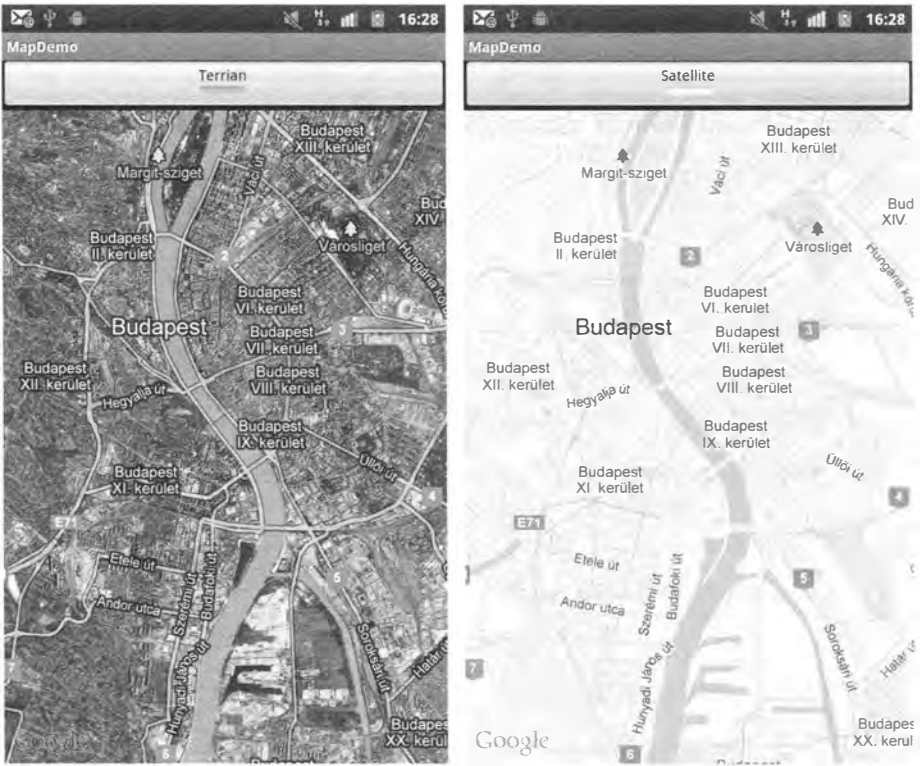
- *setStreetView(boolean streetView)*: utcanézet bekapcsolása,
- *setSatellite(boolean satellite)*: műholdnézet engedélyezése,
- *setTraffic(boolean traffic)*: forgalmi adatok megjelenítése,
- *preLoad()*: középpont körüli térképrész előre letöltése.

Ezt követően a térkép vezérléséhez a *MapView* objektumtól elkérjük a *MapController* objektumot. Néhány további függvény, amelyet a *MapController* támogat:

- *setCenter(GeoPoint center)*: középpont beállítása,
- *animateTo(GeoPoint to)*: adott koordinátára való navigálás,
- *stopAnimation(boolean jumpToFinish)*: koordinátára való mozgás leállítása,
- *zoomIn()*, *zoomOut()*: közelítés, távolítás,
- *zoomToSpan(int latSpanE6, int lonSpanE6)*: közelítés egy adott területre.

A fenti példában a *setCenter()* függvényben megadott térkép-középpont koordinátáit nem *double* értékként, hanem egy felszorozott egész értékként (*E6*) adjuk meg. Ennek az az oka, hogy egész számokkal a rendszer gyorsabban dolgozik.

Az elkészült alkalmazást a következő ábra szemlélteti.



7.3. ábra. Térképalkalmazás-példa

Ha bekapcsoljuk a forgalmi nézetet, az útvonalakra az aktuális forgalmi terhelés is megjelenik.



7.4. ábra. Forgalmi nézet

A következő példánkban egészítsük ki az alkalmazást úgy, hogy legyen lehetőség különféle pontokat megjeleníteni a térképen, amelyekre kattintva egy felugró ablakban az adott hely leírása jelenik meg.

Ennek megvalósításához létre kell hoznunk egy saját osztályt, amely az *ItemizedOverlay* osztályból származik le. Az *ItemizedOverlay* tehát már egy előre elkészített osztály, amely lehetővé teszi, hogy ikonokat helyezzünk el adott pozícióra *OverlayItem* elemek formájában. A következő osztály a leírt funkciót valósítja meg:

```
public class MyMarkers extends
ItemizedOverlay<OverlayItem> {
    private ArrayList<OverlayItem> mOverlays =
        new ArrayList<OverlayItem>();
    private Context mContext;

    public MyMarkers(Drawable defaultMarker,
        Context context) {
        super(boundCenterBottom(defaultMarker));
        mContext = context;
    }
}
```

```

    }

    public void addOverlay(OverlayItem overlay) {
        mOverlays.add(overlay);
        populate();
    }

    @Override
    protected OverlayItem createItem(int i) {
        return mOverlays.get(i);
    }

    @Override
    public int size() {
        return mOverlays.size();
    }

    @Override
    public boolean onTouchEvent(MotionEvent event,
        MapView mapView) {
        return false;
    }

    @Override
    protected boolean onTap(int index) {
        OverlayItem item = mOverlays.get(index);
        AlertDialog.Builder dialog =
            new AlertDialog.Builder(mContext);
        dialog.setTitle(item.getTitle());
        dialog.setMessage(item.getSnippet());
        dialog.show();
        return true;
    }
}

```

Az előző osztályban tehát az *addOverlay()* függvény segítségével tudunk új elemeket felvenni a pontjaink közé, illetve az *onTap(index)* függvény megvalósításával tudjuk kezelni azt, hogy a paraméterben megkapott indexű pontra kattintva mi történjen.

Új térképpontokat a fenti osztály segítségével a következőképpen tudunk felvenni egy létező *mapView*-ra:

```

MyMarkers myMarkers =
    new MyMarkers(getResources().getDrawable(
        R.drawable.ic_launcher), this);
myMarkers.addOverlay(new OverlayItem(

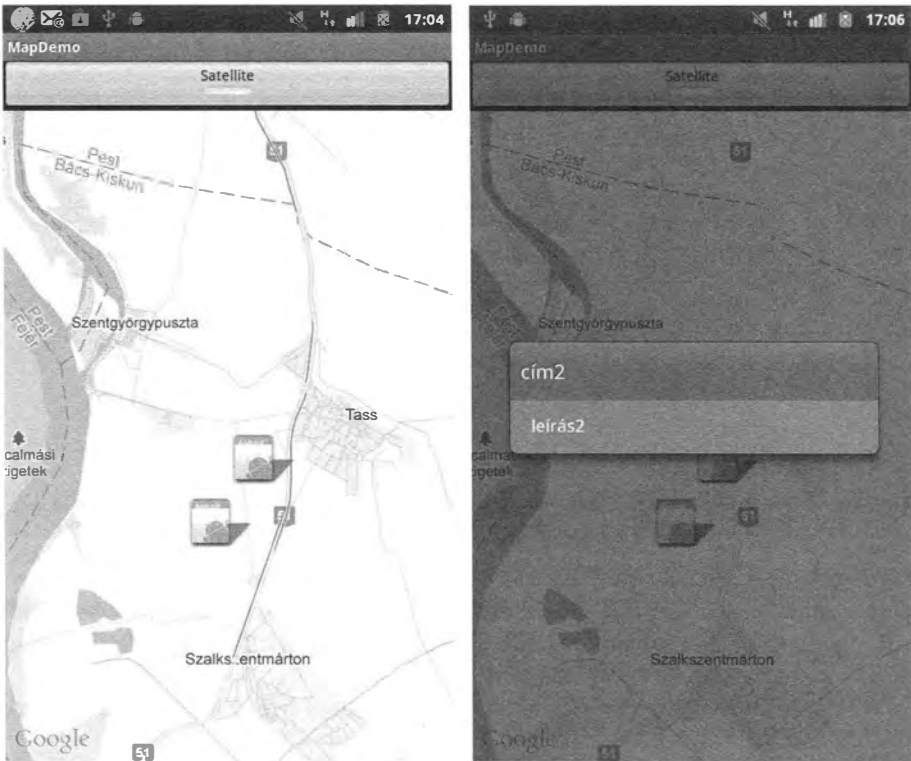
```

```

new GeoPoint(47000000, 19000000), "cím1", "le-
írás1"));
myMarkers.addOverlay(new OverlayItem(
    new GeoPoint(47010000, 19010000), "cím2", "le-
írás2"));
mapView.getOverlays().add(myMarkers);

```

Példánkban a szöveges értékeket nem erőforrásból, hanem közvetlenül töltjük be, valós körülmények között pedig mindig alkalmazunk erőforrásokat.



7.5. ábra. Jelölők elhelyezése a térképen

A *MapView* nézet lehetőséget biztosít saját *Overlay*-ek elhelyezésére is, amelynek teljes kirajzolási módját meghatározhatjuk. Ebben az esetben egy saját osztályt kell megvalósítanunk, amely az *Overlay* osztályból származik le. Ebben az osztályban a *draw()* függvény felüldefiniálásban adhatjuk meg, hogy milyen alakzatot kívánunk kirajzolni.

A következő példában egy olyan *Overlay*-t valósítunk meg, amely egy útvonalat jelenít meg:

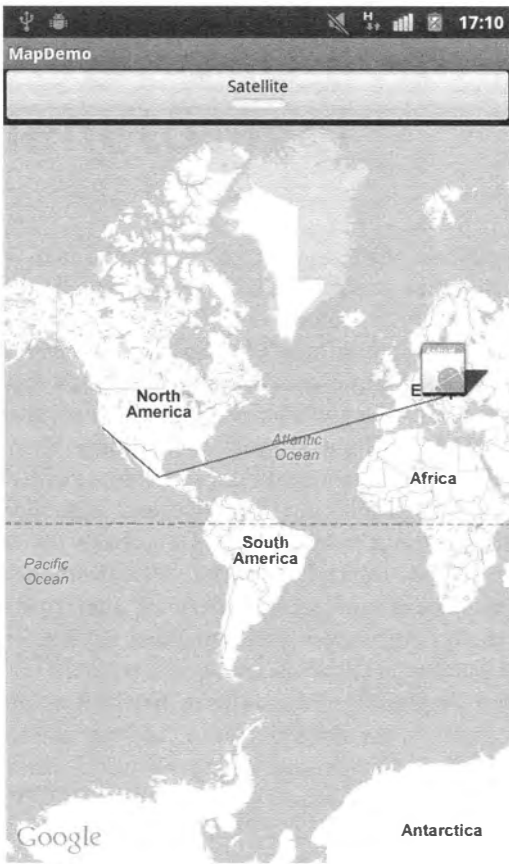
```
public class MyPath extends Overlay {
    public void draw(Canvas canvas, MapView mapv,
        boolean shadow) {
        super.draw(canvas, mapv, shadow);
        Paint mPaint = new Paint();
        mPaint.setDither(true);
        mPaint.setColor(Color.RED);
        mPaint.setStyle(Paint.Style.STROKE);
        mPaint.setStrokeWidth(2);
        GeoPoint gP1 = new GeoPoint(19240000, -99120000);
        GeoPoint gP2 = new GeoPoint(37423157, -122085008);
        GeoPoint gP3 = new GeoPoint(47000000, 19000000);
        Point p1 = new Point();
        Point p2 = new Point();
        Point p3 = new Point();
        Path path = new Path();
        Projection projection = mapv.getProjection();
        // leképezés koordináta és pixel között!
        projection.toPixels(gP1, p1);
        projection.toPixels(gP2, p2);
        projection.toPixels(gP3, p3);
        path.moveTo(p2.x, p2.y);
        path.lineTo(p1.x, p1.y);
        path.lineTo(p3.x, p3.y);
        canvas.drawPath(path, mPaint);
    }
}
```

Figyeljük meg, hogy a példában hogyan történik az átalakítás a földrajzi koordináta és a képpont között. Elsőként a paraméterül kapott *MapView*-tól kell a *Projection* (leképezés) objektumot elkérni, amelynek *toPixels()* függvényével tudjuk az átalakítást végrehajtani.

Az elkészült *Overlay* az alábbiak szerint illeszthető rá egy létező *MapView*-ra:

```
MyPath path = new MyPath();
mapView.getOverlays().add(path);
```

Az elkészült alkalmazást a következő ábra szemlélteti.



7.6. ábra. Útvonal kirajzolása

A hálózati kommunikáció lehetőségei

A mobil eszközök és az operációs rendszerek mellett napjainkban a mobilhálózatok is rohamos fejlődésen mennek keresztül. Ennek következtében a mobil-szélessáv egyre inkább elérhetővé válik a felhasználók számára a fix előfizetési díjaknak köszönhetően. Emellett az androidos készülékek tipikusan rendelkeznek wifikommunikációs interfésszel is, ennek következtében a nagymennyiségű adatletöltés sem okoz problémát.

Megfigyelhető, hogy a legújabb hálózati technológiák általában gyorsan bekerülnek az új mobil eszközökbe, így a hálózati kommunikáció megismerése fejlesztői szempontból mindenképpen fontos. Szerencsére az Android rendkívül gazdag API-támogatást nyújt, és nemcsak az adatok küldését, illetve fogadását teszi lehetővé, hanem nagyon nagy szabadságot kapunk a hálózat irányítása fölött is, így például lehetőségünk van a wifi ki/be kapcsolására, jelerősség lekérdezésére, hozzáférési pontok neveinek lekérdezésére stb.

Ebben a fejezetben elsőként áttekintjük a hálózati kapcsolatok felügyeletét és az Android nyújtotta lehetőségeket, majd bemutatjuk az értesítések használati módját, amelyre gyakran szükség van az aszinkron hálózati műveleteknél. Ezt követően ismertetjük a *WebView* komponenst, majd rátérünk a HTTP-kommunikáció használatára. Ezután az ismert kommunikációs formátumokat (JSON, XML) mutatjuk be, végül pedig a TCP/IP socket kommunikációt és a *Push*-típusú üzeneteket tekintjük át.

8.1. Hálózati kapcsolatok felügyelete

Hálózati kommunikációt igénylő alkalmazásoknál gyakran szükség lehet a hálózati eszközök, interfészek vezérlésére, felügyeletére. Ilyen funkciók megvalósítására használható a *ConnectivityManager*, amelyet a *CONNECTIVITY_SERVICE* rendszerszolgáltatással érhetünk el.

A *ConnectivityManager* feladatai a következők:

- hálózati kapcsolatok monitorozása (wifi, GPRS, UMTS stb.),
- *Broadcast Intent*ek küldése, ha változik a hálózat állapota,
- automatikus csatlakozás másik hálózatra, ha megszakad a kapcsolat,
- hálózati kapcsolatok részletes adatainak a lekérdezése.

A *ConnectivityManager* használatához tipikusan szükség van az `ACCESS_NETWORK_STATE` engedélyre.

Néhány fontos függvény, amelyet a *ConnectivityManager* biztosít, a következő:

- *NetworkInfo getActiveNetworkInfo()*: aktív hálózati kapcsolat információinak a lekérdezése,
- *startUsingNetworkFeature(int networkType, String feature)*: adott hálózati eszköz indítása,
- *stopUsingNetworkFeature(int networkType, String feature)*: adott hálózati eszköz leállítása.

A következőkben nézzünk meg egy egyszerű példafüggvényt, amely mobilhálózat használatakor kiírja a kapcsolódási pont nevét (APN – Access Point Name):

```
private void logNetworkInfo() {
    ConnectivityManager connectivityManager =
        (ConnectivityManager)this.getSystemService(
            Context.CONNECTIVITY_SERVICE);
    NetworkInfo mobile = connectivityManager.
        getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
    if (mobile != null && mobile.isAvailable() &&
        mobile.isConnected())
    {
        String operatorAPN = mobile.getExtraInfo();
        log.d("MYLOG", operatorAPN);
    }
}
```

A mobilhálózat mellett sokszor a wifi felügyeletére is szükség lehet, ehhez a *WifiManager* osztályt használhatjuk, amellyel lekérdezhetők az elmentett wifilérési pontok nevei, új wifipontokat kereshetünk és menthetünk el, megnyithatunk és leállíthatunk adott wifipontokat, valamint az egyes pontok jelérésségei is lekérdezhetők. A *WifiManager* használatához tipikusan az alábbi két engedélyre van szükség:

- *ACCESS_WIFI_STATE*: állapot lekérdezése,
- *CHANGE_WIFI_STATE*: állapot módosítása (például a wifi felderítéséhez is szükség van rá).

A készüléken elmentett wifihozzáférési pontok az alábbiak szerint listázhatók:

```
// WiFi szolgáltatás elkérése
WifiManager wifiManager = (WifiManager)
getSystemService(Context.WIFI_SERVICE);
// WiFi állapot lekérdezése
WifiInfo info = wifiManager.getConnectionInfo();
Log.d("WiFi", "WiFi status: " + info.toString());
// Elmentett WiFi hálózatok listázása
List<WifiConfiguration> configs = wifiManager.
getConfiguredNetworks();
for (WifiConfiguration config : configs) {
    Log.d("WiFi", "Saved WiFi: " + config.toString());
}
```

Következő lépésként készítsünk egy olyan alkalmazást, amelyben a látható wifihozzáférési pontokat listázzuk ki. A megoldáshoz *BroadcastReceiver* használatára lesz szükség.

```
public class WifiScanActivity extends Activity {

    private TextView tvStatus;
    private WifiManager wifiManager;

    private BroadcastReceiver wifiStateChangedReceiver =
        new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent
            intent) {
                for (ScanResult result : wifiManager.
                getScanResults()) {
                    tvStatus.append("\n" + result.SSID);
                }
            }
        };

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        tvStatus = (TextView) findViewById(R.id.tvStatus);
        wifiManager =
            (WifiManager) getSystemService(Context.WIFI_SER-
            VICE);
```

```

        Button btnStartScan =
            (Button) findViewById(R.id.btnStartScan);
        btnStartScan.setOnClickListener(new
        OnClickListener() {
            public void onClick(View arg0) {
                tvStatus.setText("");
                wifiManager.startScan();
            }
        });

        registerReceiver(wifiStateChangedReceiver,
            new IntentFilter(
                WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
    }

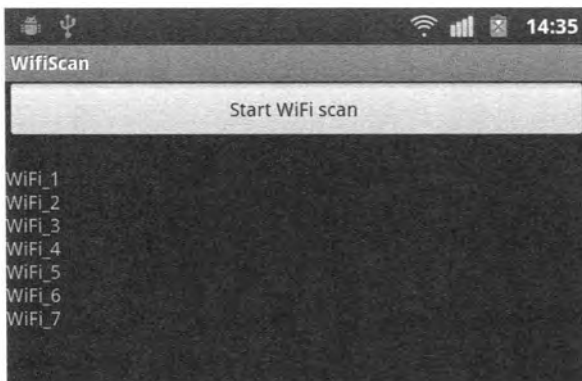
    @Override
    protected void onStop() {
        super.onStop();
        // Ne felejtsünk el leiratkozni!
        unregisterReceiver(wifiStateChangedReceiver);
    }
}

```

Elsőként a *WifiManager* szolgáltatást kértük el, majd a *btnStartScan* gomb hatására a *WifiManager startScan()* függvényével indítjuk el a felderítést. Az eredmények megjelenítése egy *BroadcastReceiver*-ben jön létre, mivel az *onCreate()* végén feliratkoztunk a *WifiManager.SCAN_RESULTS_AVAILABLE_ACTION* akcióra.

Az *onStop()* függvényben leiratkozunk a *Broadcast* eseményről, erről soha ne feledkezzünk el a későbbiekben sem.

Az alkalmazást futtatva a következő ábrán látható az eredmény.



8.1. ábra. *Wifi*hozzáférési pontok felderítése

A *WifiManager* további használatára nézzük meg a következő példát, amely egy *TextView*-n mutatja, ha a készülék wifieszközét be- vagy kikapcsoljuk. Tételezzük fel, hogy az XML-ben megadott felhasználói felületen már létezik egy *tvStatus* azonosítójú *TextView*.

```
public class ActivityWiFiObserver extends Activity {
    private TextView tvStatus;

    private BroadcastReceiver wifiStateChangedReceiver =
        new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                int extraWifiState =
                    intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE,
                        WifiManager.WIFI_STATE_UNKNOWN);
                switch (extraWifiState) {
                    case WifiManager.WIFI_STATE_DISABLED:
                        tvStatus.setText("WIFI STATE DISABLED");
                        break;
                    case WifiManager.WIFI_STATE_DISABLING:
                        tvStatus.setText("WIFI STATE DISABLING");
                        break;
                    case WifiManager.WIFI_STATE_ENABLED:
                        tvStatus.setText("WIFI STATE ENABLED");
                        break;
                    case WifiManager.WIFI_STATE_ENABLING:
                        tvStatus.setText("WIFI STATE ENABLING");
                        break;
                    case WifiManager.WIFI_STATE_UNKNOWN:
                        tvStatus.setText("WIFI STATE UNKNOWN");
                        break;
                }
            }
        };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        tvStatus = (TextView)findViewById(R.id.tvStatus);
        registerReceiver(wifiStateChangedReceiver,
            new IntentFilter(WifiManager.WIFI_STATE_CHANGED_ACTION));
    }

    @Override
    public void onStop(Bundle savedInstanceState) {
        unregisterReceiver(wifiStateChangedReceiver);
    }
}
```

A megoldáshoz hasonlóan *BroadCastReceiver*t használtunk. Az Android számos hálózati felügyeleti eszközt biztosít, amelyek közül csak néhányat mutatunk be, ám érdemes további példákat is megvizsgálni.

8.2. Értesítések megjelenítése

Az Android rendkívül hasznos módszert biztosít az értesítések megjelenítésére háttérfolyamatok vagy éppen hosszú ideig tartó hálózati kommunikáció befejezésének a jelzésekor. Az Android rendszer beépítve támogat egy úgynevezett *NotificationBar*t (értesítéssáv), amelyre a fontos események kellően figyelemfelkeltően, ugyanakkor nem zavaróan megjeleníthetők.

Egy ilyen értesítés nemcsak egy egyszerű üzenet kiírásából áll, hanem egyedi felületet is meghatározhatunk hozzá, illetve beállíthatjuk az üzenet kiválasztásakor futtatandó eseményt. Az értesítésekhez kapcsolható fő funkciók a következők:

- ikon megadása,
- részletes leírás,
- LED-ek felvillantása (ha van),
- rezgés,
- értesítési hang.

Az értesítésszolgáltatás Android platformon rendszerszolgáltatásnak számít, éppen ezért a használatához egy *NotificationManager* példányra lesz szükség, amelyet a rendszertől kell elkérnünk.

```
NotificationManager notificationManager =
    (NotificationManager) getSystemService(
        Context.NOTIFICATION_SERVICE);
```

Ezt követően össze kell állítanunk egy *Notification* objektumot, amelynek meg kell adni az ikonját, a címsorát és az időbélyegét. Részletesen a következőképpen néz ki:

```
int icon = R.drawable.icon;
String tickerText = "Címsor";
// Az extended status bar-on levő sorrend kialakításá-
hoz
long when = System.currentTimeMillis();
Notification notification = new Notification(
    icon, tickerText, when);
```


Ezt követően szükségünk lesz arra a kontextusra, amely az értesítést indítja, valamint megadhatunk részletes címet és leírást, illetve definiálhatunk egy *PendingIntent*-et (függő szándék), amelyet az értesítésre kattintva küld ki a rendszer.

```
Context context = getApplicationContext();
String expandedText = "Expanded text";
String expandedTitle = "Expanded title";
Intent intent = new Intent(this, MyActivity.class);
//statikus fv-nyel szerzünk PendingIntent-et
// (context, requestCode, intent, flags)
PendingIntent launchIntent =
    PendingIntent.getActivity(context, 0, intent, 0);
```

Ezután a *notification* objektumnak megadjuk a legutolsó értesítés paramétereit.

```
notification.setLatestEventInfo(context,
    expandedTitle, expandedText, launchIntent);
```

A *setLatestEventInfo()* függvénnyel újabb eseményeket is jelezhetünk, és megváltoztathatjuk az értesítés tartalmát.

Végül az értesítés megjelenítése a *NotificationManager notify()* függvényével történik. A *notify()* függvény első paramétere egy értesítésazonosító, míg a második paraméter a létrehozott *notification* objektum.

```
final int NOTIF_ID=1;
notificationManager.notify(NOTIF_ID, notification);
```

Ha az értesítést vissza szeretnénk vonni valamilyen okból (például mert elavult), akkor ezt a *NotificationManager cancel()* függvényével tehetjük meg, amelynek szintén át kell adni a *notification* objektum referenciáját:

```
notificationManager.cancel(notification);
```

Ha alapértelmezett hangot vagy rezgést szeretnénk beállítani az értesítéshez, akkor ezt a *notification* objektum *defaults* flagjéhez kell felvenni.

```
notification.defaults |= Notification.DEFAULT_SOUND;
notification.defaults |= Notification.DEFAULT_VIBRATE;
```

Egyedi rezgés is megadható.

```
long[] vibrate = new long[] { 1000, 1000, 1000, 1000,
1000 };
notification.vibrate = vibrate;
```

Rezgés használatakor fel kell venni az *android.permission.VIBRATE* engedélyt. Továbbá lehetőség van még egyéb *flaget* beállítani a *Notification*öknek. A folyamatban lévő esemény jelzése a következő:

```
notification.flags =
    notification.flags | Notification.FLAG_ONGOING_EVENT;
```

Kitartó események jelzése a következő:

```
notification.flags =
    notification.flags | Notification.FLAG_INSISTENT;
```

Példaként nézzünk meg egy *Activityt*, amely egy gombnyomás hatására egy értesítést jelenít meg:

```
public class ActivityNotificationExample extends
Activity {

    private static final int NOTIFICATION_ID = 1;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button btnShowNotification =
            (Button) findViewById(R.id.btnShowNotification);
        btnShowNotification.setOnClickListener(
            new OnClickListener() {
                @Override
                public void onClick(View v) {
                    showNotification();
                }
            }
        );
    }
}
```

```
public void showNotification() {
    NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(
            Context.NOTIFICATION_SERVICE);

    Intent notificationIntent =
        new Intent(this, ActivityNotificationExample.
class);
    PendingIntent contentIntent =
        PendingIntent.getActivity(this, 0,
            notificationIntent, 0);

    // Notificationobjektum összeállítása
    Notification notification =
        new Notification(R.drawable.icon,
            "Notification cimsor",
            System.currentTimeMillis());
    notification.defaults |= Notification.DEFAULT_SOUND;
    notification.defaults |= Notification.DEFAULT_
VIBRATE;
    notification.setLatestEventInfo(this,
        "Notification cim", "Notification szoveg",
        contentIntent);

    mNotificationManager.notify(NOTIFICATION_ID,
        notification);
}
}
```

Az elkészült alkalmazást a következő ábra szemlélteti.



8.2. ábra. *Értesítések megjelenítése*

8.3. A *WebView* nézet bemutatása

A *WebView* komponens lehetővé teszi, hogy webes tartalmakat jelenítsünk meg az Android-alkalmazásokban. Alapja egy WebKit-alapú renderelő motor, és ennek köszönhetően a gazdag, JavaScript- és CSS-alapú tartalmak is megjeleníthetők.

Az Androidban található *WebView* komponens rendkívül gazdag API-t biztosít a fejlesztőknek, így nemcsak egyszerű tartalom megjelenítésére használhatjuk, hanem különféle interakciókat is folytathatunk vele.

A fő funkciók közül néhány a következő:

- előre/hátra navigáció,
- előzmények,

- nagyítás/kicsinyítés támogatása,
- szöveg keresése tartalomban,
- JavaScript-támogatás.

A *WebView* használatához a webről letöltött tartalom megjelenítésekor mindenképpen szükség van az *android.permission.INTERNET* engedélyre, amelyet a *manifest* állományban jeleznünk kell.

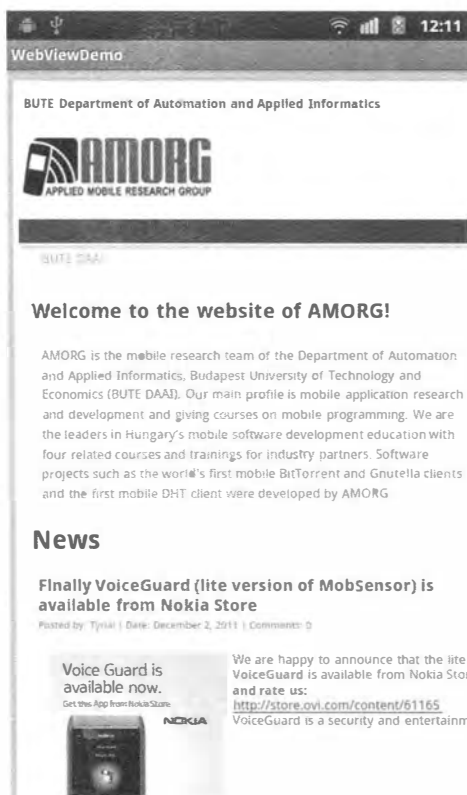
```
<uses-permission android:name="android.permission.INTERNET"/>
```

A *WebView*-t XML-erőforrásban könnyen definiálhatjuk, például:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

De ugyanígy bármilyen layoutba is beágyazható, nem kötelező a teljes képernyőt kitöltenie, illetve egyszerre akár több *WebView*-t is megjeleníthetünk. Tételezzük fel, hogy az előző erőforrás (*main.xml* névvel) alapján készítettünk egy *Activity*-t. Ebben az esetben a *WebView*-n a nagyításvezérlőket és a kezdő-URL-t a következőképpen állíthatjuk be (*loadUrl()* függvény):

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    WebView mWebView = (WebView) findViewById(R.
id.webview);
    mWebView.getSettings().setBuiltInZoomControls(true);
    mWebView.loadUrl("http://amorg.aut.bme.hu");
}
```



8.3. ábra. WebView nézet

A *WebView*-t többféleképpen vezérelhetjük. Előző példánkban már használtuk a *WebView*-hoz tartozó *WebSettings* objektumot, amelyet a *WebVeiw.getSettings()* függvényén keresztül érhetünk el. A *WebSettings* segítségével a *WebView* megjelenését módosíthatjuk. Többek között felvehetünk közelítő/távolító vezérlőket, állíthatjuk a betűméretet, engedélyezhetjük a JavaScript-használatot stb.

Elsőre nehezen értelmezhető probléma az előző példánkban, hogy ha a *WebView*-n megjelenített oldalon belül egy linkre kattintunk, akkor nem a *WebView*-n belül jutunk el a következő oldalra, hanem a telefon beépített böngészője nyílik meg, és az jeleníti meg az új oldalt. Amikor egy weboldalt megjelenítünk a *WebView*-n keresztül, számos olyan esemény léphet fel, amelyek befolyásolására gyakran szükség lehet. Ilyen események kezeléséhez megadhatunk a *WebView*-nak egy *WebViewClient* objektumot, ennek különféle eseményeit definiálhatjuk felül és kezelhetjük megfelelően. Ilyen események például a következők:

- *onFormResubmission(...)*: űrlapújraküldés,
- *onLoadResource(...)*: erőforrás-betöltés (pl. kép),

- *onPageFinished(...)*: oldal betöltésének befejeződése,
- *onReceivedError(...)*: hibaesemény,
- *onReceivedHttpAuthRequest(...)*: autentikációs kérés,
- *onReceivedLoginRequest(...)*: bejelentkezési kérés,
- *onReceivedSslError(...)*: titkosítási probléma,
- *shouldOverrideKeyEvent(...)*: esemény felüldefiniálása,
- *shouldOverrideUrlLoading(...)*: új URL megnyitásának a jelzése.

A legtöbb eseményhez tartozó függvény paraméterében megkapja az adott eseményhez releváns objektumokat, értékeket, így ezeket megfelelően kezelhetjük. Továbbá gyakori viselkedés, hogy a felsorolt függvények egy *boolean*-típusú visszatérési értéket várnak, és *true* értékkel való visszatéréssel tudjuk jelezni, hogy az adott eseményt megfelelően lekezeltek, nem kell tovább küldeni a rendszeren keresztül.

Az előző példát kiegészítve az alábbi kódrésszel biztosítható, hogy a *WebView*-n belül egy linkre kattintva ugyanazon a *WebView*-n jelenjen meg a tartalom:

```
mWebView.setWebViewClient(new WebViewClient() {
    @Override
    public void onReceivedError(WebView view,
        int errorCode, String description,
        String failingUrl) {
        // Hiba kezelése
    }

    @Override
    public boolean shouldOverrideUrlLoading(
        WebView view, String url) {
        mWebView.loadUrl(url);
        return true;
    }
});
```

A *WebView* támogatja a előzmények kezelését is, tehát tudunk előre/vissza navigálni. Ehhez a *WebView goForward()* és *goBack()* függvényeit használhatjuk. Egészítsük ki az előző példánkat úgy, hogy a vissza gomb hatására ne azonnal kilépjen az Activity, hanem a *WebView*-n lépünk vissza, ameddig lehetséges.

Ehhez az Activity-n belül az *onKeyDown()* függvényét kell felüldefiniálni (előtte az *mWebView*-t mezőként vegyük fel az osztályba, hogy el tudjuk érni).

```

@Override
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if ((keyCode == KeyEvent.KEYCODE_BACK) &&
        mWebView.canGoBack()) {
        mWebView.goBack();
        return true; // ne legyen alapértelmezett vissza
        esemény
    }
    return super.onKeyDown(keyCode, event);
}

```

További eseményeket is kezelhetünk a *WebView*-n keresztül, ezek közé a különféle JavaScript-események tartoznak. Kezelésükhöz egy *WebChromeClient* objektum megadására lesz szükség. A *WebChromeClient* segítségével felüldefiniálható néhány esemény:

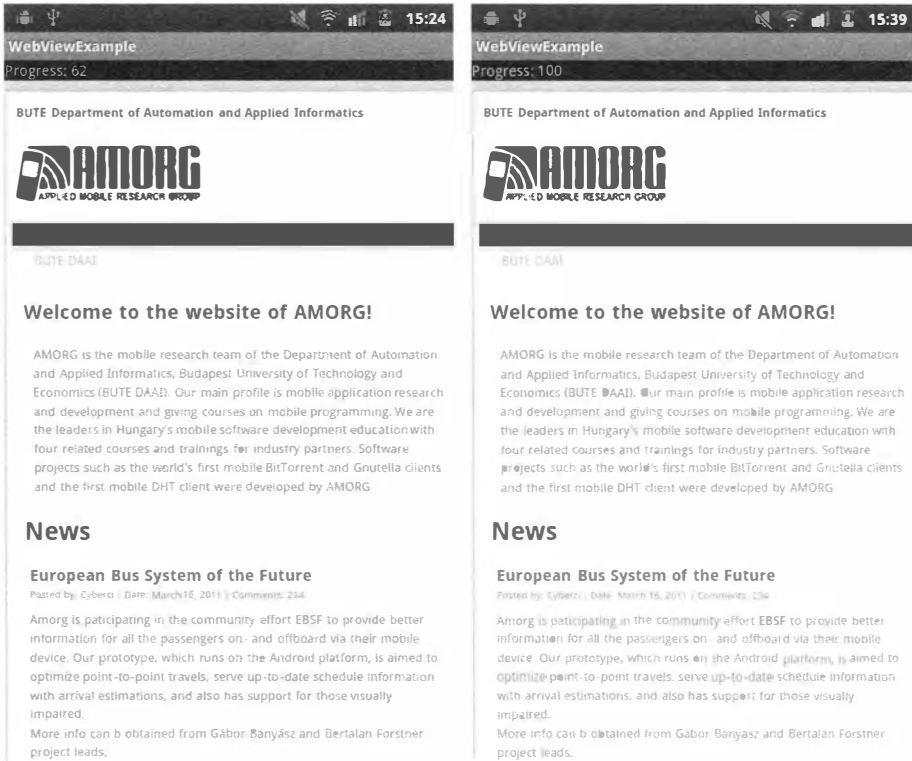
- *onJSTimeout(...)*: JavaScript-időtúllépés (timeout),
- *onJSAlert(...)*: JavaScript-figyelmeztetés,
- *onProgressChanged(...)*: oldal letöltésállapota százalékban,
- *onJSConfirm(...)*: JavaScript-megerősítés,
- *onCreateWindow(...)*: új ablak megjelenítése.

A *WebChromeClient* gyakorlásához tételezzük fel, hogy a felhasználói felületünk egy *LinearLayout*-ból áll, amelynek tetején van egy egyszerű *TextView* (*tvStatus*), alatta pedig egy *WebView* (*mWebView*). A következő példa megmutatja, hogyan tudjuk a *TextView*-n megjeleníteni a weboldal betöltődésének az állapotát:

```

mWebView.setWebChromeClient(new WebChromeClient() {
    @Override
    public void onProgressChanged(WebView view, int
    progress) {
        tvStatus.setText("Progress: "+progress);
    }
});

```

8.4. ábra. Oldal betöltésállapotának jelzése

Végül az utolsó példában tételezzük fel, hogy egy olyan oldalt szeretnénk a *WebView*-n megjeleníteni, amelyen egy gombnyomásra egy JavaScript-figyelmeztető (*alert*) jelenik meg. A példában használt, *alert*et megjelenítő gomb HTML- és JavaScript-forrása a következő:

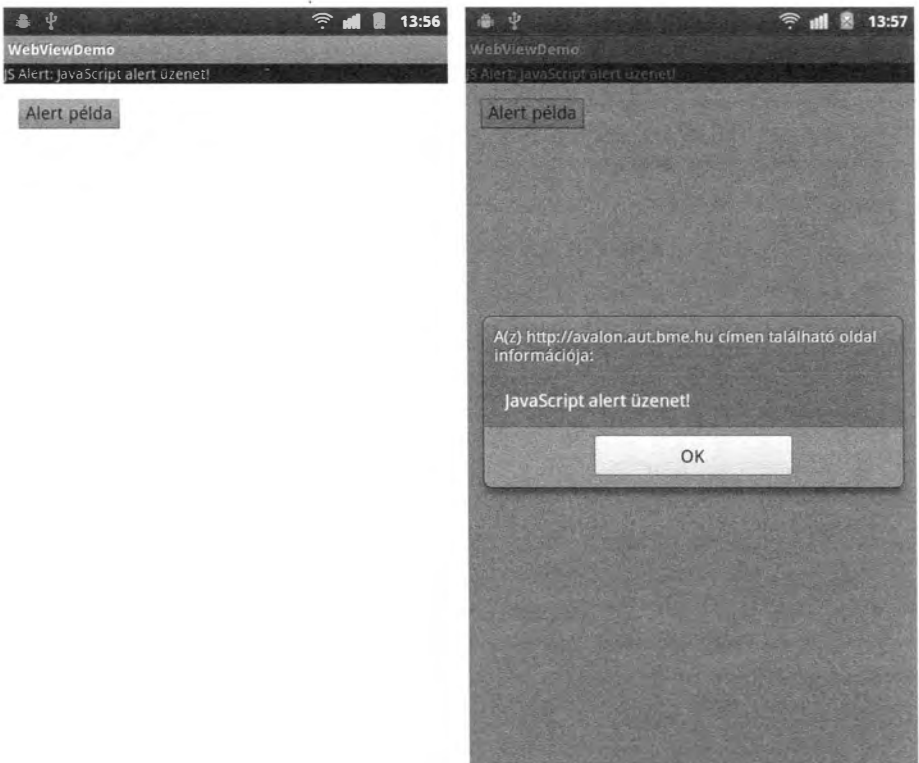
```
<input type="button" value="Alert példa"
      onClick='alert("JavaScript alert üzenet!")'>
```

A *WebChromeClient* beállítása a JavaScript-alert kezeléséhez a következő:

```
mWebView.getSettings().setJavaScriptEnabled(true);
mWebView.setWebChromeClient(new WebChromeClient() {
    @Override
    public boolean onJsAlert(WebView view,
        String url, String message, JsResult result) {
        textViewStatus.setText(
            "JS Alert: " + message);
    }
});
```

```
// return true; esetén nem fut le a default  
// implementáció (alapértelmezett pop-up ablak)  
return super.onJsAlert(  
    view, url, message, result);  
}  
});
```

Az elkészült alkalmazást a következő ábra szemlélteti.



8.5. ábra. JavaScript-alert kezelése

8.4. HTTP-kapcsolatok kezelése

Mobilalkalmazások fejlesztésekor gyakran szükséges egy szerveroldali komponenssel kommunikálni, ahol például egy nagy adatbázist érhetünk el, információkat kérhetünk le, adatokat tölthetünk fel stb. Napjainkban az egyik legnépszerűbb kommunikációs protokoll a HTTP, népszerűségének okai az egyszerűségében, a megbízhatóságában és a rugalmasságában kereshetők.

Az Android gazdag HTTP API-val rendelkezik, amely a közismert Java API-kon alapul, illetve az Apache gondozásában továbbfejlesztett HTTP API is megtalálható az Androidban. A rendszer így egy teljes körű HTTP-osztálykönyvtárat biztosít a fejlesztőknek, amely támogatja az ismert *http* üzenettípusokat (GET, POST stb.), a HTTPS-titkosítást, valamint gazdag konfigurációs felületet nyújt. Könyvünknek nem célja az Android teljes HTTP-eszköztárát bemutatni, csupán a gyakran előforduló eseteket ismertetjük.

A HTTP-kapcsolat használatakor jeleznünk kell a *manifest* állományban az *android.permission.INTERNET* engedélyt, különben a rendszer nem engedi az adatforgalmat.

A hálózati kommunikációt mindig külön számban valósítsuk meg, ugyanis a felhasználói felület blokkolása nem elfogadott a kapcsolat lassúsága miatt. A válasz beérkezésekor a felhasználói felületet nem módosíthatjuk ebből a külön számból, hiszen a felhasználói felület csak a fő számból módosítható: ennek kezelésére különféle technikákat mutatunk be.

Emellett HTTP-alapú kommunikáció esetében mindig gondosan ellenőrizzük a válasz státuskódját,²¹ és ennek megfelelően kezdjük csak el az adatfeldolgozást.

Hálózati kommunikáció esetén a hiba- és kivételkezelésre fokozottan ügyeljünk, hiszen ilyenkor gyakrabban találkozhatunk kivételekkel hálózati jellemzők, időtűllépés, kapcsolat megszakadása stb. miatt.

A következőkben az Apache HTTP-osztálykönyvtárait ismertetjük, és nem a Java közismert *URLConnection* osztályán keresztül mutatjuk be a HTTP-kommunikáció lépéseit.

8.4.1. A HTTP GET támogatása

A kapcsolat megnyitásához elsőként szükség van egy *HttpClient* példányra, amelynek főbb feladatai a következők:

- HTTP-kliens-interfész megvalósítása,
- HTTP-kérések végrehajtása,
- cookie-k kezelése,
- autentikáció támogatása,
- kapcsolat fenntartása és kezelése.

HttpClient példányként használható az alapértelmezett *DefaultHttpClient* is, ám létezik egy úgynevezett *AndroidHttpClient* osztály is (az Android 2.2-től), amely a *DefaultHttpClient* kiterjesztése, és előre definiáltan az Androidon tipikusan használt beállításokat tartalmazza. Támogatja

²¹ HTTP-állapotkódok: <http://www.w3.org/Protocols/HTTP/HTRESP.html>

a *HttpRequestInterceptor*t, amelynek segítségével a szerverre küldés előtt közvetlenül vagy az érkezés után közvetlenül hozzáférhetünk a kéréshez. Az *AndroidHttpClient* példányosítása a *newInstance(url)* *factory* metódussal történik.

AndroidHttpClient használata esetén ne felejtsük el használat után bezárni a *close()* függvénnyel.

A HTTP-kérés „megfogalmazásához” szintén egy külön objektum példányra van szükség. HTTP GET kérés esetén az *org.apache.http.client.methods.HttpGet* osztályt kell példányosítanunk, ahol a konstruktorban megadható a kívánt URL.

```
HttpGet httpGet = new HttpGet("http://amorg.aut.bme.hu");
```

Természetesen az URL-ben GET-paraméterek is átadhatók a szokásos módon. Példaként létrehoztunk egy egyszerű PHP-oldalt, amely a *name* paraméterében kapott érték elé fűzi a „Hello” szöveget:

```
HttpGet httpGet = new HttpGet(
    "http://avalon.aut.bme.hu/~tyrael/phpget.php?name=JohnDoe");
```

Ám ha szóköz vagy más, nem általános karakter szerepel az URL-paraméterben, például: „John Doe”, akkor egy ilyen hívásnál az alábbi kivételt kapnánk:

```
11-26 18:39:24.417: ERROR/AndroidRuntime(17232): java.
lang.IllegalArgumentException: Illegal character in
query at index 53:
    http://avalon.aut.bme.hu/~tyrael/phpget.php?name=John Doe
```

URL-paraméterek kezelése esetén alapszabály, hogy az (a..z, A..Z) karaktereken, számokon (0..9) és ., -, *, _ karaktereken kívül minden karaktert a hexadecimális megfelelőjévé kell konvertálni, például: # -> %23. A konvertálást nem kell manuálisan megvalósítani, használhatjuk a beépített *URLEncoder* osztályt, amelynek statikus *encode()* függvényével tetszőleges *String* értéket alakíthatunk át. Opcióként az *encode()* függvény második paraméterének a karakterkészletet is megadhatjuk.

Paraméteres HTTP GET kérés összeállítása *URLEncoder* használatával a következő:

```
String name = URLEncoder.encode("John Doe");
connect(
    "http://avalon.aut.bme.hu/~tyrael/phpget.
    php?name="+name);
```

Ha az encode-olt paramétert valami miatt vissza kell állítanunk (például a kérésülék HTTP-szerverként viselkedik), akkor az *URLDecoder* osztály statikus *decode()* függvényét használhatjuk.

Az összeállított *HttpGet* objektumot a korábban említett *HttpClient*et megvalósító objektum *execute()* függvényével tudjuk elküldeni. A kérés végrehajtásakor az *execute()* függvény eredménye egy *HttpResponse* objektum lesz. A *HttpResponse* objektum *getStatusLine()* függvényével érdemes elsőként elkérni az állapotot leíró *StatusLine* objektumot, amelynek *getStatusCode()* függvényével a korábban említett HTTP-válasz kódja lekérdezhető. A válaszkódokat azonban nem kell fejből tudni, hiszen a beépített *HttpStatus* osztályban mindegyik megtalálható, és elérhető statikus konstansként (pl. *HttpStatus.SC_OK*).

A válasz tartalmát egy *HttpEntity* objektum jelképezi, ezt szintén a *HttpResponse* objektumtól tudjuk elkérni a *getEntity()* függvény segítségével. A *HttpEntity*nek három fő típusa lehet:

- *streamed*: a tartalom egy *stream*ben érhető el;
- *self-contained*: a tartalom a memóriában van, vagy egy kapcsolattól függ még;
- *wrapping*: a tartalom egy másik entitás része.

Végül a *HttpEntity* által hordozott konkrét válasz, *InputStream* formájában a *HttpEntity* *getContent()* függvényével válik elérhetővé.

Összefoglalva a HTTP GET kérés folyamata a következő:

- *HttpClient* példány:

```
DefaultHttpClient httpClient = new
DefaultHttpClient();
// vagy
AndroidHttpClient httpClient =
    AndroidHttpClient.newInstance("Android");
```

- *HttpGet* kérés összeállítása:

```
HttpGet httpGet = new HttpGet("http://www.aut.bme.hu");
```

- HTTP-kérés végrehajtása és az eredmény eltárolása:

```
HttpResponse response = httpClient.execute(httpGet);
```

- Állapotkód lekérdezése:

```
StatusLine sl = response.getStatus();  
if (sl.getStatusCode() == HttpStatus.SC_OK) {  
    // válasz kiolvasása  
}
```

- Válasz lekérése:

```
HttpEntity entity = response.getEntity();
```

- Eredmény kiolvasása:

```
InputStream inStream = entity.getContent();
```

Az eddigiek alaposabb megismeréséhez elsőként nézzünk meg egy egyszerű példát, ahol egy gomb megnyomására hívódik meg egy HTTP GET kérés, és a válasz egy *Toast* formájában jelenik meg.

```
public class HttpDemoActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
  
        Button btnHttpBasicGet =  
            (Button)findViewById(R.id.btnHttpBasicGet);  
        btnHttpGet.setOnClickListener(new  
        OnClickListener() {  
            @Override  
            public void onClick(View arg0) {  
                new Thread() {  
                    public void run() {  
                        httpBasicGetDemo();  
                    }  
                }  
            }  
        });  
    }  
}
```

```

        }
    }.start();
}
});
}

private void httpBasicGetDemo() {
    AndroidHttpClient httpClient = null;
    try {
        httpClient =
            AndroidHttpClient.newInstance("Android");
        String name = "John Doe";
        name = URLEncoder.encode(name);
        HttpGet httpGet = new HttpGet(
            "http://avalon.aut.bme.hu/~tyrael/phpget.
php?name="+
            name);
        final String resp = httpClient.execute(
            httpGet, new BasicResponseHandler());
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(HttpDemoActivity.this,
                    resp, Toast.LENGTH_LONG).show();
            }
        });
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (httpClient != null)
            httpClient.close();
    }
}
}
}

```

A fenti példában a GET-kérést külön anonim számban küldjük el. A GET-kérés elküldéséhez *AndroidHttpClient*-et használtuk, amely egyébként is megköveteli, hogy külön számban történjen a kommunikáció, ellenkező esetben kivétel dobódna.

Az előző példában a válasz kiolvasásához nem használtuk a *HttpResponse* és a *HttpEntity* osztályokat, mivel az Android által nyújtott *BasicResponseHandler* elegendő volt. A *BasicResponseHandler* jól használható, ha rövid, szöveges válasz a visszatérési érték.

A válasz megjelenítését a felhasználói felületen már nem a létrehozott új számból végezzük el, hanem a *runOnUiThread()* függvény segítségével visszatérünk a fő számba.

Alakítsuk át a példánkat úgy, hogy a válasz kiolvasását már az *InputStream* segítségével valósítjuk meg.

```
private void httpGetDemo() {
    AndroidHttpClient httpClient = null;
    try {
        httpClient = AndroidHttpClient.
newInstance("Android");
        String name = "John Doe";
        name = URLEncoder.encode(name);
        HttpGet httpGet = new HttpGet(
            "http://avalon.aut.bme.hu/~tyrael/phpget.
php?name=" +
            name);
        HttpResponse response = httpClient.
execute(httpGet);
        if (response.getStatusLine().getStatusCode() ==
            HttpStatus.SC_OK) {
            HttpEntity entity = response.getEntity();
            if (entity != null) {
                InputStream is = entity.getContent();
                processResponse(is);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (httpClient != null)
            httpClient.close();
    }
}

private void processResponse(final InputStream aIs) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            try {
                StringBuilder sb = new StringBuilder();
                int chIn;
                while ((chIn=aIs.read()) != -1) {
                    sb.append((char)chIn);
                }
                Toast.makeText(HttpDemoActivity.this,
sb.toString(),
                    Toast.LENGTH_LONG).show();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (aIs != null) {
```



```

        try {
            aIs.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
});
}

```

Ha az alkalmazásunk több HTTP-kérést hajt végre, érdemes egyetlen *HttpClient* objektumot használni alkalmazásszinten. Ilyenkor ideálisan alkalmazható a Singleton tervezési minta, amelyben egy *synchronized* „getter” függvényen keresztül biztosítunk elérést a *HttpClient* objektumhoz.

8.4.2. *AsyncTask* használata a HTTP-kommunikációban

Android platformon egy alkalmazás indításakor a rendszer létrehoz egy úgynevezett fő szálát és minden, felhasználói felületet érintő utasításnak ebben a szálban kell lefutnia. Ám a sokáig tartó műveletek, mint a hálózati kommunikáció, blokkolhatják a felhasználói felületet, ezért ezeket új szálban kell végrehajtani. A probléma azonban az, hogy az ilyen műveletek végén az eredményt tipikusan a felhasználói felületen kell megjelenítenünk, ezt viszont nem tehetjük meg másik szálból. Az Android erre a szituációra több megoldást is biztosít:

- *Activity.runOnUiThread(Runnable)*: A függvényen belül, a megadott *Runnable* objektum által leírt folyamat már hozzáfér a UI-hoz.
- *View.post(Runnable)*: Egy konkrét nézet aszinkron módon frissítheti a tartalmát.
- *View.postDelayed(Runnable, long)*: Egy konkrét nézet aszinkron módon frissíti a tartalmát, ahol a második paraméter a késleltetés.
- *Message-Handler* mechanizmus: A másik szál üzenetek formájában jelez vissza a főszálnak.
- *AsyncTask*: Android által biztosított megoldás, amely elfedi, hogy a hosszan tartó művelet a háttérben fut.

A következőkben nézzünk meg egy egyszerű példát az *AsyncTask* használatára. *AsyncTask* használatakor egy saját osztályt kell létrehozni, amely az *AsyncTask* absztrakt osztályból származik le. Egyedül a *doInBackground()* függvényt kell felüldefiniálnunk, amely a háttérben egy új szálon fut le,

legtöbbször azonban egyéb függvényeket is felüldefiniálunk, ezek jelentése a következő:

- *onPreExecute()*: Előkészületek megtétele, ez még a fő szálon fut, módosíthatja az UI-t.
- *doInBackground()*: Külön háttérzálon fut.
- *onProgressUpdate()*: Fő szálon fut, a *doInBackground()* függvényből tudunk jelezni neki a *publishProgress()* függvény segítségével.
- *onPostExecute()*: Fő szálon fut, tipikusan az eredmények megjelenítésére használatos.

Az *AsyncTask* generikus osztály, template-ekkel dolgozik: megadható, hogy a hívási paramétereknek, a státuszfrissítési paramétereknek, illetve az eredményparaméternek milyen legyen a típusa.

Az *AsyncTask* használatára nézzünk egy egyszerű példát. A példában egy HTTP GET kérést valósítunk meg *AsyncTask*kal.

```
public class HttpGetAsyncTask extends
    AsyncTask<String, Integer, String> {

    private Context context = null;
    private ProgressDialog progressDialog = null;
    private long responseLength;

    public HttpGetAsyncTask(Context context) {
        this.context = context;
    }

    @Override
    protected void onPreExecute() {
        progressDialog = new ProgressDialog(this.context);
        progressDialog.setMessage("Kérem várjon...");
        progressDialog.show();
    }

    @Override
    protected void onCancelled() {
        // TODO Auto-generated method stub
        super.onCancelled();
    }

    @Override
    protected String doInBackground(String... urls) {
        String retVal = "";
        AndroidHttpClient httpClient =
            AndroidHttpClient.newInstance("Android");
        HttpGet httpGet = new HttpGet(urls[0]);
```

```

InputStream is = null;
try {
    HttpResponse response = httpClient.execute(httpGet);
    if (response.getStatusLine().getStatusCode() ==
        HttpStatus.SC_OK) {
        HttpEntity entity = response.getEntity();
        responseLength = entity.getContentLength();
        if (entity != null) {
            is = entity.getContent();
            StringBuilder sb = new StringBuilder();
            int chIn;
            int idx = 0;
            while ((chIn=is.read()) != -1) {
                sb.append((char)chIn);
                publishProgress(++idx);
            }
            retVal = sb.toString();
        }
    }
} catch (Exception e) {
    retVal = "Error: "+e.getMessage();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (httpClient != null)
        httpClient.close();
}
return retVal;
}

@Override
protected void onProgressUpdate(Integer... values) {
    double percent = 0;
    if (responseLength != 0)
        percent = 100*values[0]/responseLength;
    Log.d("AsyncTask progress", ""+percent+" %");
}

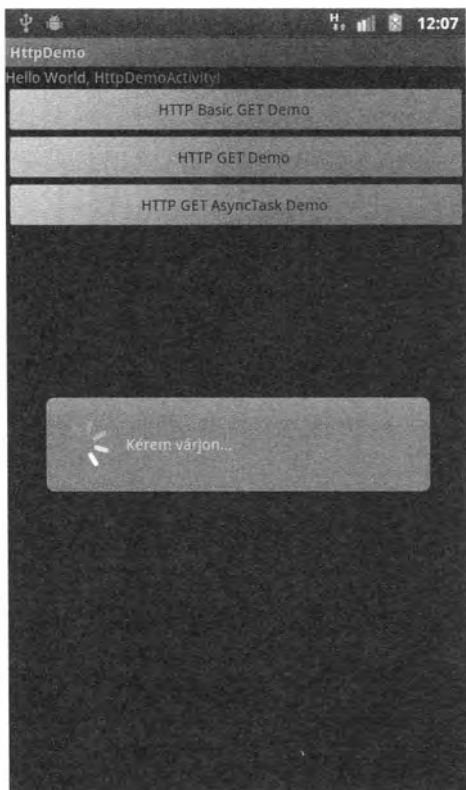
@Override
protected void onPostExecute(String result) {
    progressDialog.dismiss();
    Toast.makeText(context, result, Toast.LENGTH_LONG).
show();
}
}

```

Figyeljük meg a megvalósított *AsyncTask* osztályt. A kérés indítása előtt egy *ProgressDialog*-ot jelenítünk meg, amely mindvégig aktív a kérés befejeződéséig. Vizsgáljuk meg a *doInBackground()* függvényben, hogyan frissítjük az állapotot a *publishProgress()* függvénnyel. Ebben a megvalósításban a *doInBackground()* elején lekérdezzük a válasz méretét, és az *onProgress Update()*-ben a *LogCatra* folyamatosan jelezzük a letöltés állapotát százalékos formában.

Az *AsyncTask* indítását a következő kódrész mutatja be, ezt tehát nem kell külön számba helyezni:

```
String name = "John Doe";
name = URLEncoder.encode(name);
String url =
    "http://avalon.aut.bme.hu/~tyrael/phpget.php?name="
+ name;
new HttpGetAsyncTask(getApplicationContext()).
execute(url);
```



8.6. ábra. *AsyncTask*-példa *ProgressDialog*-használattal

AsyncTask használatakor ügyeljünk arra, hogy ha a hívó Activity valamiért háttérbe kerül, mielőtt az *AsyncTask* befejeződné, akkor az Activityhez tartozó felhasználói felületet már nem érjük el, ezért ezt az esetet fejlesztőként külön kell kezelnünk. Gyakran találkozhatunk ezzel, ha elfordítjuk a képernyőt, hiszen ilyenkor új Activity jön létre, és lehet, hogy az *AsyncTask* már egy nem létező Activitynek jelezne vissza. Megoldásként az *AsyncTask* és az Activity közti laza csatolást javasoljuk, például az alkalmazásszintű Broadcasttal való kommunikációt.

8.4.3. HTTP POST támogatása

A HTTP POST üzenetek kezelése szinte teljesen ugyanúgy történik Android-környezetben, mint a HTTP GET, ám nem mindegyik Android-verziót tartalmazza beépítetten, hanem külön osztálykönyvtárként le kell töltenünk, és importálnunk kell a projektünkbe.²²

POST-típusú üzenet küldéshez a megszokott módon egy *HttpClient* objektumpéldányra van szükség, majd a POST-üzenetet egy *HttpPost* objektumban kell összeállítani az URL megadásával.

```
HttpPost httpPost =
    new HttpPost("http://www.example.com/post.php");
```

A POST-paraméterek beállításához a *HttpPost* objektum *setEntity()* függvényét használhatjuk, amely például kulcs-érték pár alapú listával inicializálható, ám előtte legtöbbször még be kell csomagolni egy *UrlEncodedFormEntity* objektumba:

```
List<NameValuePair> nameValuePairs =
    new ArrayList<NameValuePair>(2);
nameValuePairs.add(new BasicNameValuePair("userid",
    "ad2435"));
nameValuePairs.add(new BasicNameValuePair("name",
    "admin"));
httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
```

A következőkben nézzünk meg egy teljes függvényt, amely egy POST-kérést valósít meg:

```
public void postData() {
    HttpClient httpClient = new DefaultHttpClient();
```

²² Apache *HttpClient* könyvtár HTTP POST támogatással: <http://apache.mirrors.crysys.hit.bme.hu/dist/httpcomponents/httpclient/binary/httpcomponents-client-4.1.2-bin.zip>

```

HttpPost httpPost =
    new HttpPost("http://www.example.com/post.php");
try {
    // Post adat összeállítása
    List<NameValuePair> nameValuePairs =
        new ArrayList<NameValuePair>(2);
    nameValuePairs.add(new
BasicNameValuePair("userid",
        "ad2435"));
    nameValuePairs.add(new BasicNameValuePair("name",
        "admin"));
    httpPost.setEntity(
        new UrlEncodedFormEntity(nameValuePairs));
    // HTTP Post kérés végrehajtása
    HttpResponse response = httpClient.
execute(httpPost);
    // ... válasz feldolgozása
} catch (Exception e) {
    e.printStackTrace();
}
}

```

Az előző példából a válasz feldolgozását kihagytuk, ugyanis az ugyanúgy működik, ahogyan a HTTP GET résznél bemutattuk.

8.4.4. A HTTPS és a proxy beállítása

Az Android platform teljes körűen támogatja a HTTPS-alapú titkosítást, amellyel biztonságos HTTP-kéréseket indíthatunk. Ennek segítségével tehát minden HTTP protokoll felett küldött adat titkosítása megvalósul, így nem lehet lehallgatni a kommunikációt. A HTTPS alkalmazásának tipikus lépései a következők:

- Certificate/aláírás generálása,
- HTTPS beállítása szerveroldalon a generált kulccsal,
- Certificate beállítása az Android-alkalmazásban és az alkalmazás aláírása,
- alapesetben az Android-alkalmazások „debug” kulccsal vannak aláírva, ám így nem kerülhetnek fel a Marketre, tehát ezt le kell cserélni.

A HTTPS-kezelés nagymértékben hasonlít a standard Java HTTPS-támogatásához, ezért a következőkben csak egy egyszerű kódrészt mutatunk be, amely egy HTTPS-kérést valósít meg:

```

URL url = new URL("https://server:443/getid.php");
HttpsURLConnection httpsConn = (HttpsURLConnection)
url.openConnection();
httpsConn.setConnectTimeout(15000);
// kérés elküldése és a válaszkód ellenőrzése
if (httpsConn.getResponseCode() == HttpsURLConnection.
HTTP_OK)
{
    InputStream is = httpsConn.getInputStream();
    //...
    is.close();
}

```

Előfordulhat, hogy Android-környezetben egy proxyt kell beállítanunk, amely a teljes HTTP-forgalmat befolyásolja (ilyen például a céges környezet). Ebben az esetben a rendszer beállításait próbálhatjuk meg átírni, ám vegyük figyelembe, hogy ezt nem minden készülékgyártó engedi meg.

A szükséges engedélyek, amelyekkel a rendszer beállításaihoz hozzáférhetünk, a következők:

```

<uses-permission android:name=
    "android.permission.WRITE_SETTINGS"/>
<uses-permission android:name=
    "android.permission.WRITE_SECURE_SETTINGS"/>

```

A proxy beállítását a következő kódrésszel tehetjük meg (ha lehetséges):

```

Settings.System.putString(getContentResolver(),
    Settings.System.HTTP_PROXY, "myproxy:8080");

```

A proxy kikapcsolása pedig az alábbiak szerint történik:

```

Settings.System.putString(getContentResolver(),
    Settings.System.HTTP_PROXY, "");

```

Végül meg kell említeni, hogy nagyobb méretű állományok letöltésére Android 2.3.x-től felfelé a *DownloadManager* osztály is használható, amely a letöltés befejeződéséről egy *BroadcastReceiver*-en keresztül küld értesítést (*DownloadManager.ACTION_DOWNLOAD_COMPLETE*).

8.5. Szabványos kommunikációs formátumok feldolgozása

Hálózati kommunikáció esetén az üzenetek átküldése gyakran valamilyen ismert formátumban történik, ilyen például a CSV, a JSON, az XML. A következőkben ezeket a formátumokat és a feldolgozásukat tekintjük át röviden.

A CSV formátuma tulajdonképpen vesszővel elválasztott oszlopértékeket jelent, ahol a sorvégjel egy új sort jelképez a képzeletbeli táblázatban. Elsőre azt gondolhatnánk, hogy egy ilyen formátum feldolgozásához elegendő a *String* osztály *split()* függvényét használnunk, amely helyénvaló is lenne például az alábbi CSV esetében:

```
john,doe,1976
kate,doe,1977
```

Ám a CSV-szabvány szerint az alábbi formátum is elfogadott:

```
john,doe,(1976,05,12)
kate,doe,(1977,15,02)
```

Láthatjuk tehát, hogy a *split()* függvény már kudarcot vallana, és a zárójeltől függetlenül a harmadik oszlopot több részre vágná szét, hibásan. Éppen ezért a CSV feldolgozásakor a zárójelezésre különösen ügyeljünk.

8.5.1. JSON-feldolgozás

A JSON egy olyan jól ismert formátum, amely alapvetően kulcs-érték párok tárolását teszi lehetővé, ahol az érték önmagában lehet egy másik JSON-objektum vagy tömb is, amely rekurzívan szintén kulcs-érték párokat tárol.

Szintaktikai elemei a következők: { }, [], :, ;. JSON-adatok feldolgozásához az Android, illetve a Java-környezet két fő osztályt biztosít: *JSONObject*, *JSONArray*.

A *JSONObject* fő tulajdonságai a következők:

- JSON-objektumok feldolgozása (parse-olása),
- értékek elérése a kulcs megadásával:
 - *getString(String key)*: direkt érték elkérése,
 - *getJSONObject(String key)*: JSON-objektum elérése,
 - *getJSONArray(String key)*: JSON-tömb elérése,
- JSON-objektum létrehozása *String*ből vagy *Map* objektumból.

A *JSONArray* fő jellemzői a következők:

- *JSONObject*hez hasonló működés JSON-tömbökkel,
- Feldolgozás, elemek lekérdezése index alapján, hossz lekérdezhetősége,
- létrehozás például *Collection*ből.

A JSON-feldolgozáshoz nézzünk meg egy egyszerű példát. A következő link segítségével a Google geocoding szolgáltatásának köszönhetően a „Budapest Astoria” cím földrajzi koordinátáit kérdezzük le, ahol a válasz JSON-formátumban érkezik meg:

<http://maps.googleapis.com/maps/api/geocode/json?address=Budapest+Astoria&sensor=false>

A JSON-válasz kicsit rövidítve a következő:

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "Astoria",
          "short_name" : "Astoria",
          "types" : [ "subway_station", "establishment",
                     "transit_station" ]
        },
        ...
      ],
      "formatted_address" : "Astoria, 1088 Budapest,
                           Magyarország",
      "geometry" : {
        "location" : {
          "lat" : 47.4943310,
          "lng" : 19.0600970
        },
        "location_type" : "APPROXIMATE",
        ...
      }
    }
  ],
  "status" : "OK"
}
```

Végül az Android-forrás, amely végrehajtja a HTTP GET kérést, és feldolgozza a JSON-választ, az alábbi:

```
private void processJSON() {
    AndroidHttpClient httpClient = null;
    try {
        httpClient = AndroidHttpClient.
newInstance("Android");
        HttpGet httpGet = new HttpGet(
            "http://maps.googleapis.com/maps/api/geocode/
json?" +
            "address=Budapest%20Astoria&sensor=false");
        final String resp = httpClient.execute(httpGet,
            new BasicResponseHandler());

        // --- JSON feldolgozás
        JSONObject root = new JSONObject(resp);
        JSONObject location = root.
getJSONArray("results").
getJSONObject(0).getJSONObject("geometry").
getJSONObject("location");
        final String lat = location.getString("lat");
        final String lng = location.getString("lng");
        // ---

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(HttpDemoActivity.this,
lat+", "+lng,
                    Toast.LENGTH_LONG).show();
            }
        });
    } catch (JSONException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (httpClient != null)
            httpClient.close();
    }
}
```

Figyeljük meg a példában, hogy hogyan jutunk el a „location” kulcs által tárolt *JSONObject*hez, és a végén hogyan kérdezzük le a *lat* és az *lng* értékeket.

Önálló alkalmazás esetén a fenti függvényt külön szálba kell meghívunk, hogy ne blokkoljuk a felhasználói felületet.

8.5.2. XML-feldolgozás

Az XML egy talán még a többinél is gyakrabban használt formátum a hálózaton küldött üzenetek számára, ugyanis jól olvasható és összetett adatstruktúrák reprezentálására is alkalmas. Az Android gazdag API-t biztosít az XML-feldolgozásra, ezek nagy része a Java-világ gazdagságából ered. Két fő típusú XML-feldolgozást szokás használni.

Az egyik a SAX- (Simple API for XML) alapú feldolgozó, amelyet a *SAXParse* osztály reprezentál. Ennek a feldolgozónak az a lényege, hogy nem tölti be a teljes XML-t a memóriába, hanem folyamatosan dolgozza fel az állományt, és az egyes csomópontokhoz/attribútumokhoz stb. érve különféle események generálódnak, amelyekre feliratkozhatunk.

A másik a DOM- (Document Object Model) alapú feldolgozó, amelyet a *DocumentBuilder* és a *DocumentBuilderFactory* osztályok valósítanak meg. DOM-alapú feldolgozás esetén a teljes XML a memóriába kerül, és egy fajlegű struktúrából lekérdezhetők az elemek.

A következőkben a DOM-feldolgozásra mutatunk egy példát. A példához használjuk a Google időjárás-API-ját, amely tesztelés céljából elérhető az alábbi helyen: <http://www.google.com/ig/api?weather=Budapest&hl=hu>.

A válasz rövidítve a következő formában érkezik:

```
<xml_api_reply version="1">
  <weather module_id="0" tab_id="0" mobile_row="0"
    mobile_zipped="1" row="0" section="0">
    <forecast_information>
      <city data="Budapest, Budapest"/>
      ...
    </forecast_information>
    <current_conditions>
      <condition data="Túlnyomóan felhős"/>
      <temp_f data="52"/>
      <temp_c data="11"/>
      <humidity data="Páratartalom: 62%"/>
      <icon data="/ig/images/weather/mostly_cloudy.
gif"/>
      <wind_condition data="Szél: ÉNy 19 km/ó"/>
    </current_conditions>
    ...
  </weather>
</xml_api_reply>
```

Készítsünk egy függvényt, amely letölti és értelmezi ezt az XML-t és a *condition* elem *data* értékét, valamint az *icon* elem *data* értékében definiált képet hozzáfűzi a kezdő *LinearLayout*hoz.

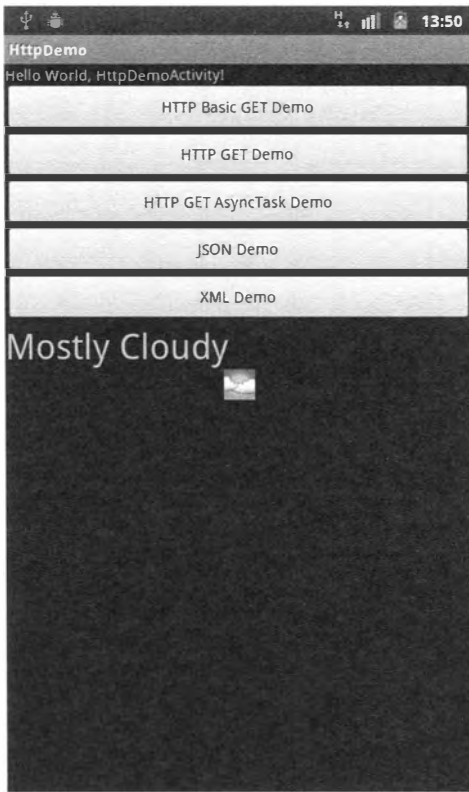
```

private void processXML(final LinearLayout aBaseLayout) {
    try {
        URL url = new URL(
            "http://www.google.com/ig/
api?weather=Budapest&hl=en");
        DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse(
            new InputSource(url.openStream()));
        doc.getDocumentElement().normalize();
        // Időjárás kiolvasása
        NodeList current_conditions = doc.
            getElementsByTagName("current_conditions");
        NodeList condition =
            ((Element) current_conditions.item(0)).
            getElementsByTagName("condition");
        final String weather = ((Element) condition.
            item(0)).
            getAttribute("data");
        // Ikon letöltése
        NodeList icon = ((Element) current_conditions.
            item(0)).
            getElementsByTagName("icon");
        URL urlIcon = new URL("http://www.google.com" +
            ((Element) icon.item(0)).getAttribute("data"));
        final Bitmap weatherIcon =
            BitmapFactory.decodeStream(urlIcon.
                openStream());

        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                // Időjárás szövege
                TextView tvWeather =
                    new TextView(HttpDemoActivity.this);
                tvWeather.setText(weather);
                aBaseLayout.addView(tvWeather);
                // Időjárás ikonja
                ImageView ivWeather =
                    new ImageView(HttpDemoActivity.this);
                ivWeather.setImageBitmap(weatherIcon);
                aBaseLayout.addView(ivWeather);
            }
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Ezt a szálát külön szálból kell futtatnunk. Figyeljük meg, hogy hogyan történik a paraméterül kapott *aBaseLayout*hoz az eredmények hozzáfűzése.



8.7. ábra. XML-feldolgozás

8.6. Socket-alapú kommunikáció

A HTTP-alapú kommunikáció mellett gyakran szükség lehet direkt Socket-alapú kapcsolatok kezelésére is, ahol a másik fél címének és portazonosítójának megadásával közvetlen kapcsolatot létesíthetünk.

Az Android a Java nyelvből ismert osztályok használatával ugyanígy képes TCP/IP kapcsolatokat kezelni, és a Socket-alapú kommunikáció is ugyanígy történik. Éppen ezért nem megyünk bele a részletekbe, csupán egy egyszerű példát mutatunk a Socket-alapú kommunikációra. (Megjegyzendő, hogy UDP-alapú kommunikációra hasonlóan a standard Java-környezetből ismert osztályok használhatók.)

Példánkban egy webszerverhez kapcsolódunk a hagyományos 80-as porton, és a megfelelő HTTP-fejléc elküldése után a választ egy *Toast*-ban jelenítjük meg.

```

private void socketDemo() {
    Socket socket = null;
    InputStream is = null;
    OutputStream os = null;
    try {
        socket = new Socket("217.20.130.97",80);
        socket.setSoTimeout(10000);
        // adatok küldése
        os = socket.getOutputStream();
        os.write("GET index.html HTTP/1.0\n".getBytes());
        os.write("From: john@doe.com\n".getBytes());
        os.write(
            "User-Agent: BMEAUT-AndroidBook/1.0\n".
getBytes());
        os.write("\n".getBytes());
        os.flush();
        // adatok fogadása
        is = socket.getInputStream();
        InputStreamReader isr = new InputStreamReader(is,
            "UTF-8");
        StringBuilder resultBuffer = new StringBuilder();
        int inChar;
        while ((inChar = isr.read()) != -1) {
            resultBuffer.append((char) inChar);
        }
        final String result = resultBuffer.toString();
        // eredmény megjelenítése
        runOnUiThread(new Runnable() {
            public void run() {
                Toast.makeText(SocketDemoActivity.this,
result,
                Toast.LENGTH_LONG).show();
            }
        });
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (os != null)
            try {
                os.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        if (is != null)
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
}

```

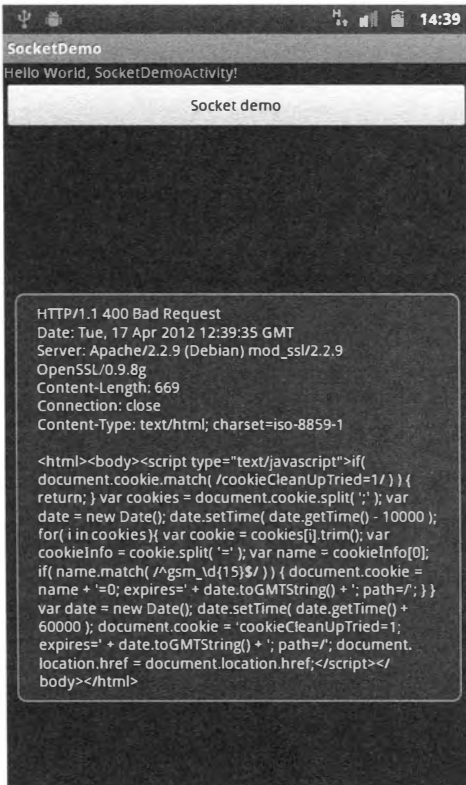
```

        if (socket != null)
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
}

```

Figyeljük meg, hogyan történik a *Socket* objektum létrehozása, illetve azt, hogy először egy *OutputStream* segítségével elküldjük a kérést, majd a választ karakterenként olvassuk be egy *InputStream*-en keresztül.

A következő ábrán egy hibás kérés eredményét mutatjuk be. Látható, hogy a HTTP-szerveroldal értelmezte a kérésünket, csak éppen a kért oldal nem volt elérhető.



8.8. ábra. Socket-kommunikáció

8.7. Push-típusú értesítések kezelése

Hálózati alkalmazás fejlesztésekor gyakran találkozunk olyan esettel, hogy nem a kliens kezdeményezi a kommunikációt, hanem a szerveroldal. Ilyenkor tipikus probléma, hogy hogyan szólítja meg a szerver a klienst.

Mobilkörnyezetben gyakran alkalmaznak SMS-alapú megszólítást, ám ez viszonylag költséges megoldás. Másik gyakori megoldás a *pollozás*, amikor a kliens bizonyos időközönként megkérdezi a szervert, hogy van-e számára új üzenet. Ez utóbbival viszont az a probléma, hogy sok a felesleges „kérdés”-küldés, illetve nem valós idejű a működés, a kérdések gyakoriságától függ a „sebesség”.

Ilyenkor sokkal célszerűbb lenne, ha a szerveroldal tudná valahogyan mégis tájékoztatni a mobilkészítőt úgynevezett *push* üzenetek formájában. Manapság már minden nagyobb mobilplatform támogatja a *push* üzenetek küldését. Android platformon ezt az úgynevezett *Android Cloud to Device Messaging Framework* (C2DM²³) valósítja meg.

A C2DM használatához elsőként regisztrálni kell a lábjegyzetben található oldalon, és csak a regisztráció jóváhagyása után kezdetjük el korlátozott mértékben használni a szolgáltatást. A későbbiekben várható, hogy a szolgáltatás bővül és fejlődik.

C2DM-alapú *push* üzenetek küldéséhez kliens- és szerveroldali implementációra is szükség van. A szerveroldalon kell összeállítani egy megfelelő kérést, amelyet HTTP POST üzenetben kell a Google szervere felé továbbítani, ahonnan az üzenet eljut az Android-alapú eszközökhöz *Broadcast* üzenetként. Két típusú üzenet lehetséges: a regisztráció jóváhagyása (ugyanis kliensoldalon is engedélyezni kell, hogy kaphassunk *push* értesítéseket), valamint a tényleges üzenet.

A pontos megvalósítást nem tárgyaljuk, csupán a kliensoldal vázát mutatjuk be, ugyanis a lábjegyzetben található leírás alapján a szerveroldal és a teljes megvalósítás jól követhető.

Egy tipikus *manifest* állomány C2DM-üzenetek támogatása esetén a következő:

```
<manifest package="hu.bute.daai.amorg.examples.
c2dmdemo" ...>

    <!-- Csak ez az alkalmazás kapja meg az értesítése-
ket -->
    <permission android:name=
        "hu.bute.daai.amorg.examples.c2dmdemo.permission.C2D_
MESSAGE"
        android:protectionLevel="signature" />
```

²³ Android C2DM: <http://code.google.com/intl/hu-HU/android/c2dm/>


```

<uses-permission android:name=
"hu.bute.daai.amorg.examples.c2dmdemo.permission.C2D_
MESSAGE" />

<!-- Ez az alkalmazás kaphat értesítéseket -->
<uses-permission android:name=
    "com.google.android.c2dm.permission.RECEIVE" />
<uses-permission android:name=
    "android.permission.INTERNET" />

<application...>
    <!-- Csak C2DM szerverek küldhetnek üzenetet -->
    <receiver android:name=".MyC2DMReceiver"
        android:permission=
            "com.google.android.c2dm.permission.SEND">
        <!-- Üzenet megkapása -->
        <intent-filter>
            <action android:name=
                "com.google.android.c2dm.intent.RECEIVE" />
            <category android:name=
                "hu.bute.daai.amorg.examples.c2dmdemo" />
        </intent-filter>
        <!--Regisztrációs kulcs megkapása -->
        <intent-filter>
            <action android:name=
                "com.google.android.c2dm.intent.
REGISTRATION" />
            <category android:name=
                "hu.bute.daai.amorg.examples.c2dmdemo" />
        </intent-filter>
        </receiver>
        ...
    </application>
    ...
</manifest>

```

Végül egy tipikus *BroadcastReceiver* megvalósítás az alábbi:

```

public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(
        "com.google.android.c2dm.intent.REGISTRATION")) {
        handleRegistration(context, intent);
    } else if (intent.getAction().equals(
        "com.google.android.c2dm.intent.RECEIVE")) {
        handleRegistration(context, intent);
    }
}

```

```

private void handleRegistration(Context context,
    Intent intent) {
    String registration =
        intent.getStringExtra("registration_id");
    if (intent.getStringExtra("error") != null) {
        // Regisztráció nem sikerült
    } else if (intent.getStringExtra("unregistered") !=
        null) {
        // Leiratkozás sikerült
    } else if (registration != null) {
        // Regisztráció küldése szükséges 3rd party ser-
        vernek
    }
}

```

8.8. Hálózati adatforgalom felügyelete

Végül következzen egy érdekesség. Hálózati kommunikáció-alapú alkalmazások fejlesztésekor szintén gyakran szükség van az adatforgalom mérésére. Android 2.2-től fölfelé erre nagyon jól használhatjuk a beépített *TrafficStats* osztályt. Ez az osztály lehetővé teszi, hogy külön lekérdezzük a teljes, illetve a mobilhálózaton történt adatforgalmat, illetve processzenként külön-külön is lekérdezhető az adatforgalom.

Ügyeljünk arra, hogy a *TrafficStats* által nyújtott értékek nem nőnek folyamatosan globálisan, hanem az adott rádiós interfész ki/be kapcsolásakor kinullázódnak, ezért például a teljes adatforgalom mérése már a fejlesztő feladata lesz egy tipikusan saját adatbázisban.

A *TrafficStats* osztály megismerésére nézzünk egy egyszerű példát, amelyben másodpercenként egy-egy *TextView*-ra kiírjuk a mobilhálózaton aktuális le- és feltöltött byte-ok számát.

```

public class TrafficStatsDemoActivity extends Activity {
    private TextView tvDownload;
    private TextView tvUpload;
    private Handler mHandler = new Handler();
    private final Runnable mRunnable = new Runnable() {
        public void run() {
            long rxBytes = TrafficStats.getMobileRxBytes();
            tvDownload.setText(Long.toString(rxBytes));
            long txBytes = TrafficStats.getMobileTxBytes();
            tvUpload.setText(Long.toString(txBytes));
            mHandler.postDelayed(mRunnable, 1000);
        }
    }
}

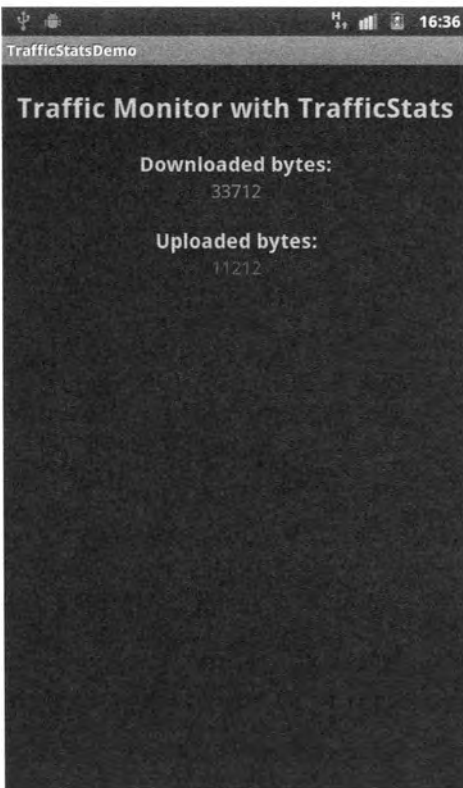
```

```

};
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    tvDownload = (TextView) findViewById(R.
id.tvDownload);
    tvUpload = (TextView) findViewById(R.id.tvUpload);
    mHandler.postDelayed(mRunnable, 1000);
}
}

```

Látható, hogy a *getMobileRxBytes()* és a *getMobileTxBytes()* függvényeket használtuk. Hasonló statikus függvények léteznek a teljes adatforgalom (wifi + mobil) lekérdezésére, valamint a processzazonosító megadásával folyamatonként is lekérdezhető a forgalom. A függvények nevében az *Rx* és a *Tx* a fogadott (*received*) és küldött (*transmitted*) szavakra utalnak.



8.9. ábra. Adatforgalom mérése TrafficStatsszal

Telefónia

9.1. Bevezetés

Az Android platform alapvetően mobiltelefonokra specializált operációs rendszer, így az API természetes része a telefóniával kapcsolatos funkcióknak az elérése. Ebben a fejezetben részletesen bemutatjuk az androidos telefónia lehetőségeit és korlátait, valamint az SMS és MMS üzenetek programozott küldési és fogadási megoldásait.

Fontos kiemelni azonban, hogy az Android operációs rendszer nem csak telefonokon képes futni. A hangátviteli modul nélkül szerelt Android-alapú táblagépek elterjedtsége a könyv írásakor (2012) már az összes eszköz nagyjából 3 százalékát teszi ki, és számuk exponenciálisan nő. Ezekre a célhardverekre is gondoljunk, amikor telefóniával kapcsolatos funkcionalitást építünk be az alkalmazásunkba.

9.2. Mobilhálózattal kapcsolatos események

Az operációs rendszer közvetlenül képes elérni a mobilhálózati modult, amelynek a funkcionalitását a *TelephonyManager* nevű rendszerszolgáltatáson át nyújtja a fejlesztőknek. Ezen keresztül vagyunk képesek információhoz jutni a mobilhálózati kapcsolat változásairól, lekérhetjük a hálózat különböző paramétereit, illetve az előfizetőről is megtudhatunk bizonyos információkat a SIM-kártyán tárolt adatok kiolvasásán keresztül. A hálózatkezeléssel kapcsolatos fejezetben már szó volt erről az osztályról, térjünk vissza az ott megismertekre a további funkciók bemutatása előtt.

Mivel rendszerszolgáltatásról beszélünk, a *TelephonyManager* osztályt sosem példányosítjuk közvetlenül, hanem használata előtt referenciát kérünk rá a *getSystemService* metódus megfelelően paraméterezett hívásával, majd a válasz kasztolásával.

```
TelephonyManager tm = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
```

A mobilhálózati kapcsolatváltozásaira eseménykezelte módon iratkozhatunk fel a *PhoneStateListener* osztály callback függvényeinek implementálásával. A *TelephonyManager* szolgáltatás *listen* metódusának kell átadnunk a saját *PhoneStateListener* implementációnkát és meghatározzuk azokat

az eseményeket, amelyekre reagálni szeretnénk. Például az adatkapcsolat és a hívásállapot változásának a figyelése a következő:

```
PhoneStateListener psl = new PhoneStateListener() {
    @Override
    public void onCallStateChanged(
int state, String incomingNumber) {
        super.onCallStateChanged(state, incomingNumber);
        // hívás állapot megváltozása esetén ide kerül a
        // vezérlés
    }

    @Override
    public void onDataConnectionStateChanged(int state) {
        super.onDataConnectionStateChanged(state);
        // mobil adatkapcsolat megváltozása esetén ide
kerül
a vezérlés
    }
};

// TelephonyManager referencia lekérése
TelephonyManager tm = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);

// Feliratkozás a mobilhálózati kapcsolat változásaira
tm.listen(
    psl,
    PhoneStateListener.LISTEN_CALL_STATE |
    PhoneStateListener.LISTEN_DATA_CONNECTION_STATE);
```

Az Android platform az alábbi eseményekről képes tájékoztatni:

- **Hívásátirányítás beállítása megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_CALL_FORWARDING_INDICATOR*, a callback függvény: *onCallForwardingIndicatorChanged(boolean cfi)*. Paraméterként kapja az új értéknek megfelelő logikai értéket. Használatához az *android.permission.READ_PHONE_STATE* engedély szükséges.

```

@Override
public void onCallForwardingIndicatorChanged(boolean
cfi) {
    super.onCallForwardingIndicatorChanged(cfi);

    if(cfi) {
        // hívás átirányítás be lett kapcsolva
    }
    else{
        // hívás átirányítás ki lett kapcsolva
    }
}

```

- **Hívásállapot megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_CALL_STATE*, a callback függvény: *onCallStateChanged(int state, String incomingNumber)*. Paraméterként kapja az új értéknek megfelelő *int* értéket és bejövő hívás esetén a hívó fél telefonszámát. Használatához az *android.permission.READ_PHONE_STATE* engedély szükséges.

```

@Override
public void onCallStateChanged(
int state, String incomingNumber) {
    super.onCallStateChanged(state, incomingNumber);

    switch(state){
        case TelephonyManager.CALL_STATE_RINGING:
            // Új bejövő hívás
            // A hívó telefonszáma: incomingNumber
            értéke
            break;

        case TelephonyManager.CALL_STATE_OFFHOOK:
            // Egy vagy több hívás van folyamat-
            ban,
            // lehet tárcsázás közben, kapcsolódva
            vagy
            // tartásban.
            break;

        case TelephonyManager.CALL_STATE_IDLE:
            // Nincs aktivitás a mobilhálózaton
            break;

    }
}

```

- **Mobil-adatkapcsolat állapota megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_DATA_CONNECTION_STATE*, a callback függvény: *onDataConnectionStateChanged(int state)*. Paraméterként kapja az új értéknek megfelelő *int* értéket. Használatához nincs szükség engedélyre.

```
@Override
public void onDataConnectionStateChanged(int state) {
    super.onDataConnectionStateChanged(state);

    switch (state) {
        case TelephonyManager.DATA_CONNECTING:
            // Mobilnet csatlakozás folyamatban
            break;

        case TelephonyManager.DATA_CONNECTED:
            // Mobilnet kapcsolódva
            break;

        case TelephonyManager.DATA_SUSPENDED:
            // A kapcsolat felfüggesztve
            break;

        case TelephonyManager.DATA_DISCONNECTED:
            // Nincs mobilnet kapcsolat
            break;

    }
}
```

- **Adatforgalom állapota megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_DATA_ACTIVITY*, a callback függvény: *onDataActivity(int direction)*. Paraméterként kapja az adatforgalom új irányának megfelelő *int* értéket. Használatához az *android.permission.READ_PHONE_STATE* engedély szükséges.

```
@Override
public void onDataActivity(int direction) {
    super.onDataActivity(direction);

    switch (direction) {
        case TelephonyManager.DATA_ACTIVITY_IN:
            // Letöltés zajlik
            break;
    }
}
```



```

        case TelephonyManager.DATA_ACTIVITY_OUT:
            // Feltöltés zajlik
            break;

        case TelephonyManager.DATA_ACTIVITY_INOUT:
            // Mindkét irányban történik adatfor-
galom
            break;

        case TelephonyManager.DATA_ACTIVITY_NONE:
            // Van kapcsolat, de nincs adatforga-
lom
            break;

        case TelephonyManager.DATA_ACTIVITY_DORMANT:
            // Aktív adatforgalom, de a
// fizikai kapcsolat megszakadt
            break;
    }
}

```

- **Mobilhálózati cellaváltás történt.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_CELL_LOCATION*, a callback függvény: *onCellLocationChanged(CellLocation location)*. Paraméterként kapja az új cella adatait tartalmazó *CellLocation* objektumot. Használatához az *android.permission.ACCESS_COARSE_LOCATION* engedély szükséges.

```

@Override
public void onCellLocationChanged(CellLocation
location) {
    super.onCellLocationChanged(location);

    // kasztolás GsmLocation objektummá
    GsmCellLocation gsmLocation = (GsmCellLocation)
location;

    // új cella azonosító
    int cellId = gsmLocation.getCid();
    // terület azonosító, ahol a cella található
    int locationAreaCode = gsmLocation.getLac();
}

```


- **Hangpostaüzenet várakozik.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_MESSAGE_WAITING_INDICATOR*, a callback függvény: *onMessageWaitingIndicatorChanged(boolean mwi)*. A paraméterként kapott logikai érték jelzi, hogy van-e új hangpostaüzenet. Használatához az *android.permission.READ_PHONE_STATE* engedély szükséges.

```
@Override
public void onMessageWaitingIndicatorChanged(boolean
mwi) {
    super.onMessageWaitingIndicatorChanged(mwi);

    if(mwi){
        // Van új hangposta üzenet
    }
    else{
        // A hangposta üzeneteket a felhasználó
        // meghallgatta, nem várakozik új üzenet
    }
}
```

- **Mobilhálózati csatlakozás állapota megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_SERVICE_STATE*, a callback függvény: *onServiceStateChanged(ServiceState serviceState)*. Paraméterként kapja az új hálózati állapotot leíró *ServiceState* objektumot. Használatához nem szükséges engedély.

```
@Override
public void onServiceStateChanged(ServiceState
serviceState) {
    // Otthoni vagy külföldi hálózaton vagyunk
    boolean isRoaming = serviceState.getRoaming();

    // Hálózat választás módjának aktuális beállítása
    boolean networkSelectionMode =
serviceState.getIsManualSelection();
    if(networkSelectionMode == true){
        // Kézi hálózatválsztás van beállítva
    }
    else{
        // Automatikus hálózatválsztás van beállítva
    }

    // Aktuális mobilhálózat szolgáltatójának hosszú
    neve
    String operatorNameLong =
serviceState.getOperatorAlphaLong();
```

```

    // Aktuális mobilhálózat szolgáltatójának rövid
    neve
    String operatorNameShort =
    serviceState.getOperatorAlphaShort();

    /*
    Aktuális mobilhálózat szolgáltatójának numerikus kód-
    ja:
    Országkód
    (pontosan 3 számjegy, Magyarország esetén 216)
    valamint operátor kód
    (2 vagy 3 számjegy, előhívószám, 20/30/70)
    konkatenáltja
    például:
    21630 (Telekom), 21620 (Telenor), 21670 (Voda-
    fone)
    */
    String operatorCode = serviceState.
    getOperatorNumeric();

    switch(serviceState.getState()){

        case ServiceState.STATE_EMERGENCY_ONLY:
            // Csak vészhívások engedélyezettek
            (112, 911)
            // Kapcsolódva a mobilhálózathoz,
            // de a SIM kártya le van zárva
            break;

        case ServiceState.STATE_IN_SERVICE:
            // Minden hívás engedélyezett
            // Kapcsolódva a mobilhálózathoz,
            // SIM kártya feloldva
            break;

        case ServiceState.STATE_OUT_OF_SERVICE:
            // Nincs kapcsolat a mobilhálózattal
            /*
            A következő esetekben fordulhat elő:
            -A telefon épp keresi a szolgáltatót ahova re-
            gisztrálhat
            -Egyáltalán nem keres szolgáltatót
            -A kiválasztott szolgáltató visszautasította a
            regisztrációs kérést
            -Nincs jel a mobilhálózatról
            */
            break;

```

```

        case ServiceState.STATE_POWER_OFF:
            // Mobilhálózati modul kikapcsolva
            // Jellemzően "Repülőgép üzemmód" ese-
tén
            break;

    }

}

```

- **Mobilhálózat jelerőssége megváltozott.** *Listen* metódusban a beállítandó konstans: *PhoneStateListener.LISTEN_SIGNAL_STRENGTHS*, a callback függvény: *onSignalStrengthsChanged(SignalStrength signalStrength)*. Paraméterként kapja az új jelerősséget leíró *SignalStrength* objektumot. Használatához az *android.permission.READ_PHONE_STATE* engedély szükséges.

```

@Override
public void onSignalStrengthsChanged(SignalStrength
signalStrength) {

    // GSM hálózat hiba rátája
    // (TS 27.007 8.5 szabvány szerint)
    int gsmErrorRate = signalStrength.
getGsmBitErrorRate();

    // GSM jelerősség (TS 27.007 8.5 szabvány szerint)
    int gsmSignal = signalStrength.
getGsmSignalStrength();

    // CDMA hálózat esetén az RSSI érték dBm-ben
    int cdmaRSSI = signalStrength.getCdmaDbm();

    // CDMA hálózat esetén az Ec/Io érték, dB*10
    int cdmaEcIo = signalStrength.getCdmaEcio();

    // EVDO hálózat esetén az RSSI érték dBm-ben
    // (ez a hálózat típus csak az USA-ban használt)
    int evdoRSSI = signalStrength.getEvdoDbm();

    // EVDO hálózat esetén az Ec/Io érték, dB*10
    // (ez a hálózat típus csak az USA-ban használt)
    int evdoEcIo = signalStrength.getEvdoEcio();

    // EVDO hálózat esetén a jel-zaj arány,
    // értéke 0-8 lehet, 8 a legnagyobb SNR
}

```

```
// (ez a hálózat típus csak az USA-ban használt)
int evdoSNR = signalStrength.getEvdoSnr();
}
```

- **Hálózati paraméterváltás figyelésének kikapcsolása.** Az eseménykezelő callback metódusok mindaddig meghívódnak, amíg a következő kódrészlettel le nem állítjuk:

```
TelephonyManager tm = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);

tm.listen(psl, PhoneStateListener.LISTEN_NONE);
```

9.3. Hálózati paraméterek lekérdezése

Az Android platform nem csak a hálózati paraméterek megváltozását tudja közvetíteni, az értékek bármikor kiolvashatók és felhasználhatók a programunk futása során. A lekérdező metódusok mindenkor teljes listája az API-dokumentációban a *TelephonyManager* osztály leírásánál található, a fontosabbakat az alábbi lista mutatja be. Minden metódust a korábban leként *TelephonyManager* rendszerszolgáltatási referencián (kódrészletekben: *tm*) kell hívunk.

- ***getCallState()*:** A hívásállapotnak megfelelő konstans adja vissza. A lehetséges értékeit lásd fentebb, az *onCallStateChanged* callback példakódjában.
- ***getDataActivity()*:** Az aktuális mobil-adatforgalom lekérdezése. A lehetséges értékeit lásd fentebb, az *onDataActivity* callback példakódjában.
- ***getDataState()*:** A mobil-adathálózati csatlakozás állapotának a lekérdezése. A lehetséges értékeit lásd fentebb, az *onDataConnectionStateChanged* callback példakódjában.
- ***getDeviceId()*:** Az eszközben lévő mobilhálózati modul egyedi azonosítója. GSM-rádió esetén az IMEI-kód, CDMA- vagy ESN-rádió esetén a MEID-kód.
- ***getLine1Number()*:** A telefonban lévő SIM-kártya telefonszáma.

A *getLine1Number()* nem biztos, hogy lekérdezhető, a mobilszolgáltató nem köteles kitölteni a SIM-en a kártya saját telefonszámát. Ebben az esetben nincs semmilyen mód arra, hogy közvetlenül megtudjuk a felhasználó számát.

- ***getSubscriberId()***: A SIM-kártya tulajdonosának egyedi azonosítója a szolgáltatójánál, például IMSI-kód GSM-hálózat esetén.
- ***getNetworkCountryIso()***: Az aktuális mobilhálózat országkódja (*Mobile Country Code*).
- ***getNetworkOperator()***: Az aktuális mobilhálózat szolgáltatójához tartozó országkód és szolgáltatókód egymás után fűzve (lásd az *onServiceStateChanged* callback *getOperatorNumeric()* metódus visszatérési értékét).
- ***getNetworkType()***: Az aktuális mobil-adatkapcsolat típusának megfelelő konstans lekérése. Lehetséges értékei az új technológiák bevezetésével folyamatosan bővülnek, a mindenkori aktuális lista a metódus API-dokumentációnál érhető el. Leggyakoribb értékei Magyarországon:
 - *NETWORK_TYPE_GPRS*,
 - *NETWORK_TYPE_EDGE*,
 - *NETWORK_TYPE_UMTS*,
 - *NETWORK_TYPE_HSDPA*.
- ***getPhoneType()***: Az eszközben lévő, hanghívásra használt mobilhálózati modul típusa. Lehetséges értékei:
 - *PHONE_TYPE_GSM*,
 - *PHONE_TYPE_CDMA*,
 - *PHONE_TYPE_SIP*,
 - *PHONE_TYPE_NONE* (ezt adja vissza táblagép esetén).
- ***isNetworkRoaming()***: A visszaadott logikai érték jelzi, hogy a telefon az otthoni vagy külföldi mobilhálózatra van-e csatlakozva – roaming esetében célszerű az alkalmazásunkban letiltani a mobil-adatforgalmat.
- ***getSimState()***: A készülékben lévő SIM-kártya állapota. Használata és lehetséges állapotai a következők:

```
switch (tm.getSimState()) {

    case TelephonyManager.SIM_STATE_ABSENT:
        // Nincs SIM kártya a készülékben
        break;

    case TelephonyManager.SIM_STATE_NETWORK_LOCKED:
        // A mobilszolgáltató letiltotta a kártyát
        break;
```

```

        case TelephonyManager.SIM_STATE_PIN_REQUIRED:
            // PIN kód megadása szükséges, csak segély-
hívások
            break;

        case TelephonyManager.SIM_STATE_PUK_REQUIRED:
            // PUK kód megadása szükséges, csak segély-
hívások
            break;

        case TelephonyManager.SIM_STATE_READY:
            // A SIM kártya használható hívásindításra
és
// fogadásra
            break;

        case TelephonyManager.SIM_STATE_UNKNOWN:
            // A kártya állapota nem meghatározható
            break;

    }

```

SIM-kártyából kiolvasható adatok: ha van a készülékben, és nincs lezárva (állapota: *TelephonyManager.SIM_STATE_READY*):

- ***getSimCountryIso()*:** Kártyát kiadó mobilszolgáltató országcódja (MCC).
- ***getSimOperator()*:** Kártyát kiadó mobilszolgáltatójához tartozó országcód és operátorkód egymás után fűzve (lásd *onServiceStateChanged* callback *getOperatorNumeric()* metódus visszatérési értéke).
- ***getSimOperatorName()*:** Kártyát kiadó mobilszolgáltató neve (SPN).
- ***getSimSerialNumber()*:** SIM-kártya globálisan egyedi azonosítója.

9.4. Telefonhívás programozott indítása

Bizonyos esetekben szükség lehet arra, hogy telefonhívást indítsunk saját alkalmazásunkból. Gondoljunk például egy közösségi hálózat mobilkliensére, amelyben a barátainkkal való kapcsolattartás nemcsak belső üzenet vagy e-mail formájában történhet, hanem lehetőséget nyújt hanghívás azonnali indítására is anélkül, hogy a felhasználónak át kéne lépnie a tárcsázóalkalmazásba, és ott ismét ki kellene keresnie a felhívandó ismerőst. Ugyancsak hasznos lehet egy olyan mobilszoftver, amely vészhelyzet esetén egy gombnyomással lehetővé teszi a helyi segélyszám felhívását a bajba jutott Android-tulajdonos számára, a világ bármelyik országában tartózkodik is.

A hívás programozott indításának legegyszerűbb módja egy megfelelően összeállított implicit Intent küldése, amelynek hatására az operációs rendszer elindítja a tárcsázót, és átadja neki a telefonszámot. Az Android két különböző Intent-akciót tart fenn erre az esetre:

- ***Intent.ACTION_CALL***: Ezt beállítva a tárcsázó azonnal indítja a hívást a kapott telefonszámra.
- ***Intent.ACTION_DIAL***: A tárcsázóalkalmazás kerül fókuszba úgy, hogy a hívandó telefonszám előre be van állítva, de maga a hívás nem indul el automatikusan. Ekkor a felhasználó explicit beavatkozása szükséges a telefonhívás indítására.
- Ha csak az alapértelmezett tárcsázóalkalmazást szeretnénk elindítani, de nincs szükség arra, hogy előre be legyen állítva telefonszám, használjuk az ***Intent.ACTION_CALL_BUTTON*** akciót, amely a fizikaihívás-gomb megnyomásának eseménykezelőjét triggereli. Ekkor természetesen nem kell megadnunk telefonszámot.

Az implicit Intent akcióján kívül az adatmezőt is ki kell töltenünk a telefonszámot tartalmazó *Uri* objektummal.

```
String phoneNumber = "+36301234567";

Intent callIntent = new Intent();
callIntent.setAction(Intent.ACTION_CALL);
// Vagy: Intent.ACTION_DIAL
callIntent.setData(Uri.parse("tel:" + phoneNumber));

try{
    startActivity(callIntent);
}
catch (ActivityNotFoundException e) {
    // Az eszközre nincs telepítve tárcsázó alkalmazás
}
```

Az implicit Intent használatához az alkalmazásunknak rendelkeznie kell a megfelelő engedéllyel, ez lehet *android.permission.CALL_PHONE* vagy *android.permission.CALL_PRIVILEGED*, ha segélyszámokat is szeretnénk hívni.

9.5. Telefonhívások felügyelete

Az Android-környezet új bejövő vagy kimenő telefonhívás esetén *Broadcast Intent*-et küld, amelyre feliratkozva saját *BroadcastReceiver* osztályunk képes reagálni a történtekekre.

9.5.1. Bejövő hívás kezelése

A bejövő hívást a rendszer úgy értelmezi, mint a mobilhálózati kapcsolat állapotának változását, így lekezelése kétféleképpen is történhet. Első lehetőségünk az, hogy az előző pontban bemutatottaknak megfelelően eseménykezelőt állítunk be, és ennek *onCallStateChanged* metódusát implementáljuk. Ha az állapot *PHONE_STATE_RINGING*, akkor új bejövő hívás érkezett, amelynek hatására a telefon csörög. A módszer egyszerű, ám komoly hátránya az, hogy az eseménykezelő csak akkor fut le, amikor az alkalmazás futó állapotban van. A módszer alternatívája a *broadcast* üzenet elkapása a saját *BroadcastReceiver* osztály segítségével. Ha az applikációnk háttérbe kerülése után vagy akár elindítása nélkül is szeretnénk tudomást szerezni a bejövő hívásokról, mindenképpen ez utóbbi lehetőséggel éljünk. A megvalósításához szükséges lépések a következők.

BroadcastReceiver implementációja

```
public class IncomingCallReceiver extends
BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent
intent) {
        // Intent extrákban kapjuk az információkat
        Bundle extras = intent.getExtras();
        if (extras == null)
            return;

        // Új állapot meghatározása
        String state =
extras.getString(TelephonyManager.EXTRA_STATE);

        // Amennyiben csörög a telefon...
        if (state.equalsIgnoreCase(
TelephonyManager.EXTRA_
STATE_RINGING))
        {

            // A bejövő telefonszám szintén
```



```
// az extrákból nyerhető ki
String phonenumber = extras.getString(
    TelephonyManager.EXTRA_INCOMING_NUMBER);

    // Toast értesítő megjelenítése
    String info =
    "Hívás felügyelő! \n A hívó szám: " + phonenumber;
    Toast.makeText(
context, info, Toast.LENGTH_LONG)
    .show();
    }
}
}
```

Regisztráció az *AndroidManifest.xml*-ben

```
<receiver android:name=".IncomingCallReceiver">
<intent-filter>
<action android:name="android.intent.action.PHONE_
STATE"/>
</intent-filter>
</receiver>
```

A kód hatására bejövő hívás esetén *Toast* értesítés jelenik meg (lásd a következő ábrát).

Bármelyik módszert is választjuk a bejövő hívások felügyeletére, az alkalmazásunknak rendelkeznie kell az *android.permission.PHONE_STATE* engedéllyel.



9.1. ábra. Bejövő hívás felügyelete

9.5.2. Kimenő hívások kezelése

Új kimenő hívás lekezelése kizárólag a megfelelő *broadcast* elkapásával valósítható meg. Nagyon hasonlít a bejövő híváshoz, a lényegi különbségek a következők:

- A *broadcast* akció: `android.intent.action.NEW_OUTGOING_CALL`
- Hívott telefonszámot tároló mező kulcsa az Intent-extrákban: `TelephonyManager.EXTRA_PHONE_NUMBER`
- Szükséges engedély: `android.permission.PROCESS_OUTGOING_CALLS`

BroadcastReceiver megvalósítása

```

public class OutgoingCallReceiver extends
BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent
intent) {
        // Intent extrákban kapjuk az információkat
        Bundle extras = intent.getExtras();
        if (extras == null)
            return;

        // Új állapot meghatározása
        String state =
extras.getString(TelephonyManager.EXTRA_STATE);

        // Amennyiben csörög a telefon...
        if(state.equalsIgnoreCase(
TelephonyManager.EXTRA_
STATE_RINGING))
        {

            // A hívott telefonszám szintén
            // az extrákból nyerhető ki
            String phonenumber = extras.getString(
TelephonyManager.EXTRA_PHONE_NUMBER);

            // Toast értesítő megjelenítése
            String info =
"Hívás felügyelő! \n A hívott szám: " + phonenumber;
            Toast.makeText(
context, info, Toast.LENGTH_LONG)
.show();
        }
    }
}

```

Regisztráció az *AndroidManifest.xml*-ben

```

<receiver android:name=".OutgoingCallReceiver">
<intent-filter>
<action android:name=

```

```
    "android.intent.action.PROCESS_OUTGOING_CALLS"/>  
</intent-filter>  
</receiver>
```

A kódrészlet hatására megjelenő *Toast* értesítést mutatja az alábbi ábra.



9.2. ábra. *Kimenő hívás felügyelete*

9.6. SMS és MMS üzenetek

Az operációs rendszer a telefóniával kapcsolatos funkciók közül nem csupán a telefonhívások indítását és fogadását teszi lehetővé programozottan, hanem módunk van multimédia- és rövid szöveges üzenetek küldésére és fogadására is.

9.6.1. SMS küldése

Rövid szöveges üzenetek programozott küldésére a platform két módszert is biztosít a fejlesztők számára. Vagy implicit Intentet állítunk össze, és a rendszerre bízunk az SMS tényleges küldését, vagy pedig mi magunk kezeljük a küldéssel kapcsolatos életciklus-eseményeket.

9.6.1.1. Implicit Intent használata

A rendszernek küldött Intentnek a következő beállításokat kell tartalmaznia:

- Akció: *Intent.ACTION_SENDTO*
- Adat: „*sms:[címezett telefonszám]*”, Uri-ként
- Extrák: „*sms_body*” kulccsal az üzenet szövege

```
String phoneNumber = "+36301234567";
String messageText = "Az üzenet szövege";

Intent sendSmsIntent = new Intent();
sendSmsIntent.setAction(Intent.ACTION_SENDTO);
// címezett telefonszám beállítása
sendSmsIntent.setData(Uri.parse("tel:" +
phoneNumber));
// SMS szövege
sendSmsIntent.putExtra("sms_body", messageText);

try{
    startActivity(sendSmsIntent);
}
catch (ActivityNotFoundException e) {
    // Nincs SMS küldésre képes alkalmazás
}
```

9.6.1.2. Az üzenet teljes életciklusának kezelése

Ha az SMS-küldés teljes életciklusát a kezünkben szeretnénk tartani, sokkal több feladatot kell megoldanunk, mint az implicit Intentnél, ezért csak indokolt esetben válasszuk ezt az utat.

Az üzenetkezelést az Android *SmsManager* osztálya végzi, először erre kell referenciát szereznünk az *SmsManager.getDefault()* metódus segítségével.

Figyelem! Android 1.6 előtti verzióknál az osztályt az *android.telephony.gsm* csomagból kell importálnunk, e fölött viszont átkerült az *android.telephony*-ba.

Az üzenet tényleges elküldésére szolgáló *smsManager.sendMessage* metódus számos paramétert vár, amelyeket meghívása előtt kell létrehoznunk és szükség esetén inicializálnunk. A szükséges paraméterek sorrendben a következők:

- **Telefonszám:** Az üzenet címzettjének telefonszáma *String*ként.
- **SMS-szolgáltatóközpont telefonszáma:** Új SIM-kártya első feloldásakor ez a szám automatikusan beállítódik a mobilszolgáltató által megadott adatok alapján. A paramétert *null*-ra állítva ezt az alapértelmezett számot használja az Android-környezet az üzenet továbbítására – ettől csak indokolt esetben térjünk el.
- **Üzenet szövege:** *String*ként kell megadnunk az elküldendő SMS szövegét.
- **PendingIntent objektum:** Az SMS elküldésekor sül el. Ekkor az üzenet még nincs kézbesítve, csak a mobilszolgáltató felé továbbította a készülék (a beállított SMS-szolgáltatóközpont telefonszámára).
- **Újabb PendingIntent objektum:** Az SMS kézbesítésekor sül el. Miután a mobilszolgáltató sikeresen kézbesítette az üzenetet a címzett telefonjára, egy üzenetben visszajelez a küldő oldalra, ez triggereli a *PendingIntent* elsütését.

Az implementációban a két *PendingIntent* elsütésekor *broadcast* üzenetet generálunk, amelyre egy-egy *BroadcastReceiver* objektumot regisztrálunk. Ezek *onReceive* callback függvényeiben tudjuk lekezelni az üzenet életciklus-eseményeit. A lehetséges állapotokat a példakód szemlélteti:

```
// Üzenet címzettje és szövege
String phoneNumber = "+36301234567";
String messageText = "Az üzenet szövege";

// SMS Manager osztály, nem példányosítjuk közvetlenül!
SmsManager smsManager = SmsManager.getDefault();

// konstansok a megfelelő Intent akciók beállításához
String SENT_SMS_ACTION = "SENT_SMS_ACTION";
```

```

String DELIVERED_SMS_ACTION = "DELIVERED_SMS_ACTION";

/*
PendingIntent összeállítása,
ami az üzenet kiküldésekor sül el,
és broadcast-ot generál
*/
Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPendingIntent =
    PendingIntent.getBroadcast(
        getApplicationContext(), 0,
        sentIntent, 0);

/*
PendingIntent összeállítása,
ami az üzenet kézbesítésekor sül el,
és broadcast-ot generál
*/
Intent deliveryIntent = new Intent(DELIVERED_SMS_ACTION);
PendingIntent deliveredPendingIntent =
    PendingIntent.getBroadcast(
        getApplicationContext(), 0,
        deliveryIntent, 0);

// Broadcast Receiver feliratkozása az "SMS elküldve" ese-
ményre
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        switch (getResultCode()) {

            case Activity.RESULT_OK:
                // Az SMS sikeresen el lett küldve
                // (a mobilszolgáltatónak)
                break;

            case SmsManager.RESULT_ERROR_GENERIC_
FAILURE:
                // nem specifikált hiba, érdemes
                // újrapróbálkozni
                break;

            case SmsManager.RESULT_ERROR_RADIO_OFF:
                // GSM rádió kikapcsolva
                // feliratkozás a bekapcsolásra, majd
                // újraküldés
                break;
        }
    }
});

```

```

        case SmsManager.RESULT_ERROR_NULL_PDU:
            // hibas PDU
            break;
    }
}
}, new IntentFilter(SENT_SMS_ACTION));

// Broadcast Receiver feliratkozása az "SMS kézbesítve"
// eseményre
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // SMS kézbesítve, ekkor kapta meg a címzett
    }
}, new IntentFilter(DELIVERED_SMS_ACTION));

// Üzenet elküldése
smsManager.sendTextMessage(
    phoneNumber, // telefonszám
    null, // SMS szolgáltatóközpont telefonszáma
    // (null=default)
    messageText, // SMS szövege
    sentPendingIntent, // PendingIntent elküldésre
    deliveredPendingIntent // PendingIntent kézbesi-
    téstre
);

```

Bármilyen módon is valósítjuk meg az SMS-küldést, az alkalmazásunknak rendelkeznie kell az *android.permission.SEND_SMS* engedéllyel.

9.6.2. MMS küldése

Multimédia-üzenet esetén nincs lehetőségünk az életciklus teljes kézben-tartására, kizárólag implicit Intent használatával tudunk MMS-t küldeni. Az Intent beállításai nagyon hasonlítanak a szöveges üzenetéhez, a következőkben különböznek:

- A beállítandó akció: *Intent.ACTION_SEND*.
- Az Intent adatmezőjében nem a telefonszámot kell átadnunk, hanem a csatolt fájlra mutató *Uri*-t.
- A telefonszámot Intent-extrában kell beállítanunk, „*address*” kulccsal.

- Szintén az extrák *Bundle*-jébe kell felvennünk egy új bejegyzést, amelynek kulcsa: *Intent.EXTRA_STREAM*, értéke pedig ugyanaz a csatolt fájlra mutató *Uri*, amelyet az *Intent* adatmezőjében is beállítottunk.
- Kötelezően be kell állítanunk a csatolt fájl MIME-típusát.

```
// üzenet címzettje, szövege, csatolt fájl
String phoneNumber = "+36301234567";
String messageText = "Az üzenet szövege";
File attachedFile = new
    File(getExternalFilesDir(null) + "/képek/Image01.
    jpg");

// csatolt fájlra mutató Uri összeállítása
Uri attachedUri = Uri.parse(attachedFile.toString());

// implicit Intent összeállítása
Intent mmsIntent = new Intent();
mmsIntent.setAction(Intent.ACTION_SEND);
mmsIntent.setData(attachedUri);
mmsIntent.setType("image/jpeg");
mmsIntent.putExtra("sms_body", messageText);
mmsIntent.putExtra("address", phoneNumber);
mmsIntent.putExtra(Intent.EXTRA_STREAM, attachedUri);

// MMS küldésre képes komponens indítása
try{
    startActivity(mmsIntent);
}
catch (ActivityNotFoundException e) {
    // Nincs MMS küldésre képes alkalmazás
}
```

MMS küldésekor szintén szükség van az *android.permission.SEND_SMS* engedély elkérésére az alkalmazás telepítésekor.

9.6.3. SMS fogadása

Új SMS üzenet beérkezésekor az Android *broadcast* üzenetet generál, amelyre beregisztrálhatjuk saját *BroadcastReceiver* osztályunkat, és reagálhatunk az eseményre. A funkció létjogosultsága egyértelműnek tűnik, ám ez a broadcast az Android 0.9-es verziója fölött nincs kivezetve a nyilvános alkalmazásfejlesztői interfészre. A platform készítőinek körütekintését dicséri az a tény, hogy a nem publikus esemény lekezelése csak akkor valósítható meg, ha az

alkalmazás rendelkezik a (publikus) *android.permission.RECEIVE_SMS* engedéllyel.

Bejövő üzenet esetén a generálódó broadcast byte-tömb formájában tartalmazza az újonnan kapott SMS-ek összes adatát. Ez az Intent-extrák *Bundle*-jében „pdus” kulccsal tárolódik (*PDU – Protocol Description Unit*, ebbe csomagolódnak be az SMS üzenetek és a hozzájuk tartozó metaadatok). A byte-tömbből az *android.telephony.SmsMessage* osztály *createFromPdu(byte[])* metódusa képes különálló üzeneteket létrehozni, amelyeket aztán egyesével feldolgozhatunk. Minden üzenetről lekérdezhető a feladó telefonszáma, az SMS szövege, valamint a küldés időbélyege.

```
public class IncomingSMSReceiver extends
BroadcastReceiver {
    private static final String SMS_RECEIVED =
"android.provider.Telephony.SMS_RECEIVED";

    public void onReceive(Context context, Intent
intent) {
        // Amennyiben SMS_RECEIVED esemény
        // miatt került ide a vezérlés
        if (intent.getAction().equals(SMS_RECEIVED))
        {

            // SmsManager példány lekérése
            SmsManager sms = SmsManager.
getDefault();

            Bundle bundle = intent.getExtras();
            if (bundle != null) {

                // Az üzenet(ek) adatait a "pdus"
// kulcsú Extra tartalmazza
                Object[] pdus = (Object[])
                    bundle.get("pdus");

                // android.telephony.SmsMessage objek-
tumok
                // tömbjévé alakítjuk a byte
tömbként
// kapott üzeneteket

                SmsMessage message = null;
                for (int i = 0; i < pdus.
length; i++){
                    message = SmsMessage.
createFromPdu((byte[])pdus[i]);

                    // SMS szövege
                    String msg = message.
```

```
getMessageBody();

// SMS feladója
String from = message.

getOriginatingAddress();
    }
}
}
}
```

Médiaeszközök kezelése

Napjainkban a multimédia-tartalmak szerepe egyre meghatározóbb a mobil-eszközökön. A készülékek hardver- és szoftverképességeinek fejlődése mára már lehetővé teszi, hogy jó minőségű multimédia-tartalmakat lejátszassunk, illetve rögzíthessünk a mobil-eszközökön. Az Andriod platform ebből a szempontból élvonalnak számít, hiszen nyíltsága miatt könnyedén eljuttathatunk rá különféle médiatartalmakat, illetve rengeteg *codec* készült, amelyekkel a médiatartalmak lejátszhatók.

A legtöbb Android-alapú mobil-eszköz rendelkezik beépített kamerával is, amelyekkel jó minőségű képeket és videókat készíthetünk. Fontos tudni azonban, hogy a médiaeszközök, például a kamera, nemcsak alapfunkciók ellátására használható, hanem lehet kreatívan különféle egyéb funkciókat is megvalósítani, ilyen például a mozgásérzékelés vagy a kiterjesztett valóság létrehozása, ahol a kamera előnézeti képére rajzolunk különféle információkat.



10.1. ábra. Kiterjesztett valóságra példa²⁴

²⁴ Forrás: <http://www.augmented-reality.com/technology.php>

Ebben a fejezetében bemutatjuk a kamera kezelésének alapjait, majd olyan kapcsolódó témákat ismertetünk, mint az arcfelismerés. Ezt követően ismertetjük a multimédia-tartalmak lejátszását és felvételét, kitérünk a hangfelvétel lehetőségeire, valamint egy konkrét példán keresztül az mp3-as lejátszást is bemutatjuk. Végül pedig érdekességgként kitérünk az Android szövegfelolvasó API-jára.

10.1. Kamerakezelés Android platformon

Az Android platform egy rendkívül gazdag kamera API-val rendelkezik, amellyel nemcsak egyszerűen fényképeket készíthetünk, hanem különféle kamerával kapcsolatos algoritmusok, elemzések is könnyedén megvalósíthatók. Emellett az API lehetővé teszi az előlapi kamera kezelését is.

A kamera-API által biztosított főbb funkciók a következők:

- előnézeti kép (preview) elérése,
- kamera vezérlése (közelítés, vaku stb.),
- kép és videó rögzítése.

Kamera használatakor érdemes néhány dolgot átgondolni, majd ezek alapján megtervezni az alkalmazás felépítését. Elsőként gondoljuk át, hogy az alkalmazásunk kamera nélkül is futtatható-e, hiszen ha nem, akkor az eszköz meglétét mint követelményt jelezzük a *manifest* állományban. Bár manapság már szinte minden Android-alapú eszköz rendelkezik kamerával, de sosem árt biztosra menni. Következő megfontolandó kérdés az, hogy szükség van-e saját kamerafelület megvalósítására, vagy elegendő a beépített kameraalkalmazás, utóbbi esetben ugyanis sokkal könnyebb dolgunk van, és ez meggyorsíthatja a fejlesztést. További megfontolandó szempont, hogy a kamera által előállított médiatartalmaknak elérhetőnek kell-e lenniük más alkalmazások számára, kell-e ehhez interfészt vagy *ContentProvidert* készíteni, vagy ezek felhasználása csak az alkalmazáson belül valósul-e meg.

A kameraeszköz meglétének szükségességét a következőképpen jelezhetjük a *manifest* állományban:

```
<uses-feature android:name="android.hardware.camera"
    android:required="[true/false]"/>
```

A kamera használatához szükséges engedély az alábbi:

```
<uses-permission android:name="android.permission.
    CAMERA"/>
```

A rögzített médiatartalom fájlrendszerben való eltárolásához szükséges engedély így néz ki:

```
<uses-permission android:name=
    "android.permission.WRITE_EXTERNAL_STORAGE"/>
```

10.1.1. A beépített kameraalkalmazás használata

Az Android egyik legnagyobb előnye az, hogy a készüléken található alkalmazások könnyen együtt tudnak működni egymással. Ennek tipikus példája a kameraalkalmazás, amelyet gyakran csak egy gyors fotó készítése céljából kell meghívunk.

Ahhoz, hogy a beépített kameraalkalmazást meghívjuk, elsőként egy megfelelő Intent objektumot kell összeállítanunk, amelynek akciójában jelezni tudjuk a szándékunkat:

- *MediaStore.ACTION_IMAGE_CAPTURE*: képkészítési szándék,
- *MediaStore.ACTION_VIDEO_CAPTURE*: videóképzítési szándék.

Ezt követően az Intent objektum *putExtra()* függvényének segítségével még egyéb paramétereket is beállíthatunk. Leggyakrabban a kép mentési helyét szoktuk beállítani a *MediaStore.EXTRA_OUTPUT* kulcs megadásával.

Videó rögzítése esetén a következő Intent-paraméterek használhatók:

- *MediaStore.EXTRA_OUTPUT*: felvett videoállomány helye (URI),
- *MediaStore.EXTRA_VIDEO_QUALITY*: 0 és 1 közötti float, 0: a legrosszabb minőség és a legkisebb fájlméret,
- *MediaStore.EXTRA_DURATION_LIMIT*: videó hosszkorlátja másodpercben,
- *MediaStore.EXTRA_SIZE_LIMIT*: méretkorlát a felvett videóra.

Miután az Intentet összeállítottuk, a szándékot a *startActivityForResult()* függvény segítségével tudjuk elküldeni, teljesen hasonlóan ahhoz, mintha egy saját Activityt nyitnánk meg. A hívás eredményeképpen megnyílik a beépített kameraalkalmazás ismert Activityje. Miután a kamerával a fényképet/videót elkészítettük és jóváhagytuk, automatikusan visszatérünk a saját Activitynkhez, és az *onActivityResult()* függvény hívódik meg. A hívás eredményéről az *onActivityResult()* függvény paramétereiben érkező állapotkódokból tájékozódhatunk, a tartalmi eredmény pedig a szintén paraméterül kapott Intent objektum *getData()* függvényével kérdezhető le. Ha a *MediaStore.EXTRA_OUTPUT* kulccsal küldtük ki a kérést, a *getData()* függvény az elkészített fotó/videó helyét tartalmazza.

A következőkben nézzünk meg egy egyszerű példát, amelyben egy gombnyomás hatására a beépített kameraalkalmazással képet készíthetünk, az elkészített képet pedig egy *ImageView* segítségével megjelenítjük. Az alkalmazás egyszerű felhasználófelület-leírása XML-ben a következő – figyeljük meg az *ImageView* paraméterezését:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <Button
        android:id="@+id/btnTakePhoto"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnTakePhoto" />
    <ImageView
        android:id="@+id/ivPhoto"
        android:src="@drawable/noimage"
        android:layout_height="300dp"
        android:layout_width="fill_parent"
        android:layout_gravity="center"
        android:scaleType="fitCenter"
        android:contentDescription="@string/
imgPhotoDesc"/>
</LinearLayout>
```

A funkcionalitást megvalósító Activity pedig a következő:

```
public class CameraSimpleDemoActivity extends Activity {

    private final int CAMERA_IMAGE_REQUEST = 101;
    private final String IMAGEPATH =
        Environment.getExternalStorageDirectory().
        getAbsolutePath() + "/tmp_image.jpg";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button btnTakePhoto =
            (Button) findViewById(R.id.btnTakePhoto);
        btnTakePhoto.setOnClickListener(new
        OnClickListener() {
            @Override
```

```

        public void onClick(View v) {
            File imageFile = new File(IMAGEPATH);
            Uri imageFileUri = Uri.fromFile(imageFile);
            Intent cameraIntent = new Intent(
                android.provider.MediaStore.ACTION_IMAGE_
CAPTURE);
            cameraIntent.putExtra(
                android.provider.MediaStore.EXTRA_OUTPUT,
                imageFileUri);
            startActivityForResult(cameraIntent,
                CAMERA_IMAGE_REQUEST);
        }
    });
}

// Visszatérés a kamerától
protected void onActivityResult(int requestCode,
int resultCode, Intent data) {
    if (requestCode == CAMERA_IMAGE_REQUEST) {
        if (resultCode == RESULT_OK) {
            try {
                File imageFile = new File(IMAGEPATH);
                FileInputStream fis =
                    new FileInputStream(imageFile);
                Bitmap img = BitmapFactory.decodeStream(fis);
                ImageView ivPhoto =
                    (ImageView) findViewById(R.id.ivPhoto);
                ivPhoto.setImageBitmap(img);
            } catch (Exception e) {
            }
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this,
                getString(R.string.txtPhotoCancelled),
                Toast.LENGTH_LONG).show();
        }
    }
}
}
}
}

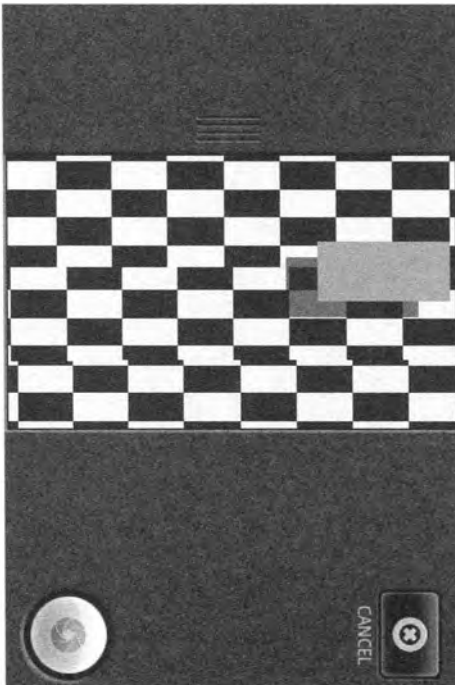
```


Az elkészült alkalmazást az alábbi ábrák szemléltetik.



10.2. ábra. Kép készítése a beépített kameraalkalmazással

Ha a kamerafunkcionalitást emulátoron teszteljük, egy fiktív videó jelenik meg: egy pepita háttéren mozgó szürke négyzet. Fotózáskor pedig a kamera által visszaadott kép egy fix kép.



10.3. ábra. Kamera tesztelése emulátoron

10.1.2. Arcfelismerés

Az Android platform alapértelmezetten támogatja az arcfelismerést, amelyhez egy egyszerű és jól használható API-t biztosít. Az arcfelismeréshez a *FaceDetector* osztályt kell használnunk, amely egy megadott kép alapján képes az arcok felismerésére, továbbá megadható, hogy egy képen maximálisan mennyi arcot keressen az algoritmus. A megadható maximális érték 64.

Arcfelismerés futtatásakor vegyük figyelembe, hogy sokáig is eltarthat, hiszen bonyolult algoritmusról van szó, tehát semmiképp se számítsunk valós idejű működésre. A megoldás egyébként használható például a kamera automatikus fókuszának beállítására, valamint különféle effektek alkalmazására direkt az arcokon.

Az arcfelismerő algoritmus az eredményt egy *Face* objektumokat tartalmazó tömb formájában adja vissza. Egy *Face* objektum az alábbi főbb paraméterekkel rendelkezik:

- *Confidence*: helyes detektálás valószínűsége,
- *EyesDistance*: szemek közti távolság,
- *MidPoint*: két szem közti középpont (*PointF* objektum),
- *Pose*: a felismert arc Euler-szöge a megadott tengelyhez képest (elfordulás a kiválasztott x, y vagy z tengelyekhez képest).

Példaképpen egészítsük ki az előző alkalmazásunkat úgy, hogy az elkészített képen futtassuk le az arcfelismerést (maximum 10 arccal), és az arcokat egy zöld karikával jelöljük, majd az így módosított képet jelenítsük meg.

A megvalósításhoz készítsünk egy saját, *View*-ből leszármazó osztályt, amely paraméterként megkapja az eredeti képet, azon lefuttatja az arcfelismerést, majd az *onDraw()* függvényben kirajzolja az eredeti képet, ezután pedig a zöld karikákat az arcfelismerő által kiszámított arcokra.

```
public class FaceView extends View {
    // max 64
    private static final int MAX_FACES = 10;
    private Bitmap sourceImg;
    private FaceDetector myFaceDetector;
    private FaceDetector.Face allFaces[] =
        new FaceDetector.Face[MAX_FACES];
    private FaceDetector.Face getFace = null;
    private PointF eyesMidPts[] = new PointF[MAX_FACES];
    private float eyesDistance[] = new float[MAX_FACES];
    private Paint tempPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private Paint eyePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private int imgWidth, imgHeight;
```

```

    private float xRat, yRat;

    public FaceView(Context context, Bitmap
aSourceImage) {
        super(context);

        eyePaint.setStyle(Paint.Style.STROKE);
        eyePaint.setColor(Color.GREEN);

        tempPaint.setStyle(Paint.Style.STROKE);
        tempPaint.setTextAlign(Paint.Align.CENTER);

        sourceImg = aSourceImage;

        imgWidth = sourceImg.getWidth();
        imgHeight = sourceImg.getHeight();

        myFaceDetector =
            new FaceDetector(imgWidth, imgHeight, MAX_
FACES);
        // ez a hívás sokáig tarthat
        myFaceDetector.findFaces(sourceImg, allFaces);

        for (int i = 0; i < allFaces.length; i++) {
            getFace = allFaces[i];
            try {
                PointF eyesMP = new PointF();
                getFace.getMidPoint(eyesMP);
                eyesDistance[i] = getFace.eyesDistance();
                eyesMidPts[i] = eyesMP;
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    protected void onDraw(Canvas canvas) {
        xRat = getWidth()*1.0f / imgWidth;
        yRat = getHeight()*1.0f / imgHeight;
        // teljes kép kirajzolása
        canvas.drawBitmap(sourceImg, null ,
            new Rect(0,0,getWidth(),getHeight()),tempPaint);
        // körök kirajzolása az arcokra
        for (int i = 0; i < eyesMidPts.length; i++) {
            if (eyesMidPts[i] != null) {
                eyePaint.setStrokeWidth(eyesDistance[i] /6);
            }
        }
    }
}

```

```

        canvas.drawCircle(eyesMidPts[i].x*xRat,
            eyesMidPts[i].y*yRat, eyesDistance[i] / 2 ,
            eyePaint);
    }
}
}
}

```

Figyeljük meg, hogy a tényleges arcfelismerés a *myFaceDetector* objektum *findFaces()* függvényének hívásakor történik, utána az *eyesDistance* tömbben eltároljuk a szemek közti távolságot, illetve az *eyesMidPts* tömbben a szemek középpontját. Az *onDraw()* függvényben ezen az *eyesMidPts* tömbön megyünk végig, és a középpontok alapján a szemtávolságok figyelembevételével kirajzoljuk a zöld karikákat.

Az előzőekben ismertetett példát úgy kell kiegészítenünk, hogy sikeres fényképezés után az előbb bemutatott *FaceView* nézet jelenjen meg, és az arcfelismerést a kamerától visszkapott képen végezzük el. Ehhez az *onActivityResult()* függvényt a következőképpen kell módosítani:

```

// Visszatérés a kamerától
protected void onActivityResult(int requestCode,
    int resultCode, Intent data) {
    if (requestCode == CAMERA_IMAGE_REQUEST) {
        if (resultCode == RESULT_OK) {
            try {
                File imageFile = new File(IMAGEPATH);
                FileInputStream fis = new
FileInputStream(imageFile);
                Bitmap img = BitmapFactory.decodeStream(fis);

                //--- Arcfelismerés aktiválása
                FaceView fv= new FaceView(this, img);
                setContentView(fv);
            } catch (Exception e) {
            }
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this,
                getString(R.string.txtPhotoCancelled),
                Toast.LENGTH_LONG).show();
        }
    }
}
}

```

A megvalósítás csupán annyiban módosult, hogy a kép elkészítése és *Bitmap*-pé való alakítása után a *FaceView*-t példányosítjuk, átadjuk neki a képet, majd a teljes nézetet megjelenítjük a *setContentView()* függvény segítségével.

10.1.3. Saját kamerakezelő készítése

Gyakran szükség lehet rá, hogy saját kamerafelületet, -funkciót valósítsunk meg, és ne a beépített kameraalkalmazást használjuk. Egy ilyen megoldásnál a kamera jobban integrálható az alkalmazásunk felhasználói felületébe, illetve speciális funkciókat is megvalósíthatunk, hiszen a kamerát sokszor nem csak fényképezésre használjuk. Tipikus példa, ha valamiért csak a kamera előnézeti képére van szükségünk, például egy mozgásérzékelő megvalósításakor.

Az Android által biztosított Camera API legfontosabb osztályai a következők:

- *Camera*: kameraeszköz vezérléséért felelős osztály,
- *SurfaceView*: kamera előnézeti képének megjelenítése,
- *MediaRecorder*: videó felvétele.

Saját kamerafelület készítésekor érdemes legelőször megvizsgálni, hogy található-e kamera az adott készüléken, majd rá kell kapcsolódnunk az eszközre (lehet, hogy több kamera is van egy telefonban). Továbbá szükség van egy saját előnézetkép-felület implementálására is (egy *Preview* osztályra, amely a *SurfaceView*-ből származik le, és megvalósítja a *SurfaceHolder* interfészt). Ha ez készen van, el kell helyeznünk az előnézeti képet a felületen, és szükség esetén meg kell valósítanunk a vezérlőgombokat (indítás, leállítás, kép/videó mentése stb). Végül a legfontosabb, hogy a kameraeszközt szabadítsuk fel, ha már nincs rá szükség.

Semmiképp ne feledkezzünk el a kameraeszköz felszabadításáról, elengedéséről (*release()* függvény), ellenkező esetben más alkalmazás, a beépített kameraalkalmazást is beleértve, nem képes elérni a kamerát.

Gyakran a kameraobjektumot Singleton tervezési minta segítségével szokták egy alkalmazásban megvalósítani, így könnyedén elérhető, és egyszerű a felszabadítása és az elengedése is.

A következőkben ezeket a lépéseket mutatjuk be részletesen.

Kameraeszköz létezésének vizsgálatára (ha a *manifest*-ben nem követeltük meg) például a következő függvényt használhatjuk:

```
private boolean checkCameraHardware(Context context) {
    if (context.getPackageManager().hasSystemFeature(
        PackageManager.FEATURE_CAMERA)) {
        // van kamera
    }
}
```

```

        return true;
    } else {
        // nincs kamera
        return false;
    }
}

```

Az Android 2.3-as verziójától felfelé létezik egy *Camera.getNumberOfCameras()* függvény is, amellyel ez az ellenőrzés még egyszerűbben megvalósítható.

A kamera megnyitásához a *Camera* osztály statikus *open()* függvényét kell használni. Android 2.3-as verziójától felfelé létezik egy *Camera.open(int)* függvény is, amelynek paraméterében megadhatjuk, hogy melyik kamerát nyissuk meg, ha több is van a készülékben. Ügyeljünk a kivételkezelésre, nem biztos, hogy mindig meg tudjuk nyitni a kamerát.

```

public static Camera getCameraInstance() {
    Camera c = null;
    try {
        c = Camera.open();
    } catch (Exception e){
        // Camera használatban van már,
        // vagy nem érhető el
    }
    return c; // hiba esetén null-lal térünk vissza
}

```

A *Camera* példánytól számos egyéb információ lekérdezhető a *getParameters()* függvénnyel, illetve a 2.3-as verziótól fölfelé a *getCameraInfo()* függvény is használható, amellyel az első és a hátsó kamera, illetve a képorientáció is megkülönböztethető. Legfőbb *Camera* paraméterek a következők:

- vakuállapot,
- közelítés (zoom),
- JPEG-minőség,
- képméret,
- effektek.

A kamera által támogatott felbontások például a következőképpen kérdezhetők le:

```
private void checkCameraParameters() {
    Camera c = null;
    try {
        c = Camera.open();
        Parameters p = c.getParameters();
        for (Size s : p.getSupportedPictureSizes()) {
            Log.d("CAM", s.width+"*" +s.height);
            tvStatus.append(s.width+"*" +s.height+"\n");
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (c != null)
            c.release();
    }
}
```

Az alábbiakban nézzünk meg egy olyan példát, amelyben a kamera előnézeti képét megjelenítjük, illetve egy olyan gombot is elhelyezünk, amelynek segítségével fényképet tudunk készíteni saját alkalmazásunkból.

A példaalkalmazás felhasználói felülete rendkívül egyszerű. Egy *RelativeLayout*-ra elhelyezünk egy *FormLayout*-ot, amelybe majd az előnézeti képet helyezzük el, továbbá lesz még egy gombunk, valamint egy kezdetben nem látható *ImageView*, amely a fotózott képet jeleníti meg.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <FrameLayout
        android:id="@+id/cameraPreview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    />
    <ImageView
        android:id="@+id/ivPhoto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"
```

```

        android:contentDescription="@string/ivPhoto"
    />
    <Button
        android:id="@+id/buttonPhoto"
        android:text="@string/btnCapture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
    />
</RelativeLayout>

```

Kamera kezelésekor mindenképp kell egy felület, amelyhez a kamerát hozzárendeljük, különben nem működik. Ám ezt a felületet már nem kötelező ténylegesen elhelyeznünk a felhasználói felületen, de a kamerához mindenképp hozzá kell rendelni.

A kamerafelületet megvalósító osztály a következő:

```

public class MyPreview extends SurfaceView implements
    SurfaceHolder.Callback {
    private SurfaceHolder mHolder;
    private Camera mCamera;

    public MyPreview(Context context, Camera camera) {
        super(context);
        mCamera = camera;
        mHolder = getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_
            BUFFERS);
    }

    public void surfaceCreated(SurfaceHolder holder) {
        try {
            mCamera.setPreviewDisplay(holder);
            mCamera.startPreview();
        } catch (IOException e) {
            Log.d("CAM", "Failed to start preview: " +
                e.getMessage());
        }
    }

    public void surfaceDestroyed(SurfaceHolder holder) {
        // TODO: további funkciók az előnézeti kép bezáró-
        // dásakor
    }

    public void surfaceChanged(SurfaceHolder holder,

```



```

        int format, int w, int h) {
            // TODO: további funkciók, ha szükséges
        }
    }
}

```

Ez az osztály megvalósítja a *SurfaceHolder.Callback* interfészt, amelynek eredményeként az osztály implementálja a *surfaceCreated()*, a *surfaceDestroyed()* és a *surfaceChanged()* függvényeket. A *surfaceCreated()* a mi esetünkben akkor fut le, amikor az előnézeti képet ráhelyeztük a tényleges felhasználói felületre, ezért ilyenkor indítjuk el ténylegesen az előnézeti képet ebben a *surfaceCreated()* függvényben.

Végül az *Activity*, amelyben felhasználjuk a *MyPreview* osztályt, a következő:

```

public class MyCameraDemoActivity extends Activity {
    private Camera mCamera;
    private MyPreview mPreview;
    private ImageView ivPhoto;
    private FrameLayout preview;

    private PictureCallback mPicture = new
    PictureCallback() {
        @Override
        public void onPictureTaken(byte[] data, Camera camera) {
            Log.d("CAM", "Size: " + data.length);
            ByteArrayInputStream imageStream =
                new ByteArrayInputStream(data);
            Bitmap theImage =
                BitmapFactory.decodeStream(imageStream);
            ivPhoto.setImageBitmap(theImage);
            mCamera.stopPreview();
            preview.removeView(mPreview);
            ivPhoto.setVisibility(View.VISIBLE);
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mCamera = Camera.open();

        // Kamera effekt
        Camera.Parameters parameters = mCamera.
        getParameters();
    }
}

```

```

        parameters.setColorEffect(Camera.Parameters.
EFFECT_SEPIA);
        mCamera.setParameters(parameters);

        mPreview = new MyPreview(this, mCamera);
        preview = (FrameLayout) findViewById(R.
id.cameraPreview);
        preview.addView(mPreview);

        final Button captureButton =
            (Button) findViewById(R.id.buttonPhoto);
        captureButton.setOnClickListener(
            new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    // get an image from the camera
                    mCamera.takePicture(null, null, mPicture);
                    captureButton.setEnabled(false);
                }
            });
        ivPhoto = (ImageView) findViewById(R.id.ivPhoto);
    }

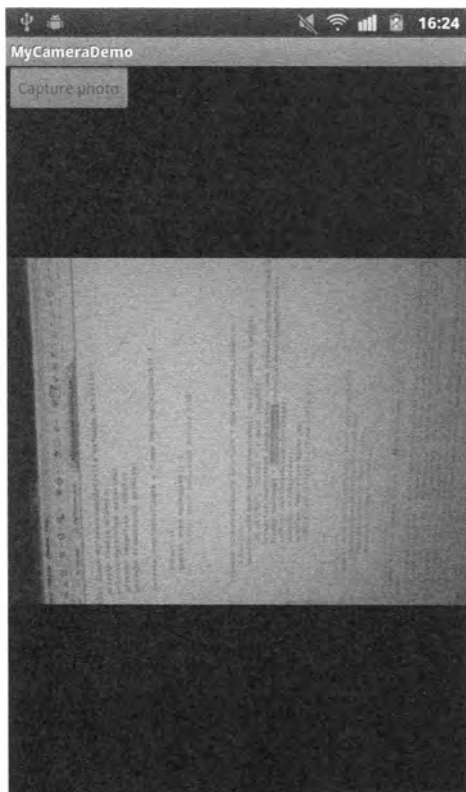
    @Override
    protected void onStop () {
        if (mCamera != null) {
            // NE FELEJTSÜK EL!
            mCamera.release();
        }
        super.onStop();
    }
}

```

Az Activity kódja több ismeretlen részt is tartalmaz. Elsőként figyeljük meg, hogy a kamera megnyitása után beállítunk egy *Sepia* effektet, amelynek eredményeképpen barnás lesz a kamera képe. Ezt követően példányosítjuk a *MyPreview* osztályunkat, és elhelyezzük az erőforrásban található *FrameLayout*-ra. Ezt követi a fényképezésgomb eseménykezelője, amelyben a *Camera* osztály *takePicture()* függvényét használjuk. A *takePicture()* függvénynek három paramétere van, amelyek *null* értékűek is lehetnek. Az első egy *ShutterCallback* objektum, amely a kép készítése előtt aktiválódik (meghívódik az *onShutter()* függvénye), ilyenkor például valamilyen hangot játszhatunk le. A második egy *PictureCallback* objektum, amely akkor aktiválódik (meghívódik az *onPictureTaken()* függvénye), amikor a nyers kép (*raw*) rendelkezésre áll, ha esetleg azon szeretnénk valamilyen manipulációt végezni. Végül a harmadik szintén egy *PictureCallback*-t implementáló objektum, amely aktiválódáskor a már tömörített JPEG-képet kapja meg az *onPictureTaken()* függvényében.

A *takePicture()* függvényben az előző példában egy *mPicture* objektumot adunk át, amelynek megvalósítása az Activity elején található. Ennek az objektumnak az *onPictureTaken()* függvényében pedig kiolvassuk a *byte[]*-ben rendelkezésre álló képet, átalakítjuk a *BitmapFactory* osztály segítségével *Bitmap*-pé, és megjelenítjük a felületen lévő *ImageView* komponensben.

Végül az Activity *onStop()* függvényében elengedjük a kameraeszközt a *Camera* objektum *release()* függvényének a meghívásával.



10.4. ábra. Saját kameraalkalmazás

Ha nem a kamera által fényképezett nagyfelbontású képre van szükségünk, hanem csak az előnézeti képen látható kisebb képekre, ezeket is könnyedén elérhetjük. Csupán annyit kell tennünk, hogy a *Camera* objektum *setPreviewCallback()* függvényén keresztül át kell adnunk egy *PreviewCallback* objektumot a kamerának, és minden egyes előnézeti kép esetén a rendszer meghívja a *PreviewCallback* objektum *onPreviewFrame()* függvényét, amelynek paraméterében átadja az előnézeti képet.

A kamera előnézeti képének *frame* sebessége (*frame rate*) beállítható a korábban említett kamerától elkérhető *Parameters* objektum *setPreviewFrameRate(int)* függvényével.

10.1.4. Kiterjesztett valóságalapok

Miután már saját kamerafelülettel rendelkezünk az alkalmazásunkban, könnyen kiegészíthetjük az alkalmazásunkat úgy, hogy a kamera előnézeti képére különféle adatokat rajzoljunk, és így megteremtjük a kiterjesztettvalóság-jellegű funkcionalitás alapjait.

A következőkben arra mutatunk egy példát, hogy hogyan tudunk a kamera képe fölé egy átlátszó dobozban valamilyen információt kiírni. A megvalósításhoz az előző példát egészítjük ki, és egy saját *View* komponenst helyezünk az alap *RelativeLayout*-ra a kamera képe fölé. A komponens egy átlátszó téglalapot rajzol ki, amelyben szöveget jelenítünk meg.

A saját nézetet megvalósító osztály a következő:

```
public class AugmentedView extends View {

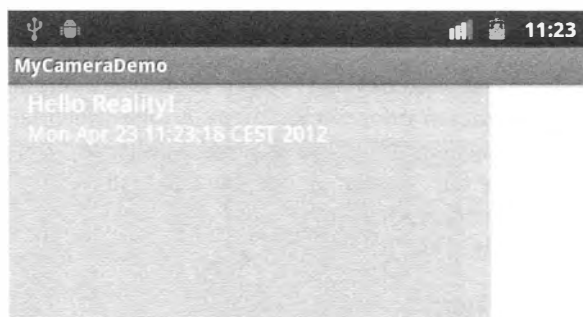
    public AugmentedView(Context context) {
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        mPaint.setColor(0x44FF0000);
        canvas.drawRect(0, 0, 500, 250, mPaint);
        mPaint.setColor(0xFFFFFFFF);
        mPaint.setTextSize(26);
        canvas.drawText("Hello Reality!", 20, 30, mPaint);
        mPaint.setTextSize(22);
        canvas.drawText(new Date(System.
currentTimeMillis()).
        toString(), 20, 60, mPaint);
    }
}
```

Tegyük fel, hogy a felhasználói felületet leíró erőforrásban (amely megegyezik az előző példában bemutatottal) a gyökér *RelativeLayout* elem azonosítója *baseLayout*. Ennek megfelelően a saját *AugmentedView*-t a következőképpen helyezhetjük a kamera fölé:

```
RelativeLayout baseLayout =
    (RelativeLayout) findViewById(R.id.baseLayout);
AugmentedView av = new AugmentedView(this);
baseLayout.addView(av, new RelativeLayout.
LayoutParams(
    LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_
CONTENT));
```

A megoldást az alábbi ábra szemlélteti. Sajnos az ábrán a kamera képe nem látszik, mivel a DDMS-en keresztüli képernyőkép készítéskor a kamerakép mentése nem történik meg.



10.5. ábra. Kiterjesztettvalóság-példa

10.1.5. Videófelvétel és -lejátszás

Videófelvétel a képkészítéshez hasonlóan történik Android platformon, kivéve azt, hogy a *Camera.open()* és *release()* függvényei mellett a *lock()* és az *unlock()* függvényeket is meg kell hívunk, mielőtt elkezdjük, illetve miután befejeztük a felvételt.

Android 4.0-tól már nem kell kezelniük a *lock()* és az *unlock()* függvényeket.

Továbbá a felvételhez több beállítást is megadhatunk, ilyen például a felvétel maximális hossza vagy maximális mérete a tárhelyen. Videókezelő alkalmazás készítésekor ügyeljünk arra, hogy ne hagyjunk felesleges videóállományokat a tárhelyen, hiszen ezek rendkívül sok helyet foglalnak, továbbá itt se felejtjük el a kamerát felszabadítani.

Videók lejátszásához használhatjuk a beépített *VideoView* komponenst, amely képes a videótartalmak megjelenítésére.

```
<VideoView
    android:id="@+id/videoView"
    android:layout_width="200dp"
    android:layout_height="200dp"
/>
```

Az előző nézetet felhasználva például egy *http*-oldalon található videó a következőképpen játszható le:

```

VideoView videoView =
    (VideoView) this.findViewById(
        R.id.videoView);
MediaController mc =
    new MediaController(this);
videoView.setMediaController(mc);
videoView.setVideoURI(
    Uri.parse("http://www.test.hu/test.mp4"));
videoView.requestFocus();
videoView.start();

```

Ha a fájlrendszerből szeretnénk egy videót lejátszani, ezt a következőképpen tehetjük meg:

```

String sdCard = Environment.
    getExternalStorageDirectory().
    getAbsolutePath()
videoView.setVideoPath(sdCard + "/movie.mp4");

```

10.2. Multimédia-kezelés

A továbbiakban röviden áttekintjük, hogyan tudunk egyszerű, illetve összetettebb hangokat lejátszani, hangot felvenni, illetve pontosan hogyan is néz ki egy *mp3*-as lejátszás folyamata.

Multimédia-tartalom lejátszásakor leggyakrabban az alábbi osztályokra van szükség:

- *RingtoneManager*: figyelmeztető hangok elérése, lejátszása (*Alarm*, *Notification*, *RingTone*),
- *MediaPlayer*: általános médialejátszó,
- *ToneGenerator*: bonyolultabb hangszekvenciák előállítása.

10.2.1. Egyszerű hangok lejátszása és felvétele

Hangok lejátszásakor elsőként nézzük meg, hogyan tudunk egyszerű rendszerhangokat lejátszani. A *RingtoneManager* osztály segítségével elérhetők a beépített riasztások:

- *RingtoneManager.TYPE_RINGTONE*: alapértelmezett csengőhang,
- *RingtoneManager.TYPE_ALARM*: alapértelmezett riasztáshang,

- *RingtoneManager.TYPE_NOTIFICATION*: alapértelmezett figyelmeztető hang.

A beépített riasztáshangokat az alábbi egyszerű módon játszhatjuk le:

```
private void playNotificationTone() {
    Uri uriNotif = RingtoneManager.getDefaultUri(
        RingtoneManager.TYPE_NOTIFICATION);
    Ringtone r = RingtoneManager.getRingtone(
        getApplicationContext(), uriNotif);
    r.play();
}
```

Tulajdonképpen egy *Ringtone* objektum jött létre, amelynek *play()* függvényével indítjuk el a lejátszást. Androidon egy hang lejátszásakor beállíthatjuk, hogy melyik hangcsatornán történjen a lejátszás. Az egyes hangcsatornák konstansként az *AudioManager* osztályon keresztül érhetők el:

- *AudioManager.STREAM_ALARM*,
- *AudioManager.STREAM_DTMF*,
- *AudioManager.STREAM_MUSIC*,
- *AudioManager.STREAM_NOTIFICATION*,
- *AudioManager.STREAM_RING*,
- *AudioManager.STREAM_SYSTEM*,
- *AudioManager.STREAM_VOICE_CALL*.

Az előző példában például az egyszerű hangot a zenei csatornán a következőképpen tudunk lejátszani:

```
r.setStreamType(AudioManager.STREAM_MUSIC);
```

Ha a lejátszást nem a zenei csatornán valósítjuk meg, hanem például az *ALARM*-on, akkor bizonyos telefonokon előfordulhat, hogy ha levesszük a hangerőt, ez a hang akkor is hangosan szólal meg. Továbbá beépített hangok esetében fontos, hogy a *play()* függvénnyel való lejátszásindítás lehet, hogy folyamatos hanglejátszást eredményez, amelyet csak a *stop()* függvénnyel lehet leállítani.

Nem minden csatorna használatát engedélyezi mindig a rendszer, így például sok telefonon a hanghívási csatornába nincs lehetőség egyedi hangokat bejátszani.

Bonyolultabb esetben általában *MediaPlayer*-t szokás használni, amelyen keresztül többféle beállítás és lehetőség áll a rendelkezésre. Elsőként nézzük meg, hogy az előző beépített hangokat hogyan tudjuk *MediaPlayer*-rel lejátszani.

```
private void playToneMediaPlayer() {
    try {
        Uri notificationTone = RingtoneManager.
        getDefaultUri(
            RingtoneManager.TYPE_NOTIFICATION);
        MediaPlayer mMediaPlayer = new MediaPlayer();
        mMediaPlayer.setDataSource(MediaDemoActivity.this,
            notificationTone);
        mMediaPlayer.setAudioStreamType(
            AudioManager.STREAM_MUSIC);
        mMediaPlayer.setLooping(false);
        mMediaPlayer.prepare();
        mMediaPlayer.start();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

A lejátszás leállítása ekkor is az *mMediaPlayer* objektum *stop()* függvényével történik, a *Ringtone* esetéhez hasonlóan. Az előző példában elsőként összeállítjuk a médiához tartozó Uri-t, majd létrehozuk a *MediaPlayer* objektumot, és beállítjuk az adatforrását. Ezt követi a csatorna kiválasztása, valamint az ismétlés letiltása (a beépített csengőhang és a riasztás ettől függetlenül is ismétlődve játszódhat le az egyes eszközökön). A *prepare()* függvény szerepe az, hogy minden erőforrást lefoglal, és felkészül az azonnali lejátszásra, így a *start()* függvény meghívásakor már biztosak lehetünk benne, hogy a lejátszás azonnal elindul.

Ha nem egyszerű figyelmeztető hangot, hanem valamilyen összetettebb médiát kívánunk lejátszani, meg kell adnunk a média elérési útját. Ehhez használhatunk például URI-alapú megadást is.

```
Uri myUri = Uri.parse(...);
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(
    AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(
    getApplicationContext(), myUri);
mediaPlayer.prepare();
mediaPlayer.start();
```


Egyszerűen egy URI-ként meg kell adnunk az elérési utat, és ezt a `setDataSource()` függvénnyel be kell állítani.

Ha egy erőforrásként megadott médiát szeretnénk lejátszani, akkor ezt a `res/raw` könyvtár alá kell elhelyeznünk, és a következőképpen tudjuk elérni:

```
MediaPlayer mediaPlayer = MediaPlayer.create(context,
    R.raw.mysound);
mediaPlayer.start();
```

Erőforrásból való lejátszáskor nincs szükség `prepare()` hívásra, a `create()` ezt elvégzi helyettünk.

Az alkalmazáshoz tartozó erőforrások egyébként URI-n keresztül is elérhetők a következő formátumban: „*android.resource://[package]/[res type]/[res name]*”. Például:

```
Uri path =
    Uri.parse(
        "android.resource://hu.bute.daai.examples/raw/
        mysound");
```

Egy HTTP-címen található média esetén a lejátszás például a következőképpen néz ki:

```
String url = "http://test.hu/test.wav";
MediaPlayer mediaPlayer =
    new MediaPlayer();
mediaPlayer.setAudioStreamType(
    AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
// bufferelés miatt sokáig tarthat
mediaPlayer.prepare();
mediaPlayer.start();
```

10.2.2. Az *AudioManager* használata

A hangkezeléssel kapcsolatos különféle funkciók az *AUDIO_SERVICE* rendszerszolgáltatáson keresztül érhetők el, amelyet egy *AudioManager* objektum formájában tudunk elérni fejlesztés közben.

```
AudioManager audioManager =
    (AudioManager) getSystemService(Context.AUDIO_SERVICE);
```

Az *AudioManager* legfontosabb függvényei a következők:

- *setMicrophoneMute()*: mikrofon elnémítása,
- *setStreamVolume()*: kiválasztott hangsáv hangerejének a beállítása,
- *setVibrateSetting()*: rezgés beállítása,
- *isWiredHeadsetOn()*: vezetékes fülhallgató ellenőrzése,
- *playSoundEffect()*: hangeffekt gyors lejátszása.

Példaképpen a zenei csatorna hangerejét az alábbiak szerint tudjuk beállítani:

```
AudioManager audioManager =
    (AudioManager) getSystemService(Context.AUDIO_SERVICE);
audioManager.setStreamVolume(AudioManager.STREAM_RING,
    100,
    AudioManager.FLAG_SHOW_UI);
```

A *setStreamVolume()* függvényben különféle *Flageket* is megadhatunk, amelyek meghatározzák, hogy a felhasználó hogyan értesüljön a hangerő változtatásáról. Példánkban a megszokott dialógusablak ugrana fel, és jelezné az új hangerőértéket.

10.2.3. A készülék erőforrásainak ébrentartása hosszú médialejátszás során

Ha hosszabb médialejátszását indítunk el, előfordulhat, hogy a készülék alvó állapotba kerül, és például kikapcsolja vagy alacsonyabb teljesítményre veszi a CPU-t és/vagy a wifit. Ez lejátszás és streamelés esetén problémát jelenthet. Android platformon ennek a kiküszöbölésére a *WakeLock* mechanizmust kell alkalmaznunk. A *WakeLock* segítségével jelezni tudjuk a rendszer felé, hogy az alkalmazásunk bizonyos erőforrásokra folyamatos igényt tart, és így az erőforrás nem kapcsol ki automatikusan. Ám ha végeztünk a médialejátszással, a *WakeLock*ot a végén mindenképpen engedjük el, semmiképpen se hagyjuk bekapcsolva, hiszen ezzel a készülék energiahatékonyságát jelentősen csökkentjük.

MediaPlayer esetén a használat a következő:

```
MediaPlayer mMediaPlayer = new MediaPlayer();
mMediaPlayer.setWakeMode(
    getApplicationContext(),
    PowerManager.PARTIAL_WAKE_LOCK);
```

Ha például a wifieszközre szeretnénk egy *WakeLock*-ot foglalni, ezt a következőképpen tehetjük meg:

```
WifiLock wifiLock =
    ((WifiManager) getSystemService(Context.WIFI_SERVICE))
    .createWifiLock(WifiManager.WIFI_MODE_FULL, "mylock");
wifiLock.acquire();
```

A folyamat végén a felengedés a *release()* függvénnyel valósítható meg:

```
wifiLock.release();
```

10.2.4. Hangfelvétel megvalósítása

Az Android Multimedia Framework támogatja hangok felvételét is a *MediaRecorder* API-n keresztül. Fontos tudni azonban, hogy a hangfelvétel nem minden eszközön van támogatva, ezt ellenőrizzük a funkció indítása előtt. Továbbá nem mindegyik emulátorverzió teszi lehetővé, hogy a számítógép mikrofonját erre a célra használjuk.

A hangfelvétel folyamata az alábbi fő lépésekből áll:

- *MediaRecorder* példány létrehozása,
- audioforrás beállítása a *setAudioSource()* függvénnyel, például: *MediaRecorder.AudioSource.MIC*,
- kimeneti formátum beállítása a *setOutputFormat()* függvénnyel, például: *MediaRecorder.OutputFormat.THREE_GPP* (gyakran használt formátum),
- kimeneti cél beállítása, például fájl esetén a *setOutputFile()* függvénnyel,
- audiokódolás beállítása a *setAudioEncoder()* függvénnyel, például: *MediaRecorder.AudioEncoder.AMR_NB* (gyakran használt formátum),

- felvétel hosszának vagy maximális méretének beállítása opcionálisan a *setMaxDuration()* vagy a *setMaxSize()* függvényekkel,
- felvétel előkészítése, indítása és leállítása a *MediaRecorder.prepare()*, *start()* és *stop()* függvényekkel,
- *MediaRecorder* objektum elengedése a *release()* függvénnyel.

A következőkben nézzünk meg egy egyszerű példát, amelyben egy hangfelvételt indítunk el:

```
MediaRecorder mRecorder = new MediaRecorder();
mRecorder.setAudioSource(MediaRecorder.AudioSource.
    MIC);
mRecorder.setOutputFormat(
    MediaRecorder.OutputFormat.THREE_GPP);
String fileName = Environment.
    getExternalStorageDirectory().
    getAbsolutePath()+"/audiorecordtest.3gp";
File outputFile = new File(fileName);
if (outputFile.exists()) {
    outputFile.delete();
}
outputFile.createNewFile();
mRecorder.setOutputFile(fileName);
mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.
    AMR_NB);
mRecorder.prepare();
mRecorder.start();
```

Ha a felvétellel végeztünk, a *MediaRecorder* objektum *release()* függvényével ne felejtjük elengedni az erőforrást.

Az előző példában be kell állítani a szükséges engedélyeket a *manifest* állományban:

```
<uses-permission android:name=
    "android.permission.RECORD_AUDIO"/>
<uses-permission android:name=
    "android.permission.WRITE_EXTERNAL_STORAGE"/>
```

10.2.5. MP3-lejátszás

Végül nézzünk meg egy összetettebb példát, amely egy egyszerű *mp3*-as lejátszást valósít meg erőforrásból. Az alkalmazás felhasználói felületén három gomb található, amellyel indítható, szüneteltethető és leállítható a lejátszás. A megoldást a következő kódrészlet ismerteti:

```
public class Mp3PlayerActivity extends Activity
implements
    OnPreparedListener, OnCompletionListener {
    private MediaPlayer mp;
    private Button btnStart;
    private Button btnPause;
    private Button btnStop;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        btnStart = (Button)findViewById(R.id.btnStart);
        btnStart.setOnClickListener(new OnClickListener()
        {
            @Override
            public void onClick(View arg0) {
                mp.start();
            }
        });
        btnPause = (Button)findViewById(R.id.btnPause);
        btnPause.setOnClickListener(new OnClickListener()
        {
            @Override
            public void onClick(View arg0) {
                mp.pause();
            }
        });
        btnStop = (Button)findViewById(R.id.btnStop);
        btnStop.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                mp.seekTo(0);
                mp.pause();
            }
        });

        preparePlayer();
    }

    @Override
```

```

protected void onStop() {
    super.onStop();
    if (mp != null) {
        try {
            mp.stop();
            mp.release();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private void preparePlayer() {
    try {
        mp = MediaPlayer.create(this, R.raw.test);
        mp.setOnPreparedListener(this);
        mp.setOnCompletionListener(this);
    } catch (Exception e) {
    }
}

public void onPrepared(MediaPlayer mp) {
    btnStart.setEnabled(true);
    btnPause.setEnabled(true);
    btnStop.setEnabled(true);
}

@Override
public void onCompletion(MediaPlayer mp) {
    Toast.makeText(this, "Media finished!",
        Toast.LENGTH_LONG).show();
}
}

```

Példánkban az *onCreate()* függvény végén meghívjuk a *preparePlayer()* függvényt, amelyben létrehozuk a *MediaPlayer* objektumot az *R.raw.test* erőforrásra, ez estünkben egy mp3-as állomány. Ebben a függvényben beállítunk egy *OnPreparedListener*, valamint egy *OnCompletionListener* objektumot a lejátszóhoz. Az előbbi *onPrepared()* függvénye akkor hívódik meg, ha a rendszer felkészült a média lejátszására, míg az utóbbi *onCompletion()* függvénye akkor hívódik meg, ha befejeződött a média lejátszása.

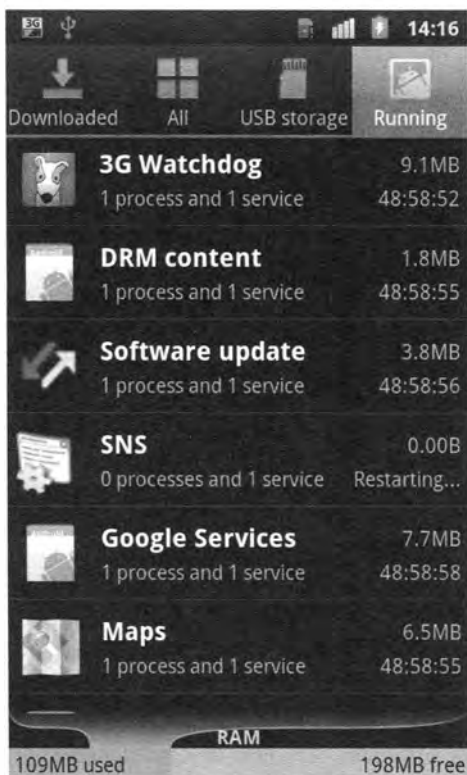
Az *onPrepared()* függvényt áttekintve látható, hogy ilyenkor engedélyezzük a médiavezérlő gombokat, illetve az *onCompletion()* függvény meghívódásakor csupán csak egy *Toast*-al jelezzük a lejátszás végét.

Android-szolgáltatások

A szolgáltatások, vagyis a *Service*-ek, az Android-alkalmazások alapkomponeisei közé tartoznak. Alapvető vagy valamilyen hosszabb ideig tartó, a háttérben futó feladatot látnak el, vagy más alkalmazásoknak nyújtanak szolgáltatásokat. *Service* lehet egy zenelejátszást végző komponens, egy a háttérben GPS-pozíció-adatokat gyűjtő osztály vagy akár egy teljes FTP kliens/szerver is. A *Service*-ekhez nem tartozik felhasználói felület, így ha a *Service* által szolgáltatott adatokat valahogy meg szeretnénk jeleníteni, akkor ezt egy *Activity*-ben tehetjük meg.

A *Service*-eknek sok lehetséges felhasználási területe van, ezek közül néhány a következő:

- A *Service*-ek lehetővé teszik, hogy az alkalmazás *Activity*-jeitől függetlenedjünk. A *Service*-ek képesek tovább futni akkor is, ha az alkalmazás összes *Activity*-jét bezárták. Még ha a felhasználó úgy is érzi, hogy „kilépett” a programból, a *Service* tovább fut a háttérben, és például adatokat tölthet le, vagy folytathatja a zenelejátszást.
- A *Service* lehetővé teszi, hogy egy program bizonyos szolgáltatásait más alkalmazások számára is elérhetővé tegyünk. Ilyenkor a *Service*-t elképzelhetjük szerverként, amely más alkalmazások kéréseit fogadja egy általunk definiált interfészen keresztül.
- A *Service*-ek végrehajthatnak ütemezett, periodikus feladatokat. Az általunk megadott kód a kívánt időközönként lefut. Például óránként felkapcsolódhatunk egy webszerverre, és feltöltetjük a telefontal készült új fotókat, vagy percenként elmenthetjük egy állományba a gyorsulásérzékelő szenzor által mért adatokat.



11.1. ábra. A készüléken futó szolgáltatások listája

11.1. Service-alapok

11.1.1. Service ösosztály

Minden Service közös őse a *Service* osztály, ebből vagy valamelyik leszármazottjából (pl. *IntentService*) kell leszármaztatnunk egy saját osztályt új Service-típus készítésekor. A *Service* osztály bizonyos callback metódusait (pl. *onCreate()*, *onStartCommand()* stb.) felüldefiniálva valósíthatjuk meg a kívánt működést. Ezeket a metódusokat az Activitykhez hasonlóan a rendszer hívja meg a megfelelő időben.

11.1.2. A Service-ek deklarálása a *manifest* állományban

A Service-eket kötelező bejegyezni az alkalmazás *manifest* állományába. Ha ezt elmulasztjuk, a rendszernek nem lesz tudomása a Service létezéséről, így nem is tudja elindítani. A `<service>` Tagben egyedül a *name* attribútum megadása kötelező.


```
<service android:name=".MyService" />
```

A megadható attribútumok listája a következő:

- **name:** A Service-hez tartozó osztály teljes neve, mindenképp meg kell adni.
- **exported:** Eldönti, hogy más alkalmazások komponensei használhatják-e a Service-t, vagy sem. Ha *false*-ra állítjuk, akkor csak annak az alkalmazásnak a komponensei férhetnek hozzá, amelyben a Service-t is definiáljuk. Alapértelmezett értéke attól függ, hogy a Service-hez megadtunk-e filtereket. Ha van legalább egy filter, akkor *true*, filterek hiányában *false*.
- **permission:** Megmutatja, hogy milyen engedéllyel kell rendelkezni a Service-t hívó félnek. Ha nincsen megadva, akkor az alkalmazás (<application> Tag) engedélyét örökli.
- **icon:** A Service ikonja. Ha nem adjuk meg, akkor az alkalmazás ikonja tartozik hozzá.
- **label:** A Service felhasználói felületen megjeleníthető neve.
- **process:** A processz neve, amelyben a Service fut. Ha nincs megadva, akkor az alkalmazás processzában fut. Ha a megadott processznév kettősponttal (:) kezdődik, akkor Service indításakor egy új processz indul a megadott névvel.
- **enabled:** Megmutatja, hogy a Service példányosítása és futtatása engedélyezve van-e, vagy sem. Ha *false*-ra állítjuk, akkor a Service-t nem lehet használni, de később a kódból még van lehetőségünk engedélyezni.

11.1.3. A Service-ek két fő típusa: *Started* és *Bound*

A Service-ek által nyújtott szolgáltatásokat alapvetően kétféleképpen érhetjük el. Az úgynevezett *Started Service*-eknél a Service egy olyan parancsot hajt végre, amelyet egy Intent formájában adunk át neki. A Service megkapja a parancsot, majd ennek megfelelően elvégez valamilyen műveletet. A hívó komponens értesítéséről külön kell gondoskodnunk. Ennél kifinomultabb hozzáférést tesznek lehetővé az úgynevezett *Bound Service*-ek, amelyeknél a Service egy saját, a programozó által definiált interfészt bocsát a hívó fél rendelkezésére. Ellentétben a *Started Service*-ek Intent-alapú parancsaival, a *Bound Service* tetszőleges számú és típusú metódust definiálhat a Service-szel való kommunikációhoz.

Egy Service működhet *Started* és *Bound Service*-ként is, a programozó dönti el, hogy lehetővé teszi-e mindkét típusú működést.

11.1.4. Service-ek a fő szálban

Ha készítettünk egy saját Service leszármazott osztályt, akkor innen kezdve megvan a lehetőség a Service elindítására és a vele való kommunikációra. Amikor a Service „fut”, akkor az osztály egy példányának bizonyos metódusai hajtódnak végre. Alapesetben minden, a Service metódusaihoz tartozó kód az alkalmazás fő szálában fut le. Ez nagyon fontos szempont, hiszen a fő szál az Android esetében egyben az UI-szál is, vagyis bármilyen hosszabb ideig tartó művelet a felhasználói felület blokkolását idézi elő. Ha 5 másodpercnél tovább foglaljuk a szálát, akkor az Activitykhez hasonlóan a rendszer feldobja a hírhedt „Application Not Responding (ANR)” ablakot. Az ilyen esetek elkerülésére a legtöbb Service tartalmaz egy vagy több háttérszálát, és azon hajtja végre a műveleteit. Csakis a legegyszerűbb, garantáltan gyorsan lefutó kód-részleteket futtassuk az alkalmazás fő szálában. A háttérszállal kapcsolatos adminisztrációt is megspórolhatjuk az *IntentService* osztály használatával, amely alapból egy külön szálban futtatja le az általunk megadott kódot (lásd később).

Arra is van lehetőségünk, hogy a Service egy általunk megadott processz szálában fusson, ennek megadása a *manifest* állományban történik.

11.1.5. Service-ek leállítása a rendszerrel

A futó Service-eket a rendszer bármikor leállíthatja, ha kevés a memória. Erre jellemzően akkor kerül sor, amikor az előtérben futó Activitynek több memóriára van szüksége. Ilyenkor a rendszer igyekszik a Service-ek közül azokat leállítani, amelyek régóta futnak, és valószínűleg „kevésbé fontosak” a felhasználó számára. A felhasználóval közvetlen kapcsolatban lévő Service-eket csak végső esetben állítja le, ilyenek a következők:

- egy Activityhez kötött (Bound) Service,
- az előtérben futó (Foreground) Service.

Ha a rendszer memóriahiány miatt leállít egy Service-t, akkor nincs rá garancia, hogy az *onDestroy()* meghívódik.

Ha elég memória szabadul fel, akkor a rendszer képes automatikusan újraindítani a leállított *Started Service*-eket. Ennek módját a *Service* osztályból örökölt *onStartCommand()* metódus felüldefiniálásával és a visszatérési érték megadásával szabályozhatjuk.

A *bindService()*-szel indított Service-eket a rendszer nem indítja újra automatikusan. Ha szeretnénk, hogy egy *Bound Service* újrainduljon, akkor küldjünk neki egy *startService()* parancsot is (egy Service üzemelhet egyszerre *Started* és *Bound Service*-ként is), és implementáljuk az *onStartCommand()*-ot.

11.2. Started Service-ek írása

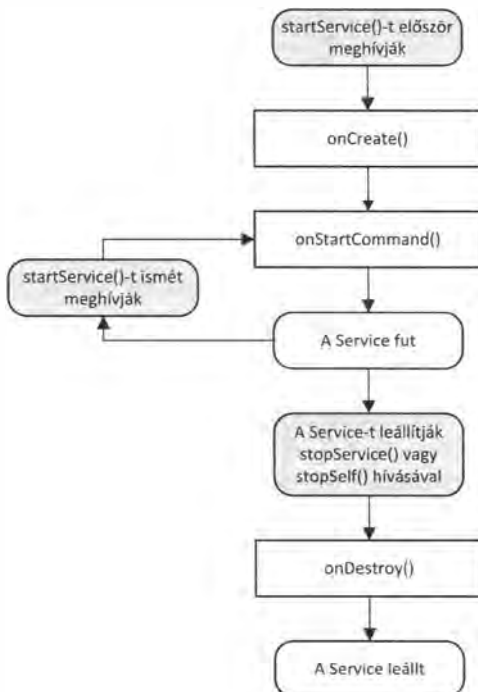
A *Started Service*-ek a leggyakrabban használt *Service*-ek, ezeken keresztül mutatjuk be a *Service*-ek programozásának alapjait. Az itt ismertetett technikák egy része ugyanúgy alkalmazható a *Bound Service*-eknél is.

A *Started Service* elnevezés onnan ered, hogy a programozó a *Service*-t explicit módon indítja el a *startService()* metódus meghívásával. Innen kezdve a *Service* egészen addig létezik, amíg a programozó le nem állítja a *stopService()* hívásával. Olyankor érdemes használni őket, amikor egy kezdeti indító parancs kiadása után nincs szükségünk folyamatos kommunikációra a *Service*-szel. Ilyen lehet egy fájl letöltésének vagy zenelejátszásnak az indítása.

A *startService()* és a *stopService()* az általános *Context* osztályban található metódusok. A *Context*-ből származik mind az *Activity*, mind a *Service* osztály, így ezen két komponens tetszőleges példányán keresztül indíthatunk *Service*-t.

Legtöbbször a *Started Service*-ek által végrehajtott műveleteknek nincs „visszatérési” értéke. Ez nem azt jelenti, hogy nem kommunikálhatnak a hívó vagy tetszőleges egyéb komponenssel, de folyamatos kérdés-válasz jellegű kommunikációra sokkal inkább alkalmasak a *Bound Service*-ek.

A következőkben végigkövetjük egy *Started Service* életciklusát a létrehozásától kezdve a leállításáig. Az egyes lépéseket és a meghívódó callback metódusokat a következő ábra szemlélteti.



11.2. ábra. A *Started Service* életciklusa

11.2.1. A Service indítása

A *Started Service* indítása a *startService()* metódus meghívásával történik, amely egy *Intent*-et vár paraméterül. Az *Intent*-en keresztül adjuk meg, hogy melyik *Service*-t szeretnénk elindítani, és ezen keresztül van lehetőségünk megmondani, hogy milyen parancsot adunk a *Service*-nek. Egy lokális *Service* (*MyService*) indítása a következőképpen történik:

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

A *Service* indításhoz használatos *Intent* első paramétere az aktuális kontextus (*Context*), második paramétere pedig a *Service* osztálya.

Egy álló *Service* a *startService()* legelső hívásakor jön létre, ekkor meghívódik a *Service* *onCreate()* metódusa. Ez a callback hívás csakis a *Service* példányosításakor, a legelső *startService()* híváskor következik be. Itt van lehetőségünk az általános inicializálásra.

```
@Override
public void onCreate() {
    super.onCreate();
    // Service inicializálása
}
```

Vegyük figyelembe, hogy *Started Service* esetén a létrejött *Service* objektum egész addig foglalja a memóriát és az egyéb erőforrásokat, amíg a programkódból le nem állítják, még akkor is, ha a *Service*-nek egyébként semmilyen szolgáltatását nem használják. Éppen ezért *onCreated()*-ben csak olyan erőforrásokat foglaljunk le, amelyekre a teljes *Service*-életciklus folyamán szükség van.

Ha a *Service* már létezik, vagy éppen létrejött (az *onCreated()* lefutott), akkor a *startService()* hatására meghívódó következő callback metódus az *onStartCommand()* lesz. Itt kell elindítanunk azt a műveletet, amely miatt a *Service*-hez fordultak a hívó oldalról. Alapesetben a kód az alkalmazás UI szálán fut, úgyhogy hosszabb műveleteket külön szálon vagy egy *AsyncTask*-on keresztül futtassunk. Paraméterként megkapjuk a *startService()* meghívásánál átadott *Intent*-et, néhány plusz információs flaget tartalmazó bitmezőt (*int flags*) és egy egyedi azonosítót (*int startId*), amely minden *startCommand()* hívást követően új érték lesz.

```
@Override
public int onStartCommand(Intent intent, int flags,
```

```

        int startId) {
    // Service műveletek elindítása...

    return super.onStartCommand(intent, flags, startId);
}

```

Az *onStartCommand()* visszatérési értéke egy *int*, amellyel megmondhatjuk, hogy a Service milyen módon induljon újra, ha a rendszer leállította. A visszaadott kód lehetséges értékei a következők:

- **Service.START_STICKY:** A leállított Service-t a rendszer mindig megpróbálja újraindítani. Az újraindításkor meghívódó *onStartCommand()*-ban kapott Intent paraméterértéke *null* lesz.
- **Service.START_NOT_STICKY:** A leállított Service nem indul újra, amíg a leállítást követően a programkódból újra nem indítják *startService()* meghívásával.
- **Service.START_REDELIVER_INTENT:** Hasonlóan *START_STICKY*-hez a rendszer újraindítja a Service-t, viszont ebben a módban azt az Intentet küldi el újra az *onStartCommand()*-nak, amellyel a leállított Service-t eredetileg elindították.

Android 2.0 (API level 5) előtt *onStartCommand()* helyett *onStart()* metódus van, amely hasonlóan működik, mint az *onStartCommand()*, csak éppen mindig *START_STICKY*-vel tér vissza.

A *startService()*-t többször is meghívhatjuk ugyanarra a Service-re, minden egyes hívás újra *onStartCommand()* lefutását idézi elő.

A Service indítása aszinkron módon történik, vagyis a *startService()* nem „blokkol”, meghívása után folytatódik a végrehajtás a szálon. A Service példányosítására és az *onStartCommand()* hívására egy különálló eseményként tekinthetünk.

11.2.2. Started Service leállítása

Egy *Started Service* akkor áll le, ha a rendszer memóriafelszabadítás céljából kilövi, vagy ha programkódból leállítják a *stopService()*, a *stopSelf()* vagy a *stopSelfResult()* hívásával.

Ha nem a *Service* osztály kódjából állítjuk le a Service-t, akkor *stopService()*-t kell használnunk, amely a *startService()*-hez hasonlóan egy Intentet vár paraméterül, amely azonosítja a leállítandó Service-t.

```

Intent intent = new Intent(this, MyService.class);
stopService(intent);

```

A megadott Service a `stopService()` hatására leáll, a Service-példány törlődik. A `stopService()` nem veszi figyelembe, hogy a Service-hez hány `startService()` hívás érkezett, egyetlen `stopService()` hívás akkor is törli a Service-t, ha éppen több helyen is használják.

Alternatív lehetőség, hogy a Service-példány saját magát állítja le a `stopSelf()` metódus meghívásával. Ezt a lehetőséget érdemes használni például akkor, ha a Service egy hosszabb műveletet hajt végre, amelynek befejezéséről értesülve leállíthatja saját magát. Hasonlóan a `stopService()`-hez, a `stopSelf()` is minden esetben leállítja a Service-t, függetlenül a `startService()` hívások számától.

Kicsivel jobban beleszólhatunk a Service leállítási folyamatába a `stopSelfResult()` metódussal, amely egy Service-azonosítót (`int startId`) vár paraméterül, és csak akkor állítja le a Service-t, ha a kapott azonosító egyezik a Service legutóbbi indítását követő `onStartCommand()`-ban kapott egyedi azonosítóval. Ha nem a legfrissebb azonosítót adjuk meg, akkor a hívásnak nem lesz következménye, és a Service nem áll le.

A Service a leállítás tényéről az `onDestroy()` callback metóduson keresztül kap értesítést. Ez az utolsó értesítés, amelyet a Service a kitörlése előtt kap. Itt felszabadíthatjuk a lefoglalt erőforrásokat: leállíthatjuk az esetleges háttérszálakat, lemondhatjuk a regisztrált listenereket.

```
@Override
public void onDestroy() {
    // Szálak leállítása, listener-ek lemondása...

    super.onDestroy();
}
```

11.2.3. Kommunikáció a Service-szel

A Service egy különálló alkalmazáskomponens, amelyet aszinkron módon érünk el, ráadásul akár külön processzben is futhat. Ezekből az következik, hogy az esetek túlnyomó többségében nem közvetlenül a Service-példányon keresztül érjük el a Service-t. Azt már láthattuk, hogy a Service-nek való adatküldéshez leggyakrabban Intenteket használunk a `startService()` hívásánál. Arról azonban még nem esett szó, hogy egy Service hogyan tud adatokat visszaküldeni a hívó félnek. Ennek több lehetséges módja is van, ezek közül itt hármat ismertetünk: *Broadcast Intent*ek, *Messenger* és *Pending Intent*.

11.2.3.1. Broadcast Intent

Broadcast Intent küldésekor egy Intentben tároljuk el a Service által kiküldendő adatokat, majd ezt elküldjük a `sendBroadcast()` metódussal. A fogadáshoz

írnunk kell egy *Broadcast Receiver*-t, amelyet be kell regisztrálnunk a *registerReceiver()*-rel és egy *IntentFilter* megadásával. Az adatokat a *Receiver onReceive()* metódusát felüldefiniálva tudjuk kiolvasni az *Intent*ből.

Ügyeljünk rá, hogy ha egy *Activity onResume()* és *onPaused()* metódusában beregisztráljuk, illetve kiregisztráljuk a *Receiver*-t, csak akkor kapja meg a *Service* által küldött *Intent*eket, amikor az *Activity* éppen fut.

11.2.3.2. Messenger és Handler

A *Handler* osztály lehetővé teszi üzenetek (*Message*) és végrehajtandó kódrészek (*Runnable*) küldését egy adott szál üzenetkezelő sorába. Egy *Handler* mindig egy adott szálhoz tartozik. A *Handler*nek átadott üzenetek feldolgozása a *handleMessage()* metódusban történik. Egy *Handler* közvetlenül csak azon a processzen belül érhető el, amelyhez tartozik, ha más processzből szeretnénk egy adott szál *Handler*ének üzeneteket küldeni, akkor szükségünk van egy *Messenger* objektumra. A *Messenger* hidat képez a hívó processz és a cél-*Handler* között: a *Messenger*nek átadott üzenetek a hozzákapcsolt *Handler*hez érkeznek meg.

Service-ek esetén átadhatunk egy lokális *Handler*hez kapcsolt *Messenger*-t a *Service*-t elindító *Intent*nek. Ezt a *Service* eltárolja, majd ezen keresztül képes üzenetek küldésére a hívó félnek.

A *Messenger* használata lehetővé teszi, hogy egy külön processzben futó *Service*-ből küldjünk üzeneteket a *Service*-t indító szálnak.

11.2.3.3. Pending Intent

Ha egy *Activity*nek szeretnénk elküldeni a *Service* által generált adatokat, akkor használhatjuk az *Activity createPendingResult()* metódusát, amely egy *Pending Intent*tel tér vissza. Ezt a *Pending Intent*et kell átadnunk a *Service*-nek, ezt a legegyszerűbben a *Service* indításához használt *Intent* extraattribútumaként megadva (*putExtra()*) tehetjük meg. A *Service* eltárolja a *Pending Intent*et, majd ha elvégezte a kívánt műveletet, ezen keresztül tud visszajelezni és adatokat átadni a hívó félnek: a *Pending Intent* valamelyik *send()* metódusának hatására az *Activity onActivityResult()* metódusa hívódik meg, ahol az *Intent*ből kiolvashatja az átadott adatokat.

11.2.4. Egy egyszerű Started Service-példa

Ebben a példában egy nagyon egyszerű *Service*-t készítünk el, amely elindítását követően külön szálaban futtat egy olyan végtelen ciklust, amely másodpercenként egy *Broadcast Intent*ben kiküldi az aktuális időt. Erre az *Intent*re iratkozunk fel egy *BroadcastListener*rel, amely az alkalmazás egyetlen *Activity*jéhez tartozó *TextView*-n keresztül kiírja a kapott időt a képernyőre.

Bár a gyakorlatban nem sok hasznát vennénk ennek a nagyon egyszerű Service-nek (másodpercenkénti időjelzést ennél sokkal egyszerűbben is leprogramozhatunk), a megírásán keresztül megérthetjük a Service-életciklus folyamatát, és láthatjuk, hogy hogyan tudunk egyszerűen kommunikálni az alkalmazás többi komponensével.



11.3. ábra. Egy Activity, amelyen keresztül elindíthatjuk és leállíthatjuk a Service-t. A Service által kiküldött időt is ez az Activity kapja el és jeleníti meg

Az első lépés egy új, Service-ből származó osztály létrehozása és ennek beregisztrálása az alkalmazáskomponensek között a *manifest* állományban.

```
<application>
    ...
    <service android:name="TimerService"/>
</application>

public class TimerService extends Service {
}
```

Mielőtt a Service callback metódusait implementálnánk, belső osztályként definiálunk egy szálát (*TimerThread*), amelyet a Service indítását követően futtatunk le. Ez a szál tekinthető a Service törzsének, itt történik az érdemi

munka és a Service-hez érkezett kérés kiszolgálása. Esetünkben a szálnak egy végtelen ciklusban, másodpercenként egy-egy új *Broadcast Intent* kiküldésén túl nincsen más feladata. A kiküldött Intentelekhez felvesszünk két statikus *String*-típusú tagváltozót (*ACTION_TIME_CHANGED* és *EXTRA_TIME*), amelyek közül az előbbi a *Broadcast Intent* akciójának azonosítója (ezen keresztül tudunk majd feliratkozni rá), utóbbi pedig az Intentben kiküldött időparaméter elérését teszi lehetővé.

```
// Broadcast Intent-hez tartozó azonosítók
public static final String ACTION_TIME_CHANGED =
    "TimerService.ACTION_TIME_CHANGED";
public static final String EXTRA_TIME =
    "TimerService.EXTRA_TIME";

// Jelzi, hogy a Service-hez tartozó szál fut-e vagy
sem
private volatile boolean running = false;

// A Service-hez tartozó háttérszál
private class TimerThread extends Thread {

    public void run() {
        while (running) {
            try {
                sleep(1000);
                String currentTime =
                    new Date(System.currentTimeMillis())
                        .toLocaleString();

                Intent broadcastIntent = new Intent();

                broadcastIntent.setAction(
                    TimerService.ACTION_TIME_CHANGED);

                broadcastIntent.addCategory(
                    Intent.CATEGORY_DEFAULT);
                broadcastIntent.putExtra(
                    TimerService.EXTRA_TIME, currentTime);
                sendBroadcast(broadcastIntent);

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

A szálhoz tartozó *run()* metódusban lévő *while* ciklus egész addig fut, amíg a *Service* tagváltozójaként definiált *running* mező értéke igaz. Ezt a *Service* indításakor állítjuk be. A ciklus magja egy másodperces várakozással indul (*sleep(1000)*), majd ezt követően lekérjük az aktuális időt egy *String*-be, felpamétezzük egy *Broadcast Intent*-et, és kiküldjük a nagyvilágba.

Ekkor következhetnek a *Service*-életciklus callback metódusai. Mivel a *Service*-ünket *Started Service*-ként szeretnénk használni, a *Service* *onBind()* metódusában *null*-al térünk vissza. Ennek a metódusnak minden esetben kötelező a felüldefiniálása.

```
@Override
public IBinder onBind(Intent intent) {
    return null;
}
```

A *Service* indításakor meghívódó *onStartCommand()*-ban elindítjuk a szálát, és beállítjuk a szál futását kontrolláló *running* mező értékét. Mivel a *Service*-ben mindig csak egyetlen szálát szeretnénk futtatni, függetlenül attól, hogy hányszor hívták meg *startService()*-t, csak akkor indítunk új szálát, ha előzőleg ezt még nem tettük meg.

```
@Override
public int onStartCommand(Intent intent, int flags,
                           int startId) {
    if (!running) {
        running = true;
        new TimerThread().start();
    }
    return super.onStartCommand(intent, flags, startId);
}
```

Végül gondoskodnunk a háttérzál leállításáról a *Service* leállításakor. A *Service* leállításakor *onDestroy()* hívódik, amelyben a *running* mező értékét *false*-ra állítva gondoskodunk arról, hogy a szálunk is kilépjen a *while* ciklusból, és végül lezáródjon.

```
@Override
public void onDestroy() {
    running = false;
    super.onDestroy();
}
```

Az alkalmazás egyetlen Activityt tartalmaz (*ServiceDemoActivity*), ennek felülete két gombot tartalmaz a Service elindításához és leállításához. Még egy *TextView*-t is használunk a Service-től beérkező idő megjelenítéséhez.

A Service által küldött *Broadcast Intent*ek fogadásához egy *Broadcast Listener* osztályra van szükségünk, amelyet a példában egy az Activityhez tartozó belső anonim osztályban definiálunk. Ebből egy példányt eltárolunk a *ServiceDemoActivity responseReceiver* mezőjében. Az Intent fogadásakor meghívódik az *onReceive()* metódus, amelyben beállítjuk a felületen megjelenő *TextView*-t a kapott szövegre.

```
public class ServiceDemoActivity extends Activity {
    private BroadcastReceiver responseReceiver =
        new BroadcastReceiver() {

        @Override
        public void onReceive(Context context, Intent
intent) {
            TextView result =
                (TextView) findViewById(R.id.info_text_view);
            String text =
                intent.getStringExtra(TimerService.EXTRA_
TIME);
            result.setText(text);
        }
    };
}
```

A *BroadcastReceivert* be kell regisztrálni, ezt megtehetjük a *manifest* állományban vagy a forráskódban. Esetünkben az utóbbi megoldást használjuk: az Activity *onCreate()* metódusában hozunk létre egy *IntentFilter*t a Service-hez tartozó akcióhoz (*TimerService.ACTION_TIME_CHANGED*), és ezzel regisztráljuk be a mezőként eltárolt *TimerResponseReceiver* példányt.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    IntentFilter filter = new
        IntentFilter(TimerService.ACTION_TIME_CHANGED);
    filter.addCategory(Intent.CATEGORY_DEFAULT);
    responseReceiver = new TimerResponseReceiver();
    registerReceiver(responseReceiver, filter);
}
```

11.2.5. *IntentService*

Gyakran merül fel az igény olyan Service-re, amely a Service indítása után egy külön szálon végzi a műveleteket, hogy ne a fő szál blokkoljuk. Az ilyen Service-ek írását egyszerűsíti le az *IntentService* osztály, amely megkönnyíti a szálkezelést.

Az *IntentService*-ből származó osztályok a *startService()* hívását követően automatikusan egy új háttérszál indítanak, amelyben meghívják az osztály *onHandleIntent()* metódusát. Ezt felüldefiniálva lehetőségünk van elvégezni a Service feladatait anélkül, hogy aggódnunk kéne a program blokkolása miatt, hiszen az *onHandleIntent()* már a háttérszálon fut.

IntentService használatakor nem kell felüldefiniálnunk az *onStartCommand()*-ot. Ha ezt mégis megtennénk, akkor mindenképpen hívjuk meg az ősoosztály (super) *onStartCommand()* metódusát, ugyanis ez gondoskodik a beérkező kérések sorba állításáról és a háttérszál elindításáról.

Az *IntentService* azon túl, hogy egy külön szálon hívja az *onHandleIntent()*-et, rendelkezik még egy hasznos funkcióval: a *startService()* hívásokból berkező Intenteket egy várakozási sorba helyezi. Ez azt jelenti, hogy bár egyszerre mindig csak egy Intent feldolgozása történik, a közben beérkezett *startService()* hívásokra is sorban meg hívódik az *onHandleIntent()*.

11.2.6. Példa az *IntentService* és a *Messenger* használatára

A következő példában egy fájlok letöltését végző Service-t készítünk, amely egy *Messenger*-en keresztül jelez vissza a hívó Activitynek. A Service-hez az *IntentService* ősoosztályt használjuk, amely éppen megfelel egy letöltéseket végző szolgáltatásnak. A Service egyszerre tetszőleges számú letöltési kérést fogadhat, de ezek közül mindig csak egyet szolgál ki, a többit szép sorban ütemezi be, mindig az előző letöltés befejezését követően. Mindezen túl a letöltések elvégzéséhez automatikusan kapunk egy külön szálát, így háttérszál létrehozásával sem kell bajlódni.

Első lépésként megírjuk a *Service* osztály vázát. Felveszünk egy mezőt a legutóbbi letöltés eredményének a tárolásához (*int result*), valamint felüldefiniáljuk az *onBind()* metódust, amelyben *null*-al visszatérve jelezzük, hogy a Service-t nem használhatjuk „bound” üzemmódban. Egy konstruktort is definiálnunk kell, mert az *IntentService* konstruktorának kötelező átadnunk a Service nevét.

```
public class DownloaderService extends IntentService {
    private int result = Activity.RESULT_CANCELED;

    public DownloaderService() {
        super("DownloaderService");
    }
}
```

```

@Override
public IBinder onBind(Intent intent) {
    return null;
}

```

A munka érdemi részét az *onHandleIntent()* metódus végzi, amelyben le-töltjük az adatokat a paraméterként megadott Intentből kiolvasott URL-ről. Az *onHandleIntent()* ekkor már külön szálon fut. A példában a letöltött adatok elmentésével vagy eltárolásával nem foglalkozunk, a művelet befejezésekor csak a letöltés sikerének vagy sikertelenségének tényét küldjük el a *Messenger*-en keresztül. A *Messenger*-t, amelyen keresztül visszajelezhetünk a hívó fél-nek, ugyancsak a kapott Intentből olvassuk ki. Létrehozunk egy elküldendő üzenetet (*Message.obtain()*), amelynek argumentumaként beállítjuk a letöltés állapotát. Végül a *Messenger.send()* metódusával történik az üzenet elküldése.

Message-ben tetszőleges *Parcelable* típus eltárolható *setData()*-val, ám csak ha *int*-eket sze-retrénnk átadni, akkor használhatjuk a könnyen és gyorsan hozzáférhető *arg1* és *arg2* mezőket.

```

@Override
protected void onHandleIntent(Intent intent) {
    int result = Activity.RESULT_CANCELED;
    InputStream stream = null;
    try {
        URL url = new URL(intent.getStringExtra("EXTRA_
URL"));
        stream = url.openConnection().getInputStream();
        InputStreamReader reader = new
InputStreamReader(stream);
        int readData = -1;
        while ((readData = reader.read()) != -1) {
            // Adatok elmentése vagy eltárolása
        }

        result = Activity.RESULT_OK;
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (stream != null) {
            try { stream.close(); }
            catch (IOException e) { e.printStackTrace(); }
        }
    }
}

```

```

// Értesítés küldése a hívó félnek Messenger-en ke-
resztül
Bundle extras = intent.getExtras();
if (extras != null) {
    Messenger messenger =
        (Messenger) extras.get("EXTRA_MESSENGER");
    Message msg = Message.obtain();
    msg.arg1 = result;
    try { messenger.send(msg); }
    catch (android.os.RemoteException e) {
        e.printStackTrace();
    }
}
}
}

```

A Service-ből jövő üzenetek fogadásához szükségünk van egy *Handler* példányra. Ezt a példában a hívást végző Activity mezőjében vesszük fel anonim osztályként (*Handler handler*). *Handler*ben a *handleMessage()*-ben kapjuk meg az üzeneteket, esetünkben a Service visszajelzését, ez a legutóbbi letöltés állapotáról tájékoztat. Az üzenetből kiolvasott eredménynek megfelelő *Toast* értesítést jelentünk meg.

```

private Handler handler = new Handler() {
    public void handleMessage(Message message) {
        if (message.arg1 == RESULT_OK)
            Toast.makeText(ServiceDemoActivity.this,
                "Sikeres letöltés", Toast.LENGTH_LONG).show();
        else
            Toast.makeText(ServiceDemoActivity.this,
                "Sikertelen letöltés", Toast.LENGTH_LONG).
            show();
    }
};

```

Végül következzen a kódrészlet a Service elindításához. Itt hozzuk létre a Service-nek átadott *Messenger*-t, amelyet a lokális *Handler*rel inicializálunk, ezen keresztül küldi el a Service az üzeneteket.

```

Intent intent = new Intent(this, DownloaderService.
    class);

Messenger messenger = new Messenger(handler);

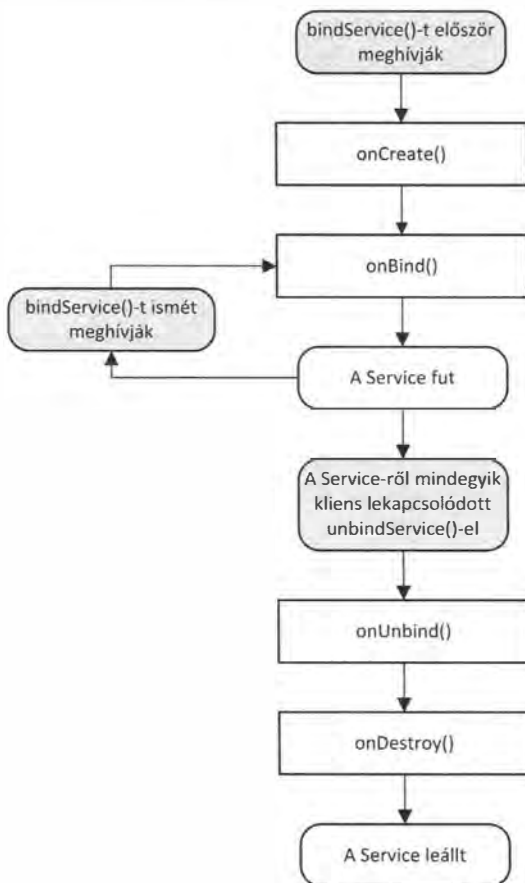
```

```
intent.putExtra("EXTRA_MESSENGER", messenger);  
intent.putExtra("EXTRA_URL",  
    "http://www.example.com/file.txt");  
  
startService(intent);
```

Bár a példában nem foglalkoztunk külön a processzek kérdésével, tudni kell, hogy a Service-nek és a hívó Activitynek nem kell ugyanabban a processzben futnia, hiszen az átadott *Messenger* lényege pontosan az, hogy átvéli a processzhatárokat. *Messenger*t akkor is használhatunk, ha végig egy processzben dolgozunk, de ez esetben több más egyszerűbb lehetőségünk is van az üzenetek átadására.

11.3. Bound Service

Az eddigi példákban azt láthattuk, hogy hogyan tudunk Service-eket elindítani és parancsokat küldeni nekik a *startService()*-szel. A Service-ek megszólításának a másik módja az, ha hozzájuk kapcsolódunk a *bindService()* hívásával. Ez esetben, miután a Service-hez kapcsolódunk, kapunk egy *IBinder*ből leszármazó interfészt, amelyen keresztül kommunikálhatunk a Service-szel. Ennek az interfésznek a definiálása a Service megírásakor történik, és ez határozza meg, hogy a Service milyen funkcióit éri el a hívó fél. Ha a Service és a kapcsolódó fél ugyanabban a processzben fut, akkor a *Binder* akár a Service-példány referenciáját is átadhatja, így közvetlenül hívhatók a Service-példány metódusai. Ha a Service külön processzben fut, akkor a *Bindertől* kapott *Messengeren* keresztül küldhetünk üzeneteket a Service-nek.



11.4. ábra. A Bound Service életciklusa

A *Bound Service* életciklusát a fenti ábra illusztrálja. Az első szembetűnő különbség a *Started Service*-hez képest, hogy *bindService()*-szel szólítjuk meg a *Service*-t. Ez a metódus három paramétert vár:

- **Intent service:** Azonosítja a *Service*-t, amelyhez kapcsolódni szeretnénk (hasonlóan mint *startService()* esetében).
- **ServiceConnection conn:** Mivel a *bindService()* aszinkron művelet, szükség van valamilyen callback mechanizmusra, hogy a rendszer értesítést küldhessen a sikeres kapcsolódásról. Erre szolgál a *ServiceConnection* osztály, amely a callback metódusain keresztül jelez a *Service* sikeres vagy sikertelen kapcsolódásakor. (Ennek használatát lásd később.)
- **int flags:** Ezek a kapcsolódáshoz tartozó beállítások, flagek formájában. Itt adhatjuk meg például a *BIND_AUTO_CREATE* flaget, amelynek hatására a *Service* automatikusan példányosodik, ha még nem létezik a *bindService()* hívásakor.


```
bindService(
    new Intent(this, MyBoundService.class),
    timerServiceConnection,
    Context.BIND_AUTO_CREATE);
```

Ha a flagek között megadtuk a *BIND_AUTO_CREATE*-et, és a Service még nem létezik a *bindService()* hívásakor, akkor ennek hatására példányosodik (hasonlóan a *startService()*-nél tapasztaltakhoz). Ezt követően a rendszer meghívja az *onBind()* callback metódust (a *bindService()* után nem hívódik meg *onStartCommand()*). Ennek a metódusnak az a feladata, hogy visszatérjen egy *IBinder* interfészt megvalósító osztály példányával, amelyet a Service-hez kapcsolódó kliens megkaphat. Az *IBinder* megvalósítása elég sok munkát igényel (az interfész 9 absztrakt metódust definiál), de szerencsére rendelkezésre áll egy kész alapimplementáció a *Binder* osztály formájában, amely a legtöbb alapfeladatra megfelel.

A legegyszerűbb esetben, amikor a Service és a kliens is egy processzben fut, a *Binder* egyszerűen átadhat egy referenciát magára a Service-re. Az alábbi kódrészletben definiált *TimerServiceBinder* osztály egyetlen új metódusa (*getService()*) visszatér magával a Service példányával. A Service eltárol egy *TimerServiceBindert* egy Tagváltózában, és ezzel tér vissza *onBind()*-ban.

```
public class TimerServiceBinder extends Binder {
    TimerService getService() {
        return TimerService.this;
    }
}
private final IBinder binder = new
TimerServiceBinder();

@Override
public IBinder onBind(Intent intent) {
    return binder;
}
```

Ahhoz, hogy a Service-t megszólító kliens megkapja a Bindert, definiálni kell egy *ServiceConnection*ből származó saját osztályt, és meg kell valósítani annak *onServiceConnected()* és *onServiceDisconnected()* metódusait. Ezen *ServiceConnection* egy példányát kell átadni a *bindService()* meghívásakor.

A *Binder* példányt, amelyet a Service *onBind()* visszatérési értékeként adott meg, az *onServiceConnected()* metódus paramétereként kapjuk meg. Az alábbi példában látható, hogy hogyan kérhetjük el a *Bindertől* a Service-példányt, hogy közvetlenül használhassuk:

```

private ServiceConnection timerServiceConnection = new
ServiceConnection() {
    public void onServiceConnected(ComponentName
className,
                                IBinder binder) {
        TimerService service =
            ((TimerService.TimerServiceBinder)binder).
getService();
        // service metódusainak hívása...
    }

    public void onServiceDisconnected(ComponentName
className) {
        // service kapcsolat megszakadt
    }
};

```

A Service-szel való kapcsolat az *unbindService()* metódus hívásával szakítható meg. A *bindService()*-szel megszo lított Service egész addig létezik, amíg még legalább egy aktív kapcsolat van hozzá. Ha mindegyik *Binding*ot megszakították kív lr l, akkor a Service-t a rendszer automatikusan kit rli a mem riából.

Ha a Service-hez kapcsolódó kliens egy Activity, és az Activity saját *bindService()* metódusát használjuk, akkor a kapcsolat megszakad a konfigurációváltásokkor. Ennek elker lésére kérj k le az *Application Context*et a *getApplicationContext()* hívásával, és ezen keresztül kapcsolódjunk a Service-hez, így a *Binding* akkor is megmarad, ha az Activity-példány törlődik.

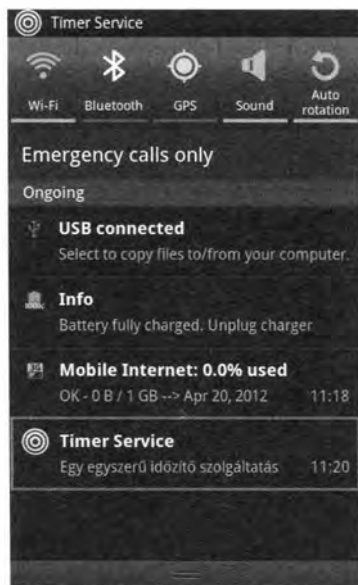
A külön processzben futó Service-ekhez való kapcsolódással nem foglalkozunk részletesen. Az Android lehetővé teszi a Service-szel való kommunikációt biztosító *Binder* automatikus legenerálását úgynevezett AIDL- (Android Interface Definition Language) f jlok segítségével. Az AIDL-f jlokban megadhatjuk azt az interfészt, amelyen keresztül a Service-szel kommunikálni szeretn nk, az Android SDK-hoz kapott eszközökkel pedig legenerálhatjuk az ennek megfelel , *IBinder*t megvalósító osztályt. Az így kapott *Binder* gondoskodik a processzek közötti kommunikáció (IPC) lebonyolításáért. A *Bindert* használva tetsz leges *Parcelable* interfészt megvalósító osztályok példányai adhatók át, a generált kód gondoskodik ezek sorosításáról és a processzek közötti átk ldésér l.

11.4. Előtérben futó Service-ek

Az androidos készüléken futó közönséges Service-ek csak a készülék „Settings/Applications/Running services” menüjében ellenőrizhetők. Erről az átlagos felhasználó vagy nem is tud, vagy ha tud is, akkor sem feltétlenül van tudatában annak, hogy melyik Service miért is felel. Ezzel szemben az előtérben futó Service-ek olyan kiemelt szolgáltatások, amelyek egy *Notification* formájában mindig láthatók a rendszerhez tartozó értesítéseket listázó ablakban. Az ilyen Service-ek futásának a felhasználó mindig „tudatában van”, és a *Notification* kiválasztva, a hozzárendelt *Pending Intent*en keresztül, elindíthat egy a Service-hez kapcsolódó Activityt. Előtérben futó Service lehet például egy zenelejátszó vagy egy fájlletöltést végző szolgáltatás.

Felmerülhet a kérdés, hogy miért foglalkozunk külön az előtérben futó Service-ek fogalmával, hiszen ezeket látszólag csak egy plusz *Notification* különbözteti meg a közönséges Service-ektől. Ennél azonban nagyobb a különbség, ugyanis az előtérben futó Service-eket a rendszer kiemelten kezeli. Mivel az ilyen szolgáltatások feltételezhetően fontosak a felhasználó számára, ezért kevés memória esetén csak később állnak le. Amíg még vannak lekapcsolható közönséges Service-ek, addig a rendszer igyekszik ezeket leállítani és nem bántani az előtérben futó szolgáltatásokat.

Bármilyen Service-t lehet az előtérben futtatni, mindössze a Service *startForeground()* metódusát kell meghívni és átadni egy *Notification*-t, amely jelzi a Service futását. A *Notification* egész addig látható marad az értesítések között, amíg a Service fut, vagy nem váltották vissza közönséges Service-szé a *stopForeground(true)* hívással.



11.5. ábra. Az előtérben futó Service-hez tartozó Notification. A Notification egészen addig látható marad az értesítések között, amíg a Service fut

A következő példa azt mutatja, hogy hogyan tudjuk a *Service*-ünket előtérben futóvá változtatni a *Service onCreate()* metódusának módosításával. Ennek hatására a *Service* példányosításától kezdve látható lesz a *Notification* (lásd a fenti ábrát).

A *startForeground()* a *Service*-életciklus tetszőleges pontján hívható, nem csak az *onCreate()*-ben. Bizonyos esetekben érdemes lehet a *Service*-t csak a beérkező parancsok függvényében, az *onStartCommand()*-ban az előtérbe tenni. Az előtérben futó *Service* bármikor visszaváltható közönséges *Service*-szé a *stopForeground()* meghívásával.

Először létrehozunk egy *Notification* példányt, a konstruktorának pedig átadjuk az értesítés ikonjának azonosítóját (*R.drawable.icon*), az indításkor a *status bar*on megjelenítendő „ticker” feliratot („Timer Service”), valamint az értesítéshez tartozó időt. Ezt követően felparaméterezünk egy *Intent* példányt, amelyet hozzárendelünk a *Notification*höz. Az *Intent* a *ServiceDemoActivity*-t indítja el, amikor a felhasználó megérinti a *Notification*höz tartozó bejegyzést. A *Notification* összeállítását követően szükségünk lesz még egy, az alkalmazáson belüli egyedi értesítésazonosítóra (*notificationId*), amelynek 0-n kívül tetszőleges számot választhatunk. Végül meghívjuk a *startForeground()*-ot, amelynek hatására megjelenik a *Notification*, és a rendszer bejegyzi előtérben futónak a *Service*-t.

```
@Override
public void onCreate() {
    Notification notification =
        new Notification(R.drawable.icon, "Timer Service",
            System.currentTimeMillis());

    CharSequence contentTitle = "Timer Service";
    CharSequence contentText = "Egy egyszerű időzítő
                                szolgáltatás";

    Intent notificationIntent =
        new Intent(this, ServiceDemoActivity.class);
    notificationIntent.setFlags(
        Intent.FLAG_ACTIVITY_CLEAR_TOP |
        Intent.FLAG_ACTIVITY_SINGLE_TOP);
    PendingIntent contentIntent =
        PendingIntent.getActivity(this, 0,
notificationIntent, 0);
    notification.setLatestEventInfo(this, contentTitle,
                                contentText,
contentIntent);

    final int notificationId = 1234;
    startForeground(notificationId, notification);

    super.onCreate();
}
```

Az Android 2.0 előtt nem volt kötelező *Notification*-t megadni az előtérben futó Service-ekhez, a *startForeground()* csak egy jelzés volt a rendszer felé, hogy a Service-t kiemelten kell kezelni. Ez azonban könnyen visszaélésekhez vezethetett: mivel az előtérben futó Service-eket később állítja le a rendszer, gyakori volt, hogy szinte minden Service-t az előtérben futtattak, függetlenül attól, hogy erre ténylegesen szükség volt-e. Az így létrehozott Service-ek megtöltötték a memóriát, és a felhasználó még csak nem is tudta egykönnyen ellenőrizni vagy leállítani őket. Ennek kiküszöbölésére vezették be a kötelezően megadandó *Notification*-öket. Ezek révén a felhasználó számára is láthatók az előtérben futó szolgáltatások, amelyeket szükség esetén le is állíthat.

11.5. Alkalmazáskomponens automatikus elindítása a készülék indulása (boot) folyamán

Eddig csak olyan Service-ekre láttunk példát, amelyeket explicit módon, a kódból indítottunk a *startService()* vagy a *bindService()* hívásával. Ám gyakran van szükség olyan szolgáltatásokra, amelyek a telefon indulását követően automatikusan elindulnak, és folyamatosan futnak a háttérben. Ehhez egy *BroadcastReceiver*-rel fel kell iratkoznunk a *BOOT_COMPLETE* eseményre, és ezt lekezelve elindíthatjuk a kívánt Service-t.

Bár a példában egy Service elindítását mutatjuk be, ugyanígy indíthatunk el egy Activityt is.

A *BOOT_COMPLETE* eseményre való feliratkozást legegyszerűbben az alkalmazás *manifest* állományában tehetjük meg. Itt egy *receiver* elemben adhatjuk meg a kívánt *BroadcastReceiver* osztály nevét és a hozzárendelt *Intent Filter*-t.

```
<receiver android:name=".
BootCompleteBroadcastReceiver">
  <intent-filter>
    <action android:name=
      "android.intent.action.BOOT_COMPLETED" />
    <category android:name=
      "android.intent.category.HOME" />
  </intent-filter>
</receiver>
```

A *BroadcastReceiver*-ben az *onReceive()* callback metódus hívódik meg a beérkező *Intent* hatására. Itt a már megismert módon, például a *startService()* metódussal indíthatjuk el a kívánt Service-t.

```
public class BootCompleteBroadcastReceiver extends
    BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent
intent) {
        Intent startupIntent = new Intent(context,
            MyService.class);
        context.startActivity(startupIntent);
    }
}
```

Vegyük figyelembe, hogy a készülék indulása (*boot*) során elindított Service-eket is leállíthatja a rendszer, ha kevés a memória.

Az Android fejlett funkciói és natív programozása

Az előző fejezetben bemutattuk az Android-szoftverfejlesztés legfontosabb elemeit. Az Android folyamatos fejlődésen és megújuláson megy keresztül, ennek következtében gyakran jelennek meg új megoldások, amelyeket a fejlesztők kihasználhatnak. Az egyik ilyen újdonság a táblagépek megjelenésekor bevezetett *Fragment*, amelyre tulajdonképpen mint egy újrafelhasználható *Activity*-re tekinthetünk. A fejlődés azonban nem állt meg, az Android 4-es verziójának megjelenése is számos újítást hozott, többek között a felhasználói felület terén is. Az újdonságok népszerűségének eredménye az, hogy elérhetővé vált egy hivatalos Compatibility Pack²⁵ is, amelynek segítségével a korábbi Android-verziókon is megvalósíthatók a 3-as verziótól megjelent újdonságok.

Amikor mobilalkalmazásokról beszélünk, nem elegendő csupán a látványos megoldásokra, gazdag funkciókra gondolnunk, hanem szem előtt kell tartanunk az adott platform hatékonyságát, az alkalmazások általános erőforrásigényét is. Az Android platform tervezői erre is gondoltak, és megnyitották a fejlesztők előtt a natív fejlesztés lehetőségét is. Ennek eredményeképpen elérhető egy úgynevezett NDK (Native Development Kit) fejlesztői csomag is, amellyel C++-ban írt natív kódokat futtathatunk az Android-alkalmazásokban. A natív módon futó kódrészeknek az az előnye, hogy kikerülhetők a virtuális gépből eredő terhelések.

Ebben a fejezetben elsőként bemutatjuk a *Fragment*ek használatát, és egy összetettebb példán ismertetjük ezek legfőbb előnyeit. Ezt követően kitérünk az *ActionBar* és a *ViewPager* felületi elemekre, amelyek alapvető fontosságúak a modern Android-alkalmazásokban. Végül pedig ismertetjük a natív alkalmazásfejlesztés módszereit.

12.1. A *Fragment*ek bemutatása

Eddig főként Android-alapú mobiltelefonokról beszéltünk, és nem tértünk ki a táblagépekre. A korábban bemutatott ismeretek ugyanúgy alkalmazhatók a táblagépeken is, egyedül a felhasználói felület tervezésében különböznek lényegesen. Ebben a fejezetben röviden bemutatjuk, hogyan érdemes felhasználói felületet tervezni a táblagépekre úgy, hogy az alkalmazás egyszerre mobiltelefonon is átlátható felületet biztosítson. A bemutatott elvek, módszerek tehát mobiltelefonon is ugyanúgy használhatók.

²⁵ Compatibility library: <http://developer.android.com/sdk/compatibility-library.html>

Mobilalkalmazások esetében megszoktuk, hogy az alkalmazásunk egy vagy több Activityből áll, amelyek között navigálunk, és ezek együttesen adják az alkalmazás funkcionalitását. Táblagépeknél ez a megoldás akkor okoz problémát, ha az Activity felülete önmagában túl kicsi egy táblagép felületéhez képest, és azt az érzést kelti a felhasználóban, hogy nincsen kihasználva a táblagép képernyőmérete. Képzeljük el például az előző fejezetekben bemutatott *ToDo* alkalmazásunkat. Az alkalmazás kezdőnézete egy lista, amelyen a tennivalók láthatók, egy tennivalóra kattintva pedig a kiválasztott teendő részletei jelennek meg.

Egy ilyen megoldás a táblagépek esetében rendkívül rossz lenne, hiszen mind a kezdőlista-nézet, mind pedig a tennivalórészletek nézete önmagában nem olyan nagy, hogy megfelelően kihasználják a táblagépek képernyőméretét, a túl nagyra skálázott felület pedig zavaróan hatna. Ehelyett sokkal célszerűbb lenne egy olyan felület, amelynek bal oldalán a tennivalók listáját látnánk, jobb oldalán pedig az éppen kiválasztott tennivalóelem részletei jelennének meg. Korábbi ismereteink alapján ezt a felületet el is tudnánk készíteni, ha *xlarge* képernyő esetében egy külön *layout* erőforrást definiálnánk, amelyben valóban egy *ListView* és egy tennivalórészleteket megjelenítő felület jelenne meg egymás mellett, ám ez nagyon nagy pluszmunkát jelent, ráadásul az alkalmazáslogikába is bele kellene építeni a képernyőmérettől függő viselkedést.

A fenti megoldás akkor lenne valóban hasznos, ha a két nézetet meg tudnánk különböztetni, különböző logikát tudnánk rendelni hozzá, tehát ha tulajdonképpen egyszerre két Activityt tudnánk megjeleníteni. Ezt a gondolatot az Android fejlesztői is felismerték, és ennek megvalósítására vezették be a *Fragment* fogalmát. A *Fragment*ek tulajdonképpen önálló életciklussal rendelkező al-Activityk, amelyek közül egy időben többet is megjeleníthetünk a felhasználói felületen, rugalmasan kezelhetők, és önálló, elkülönült üzleti logika rendelhető mögéjük.

A *Fragment*ek tehát az Activityhez vannak rendelve, és Activitynként önálló *Back Stack*kel is rendelkeznek, tehát megoldható például, hogy a *Vissza* gomb hatására ne az Activityből lépünk vissza, hanem csak az Activityn belül az előző *Fragment* kerüljön ismét előtérbe, ha már létezett. Ha már nincs több *Fragment*, akkor a megszokott módon az előző Activity kerül előtérbe.

*Fragment*ek felhasználásával megoldható az is, ha külön szeretnénk támogatni a fekvő és az álló nézetet úgy, hogy a fekvő esetében a két *Fragment* egymás mellett jelenjen meg, álló nézeten viszont csak az egyik.



12.1. ábra. *Fragment* elrendezése fekvő és álló nézetben

A *Fragmentek* nagy előnye, hogy önálló felülettel rendelkeznek, és egy Activityn belül elkülönül az általuk megvalósított logika, így rugalmasan rendelkezhetők akár több Activityhez is a futási körülményeknek megfelelően.

12.1.1. A *Fragment* tulajdonságai

Mielőtt egy konkrét példán keresztül bemutatnánk a *Fragmentek* használatát, vizsgáljuk meg a főbb tulajdonságait. Az Activityvel ellentétben a *Fragmentek* nem a *Context*-ből származnak le; nem az Activity egyfajta kiterjesztéséről van szó. Egy *Fragment* osztály megvalósításakor az Activitykhez hasonlóan azt mindig a *Fragment* ősosztályból kell leszármaztatni, amely azonban az *Object* osztály leszármazottja az *android.app* csomagon keresztül.

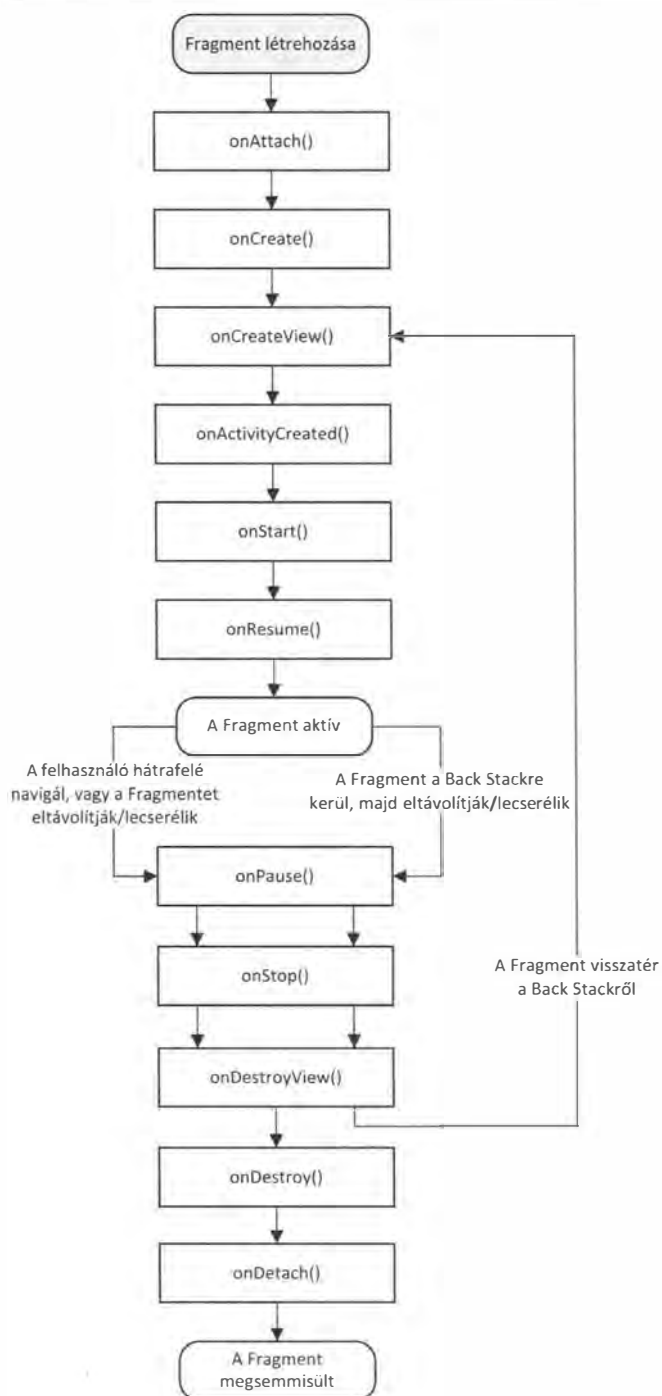
Minden *Fragment*hez egyedi *view* hierarchia rendelhető, amely XML-erőforrásból és -kódból is állítható elő. Nem kötelező azonban, hogy egy *Fragment* felhasználói felülettel rendelkezzen, készíthetünk úgynevezett háttér *Fragment*eket is, amelyek szintén az Activityhez vannak rendelve, és nem felhasználói felülethez kapcsolódó feladatokat látnak el.

A *Fragmentek* az Activitykhez hasonlóan saját életciklussal rendelkeznek, létrehozásukkor ugyanúgy egy inicializálásra használható *Bundle* objektumot kapnak, és elmenthetik, valamint betölthetik az állapotukat. Egy Activityhez tehát több *Fragment* is tartozhat, és ahogy a *Fragment*-váltás megtörténik, az előző *Fragment* egy Activityhez tartozó *Back Stack*-re kerülhet (de nem kötelező). A *Fragmentek* kezeléséért egy *FragmentManager* felelős, és minden *Fragment* ismeri, hogy melyik Activityhez tartozik.

Látható tehát, hogy a *Fragmentek* dinamikusan kezelhetők és érhetők el az Activityken belül, ezt egy egyszerű azonosítási módszer biztosítja. Minden *Fragment* egyedi *ID*-vel és *Tag*-gel rendelkezik, amelyek segítségével a *Fragmentek* könnyen visszakereshetők.

12.1.2. *Fragment*-életciklusmodell

Mielőtt egy konkrét alkalmazásban ismertetnénk a *Fragmentek* programozását, vizsgáljuk meg az életciklusmodelljüket. A *Fragment*-életciklusmodell hasonlít az Activity életciklusmodelljéhez, ám annál összetettebb, mivel gyakrabban történnek *Fragment*eket érintő események, és például több esemény is hathat rá, miközben az Activityre csak egy esemény hat.



12.2. ábra. Fragment-életciklusmodell

Nézzük át a *Fragment*ekhez kapcsolódó életciklus callback függvényeit, amelyek meghívását a rendszer biztosítja a megfelelő események bekövetkezésekor:

- *onInflate(Activity activity, AttributeSet attrs, Bundle savedInstanceState)*: Az ábrán ez a függvény nem látható, mivel közvetlenül a *Fragment* létrehozása után hívódik meg még az *onAttach()* függvény meghívódása előtt. Ebben a függvényben elérhetők a *Fragment*hez tartozó attribútumok, amelyeket például az XML-erőforrásként leírt `<fragment>`-en belül definiáltunk, valamint a *Bundle* objektumot is megkaptuk, amely esetleg a korábban, az *onSaveInstanceState()*-ben elmentett értékeket tartalmazza. A függvényben tehát a későbbiekben felhasználandó értékeket kell elmentenünk, amelyekre majd a futás során szükségünk lesz.
- *onAttach(Activity activity)*: A függvény azután hívódik meg, miután a *Fragment*et egy *Activity*hez hozzárendeltük. A függvény paraméterül megkapja az *Activity* referenciáját, amellyel például el tudjuk dönteni, hogy melyik *Activity*hez is csatolódt a *Fragment*, vagy használhatjuk a referenciát olyan művelethez is, amely *Context*et igényel.

A későbbiekben a tulajdonos *Activity* a *getActivity()* függvényen keresztül is elérhető. A *Fragment* teljes életciklusa alatt az inicializálásra használt *Bundle* argumentum a *getArguments()* függvénnyel érhető el, ám az inicializáló *setArguments()* függvényt csak addig hívhatjuk meg, amíg a *Fragment*et még nem csatoltuk *Activity*hez.

- *onCreate(Bundle savedInstanceState)*: Jelentős eltérést mutat az *Activity* *onCreate()* függvényéhez képest, mivel itt még nem lehetünk biztosak abban, hogy az *Activity* *onCreate()* függvénye befejeződött-e, így az *Activity* *view* hierarchiája sem áll rendelkezésre. Ebből következik, hogy semmilyen UI-jellegű funkciót nem hívhatunk ebben a függvényben. Itt tipikusan a háttérszálakat szokás elindítani, amelyek esetleg majd a *Fragment* megjelenítéséhez szükséges adatokat lekérlik. Az életciklusfüggvények a UI-szálon futnak, így az erőforrás-igényes műveleteket mindenképpen a háttérszálakon kell elhelyezni. A háttérszál által előállított adatok eléréséhez használhatunk például *handlers*eket, vagy a *Loader* osztályt is.
- *onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)*: A callback függvény feladata az, hogy egy *view*-val térjen vissza, amely a *Fragment* felületét írja le. Paraméterül megkapunk egy *LayoutInflater* objektumot, az ősnézetet, valamint az elmentett állapotot. A *LayoutInflater* segítségével egy XML-ben leírt erőforrás-felületet rendelhetünk a *Fragment*hez. Például:

```

@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    // Ha nincs ős felület, akkor felesleges a UI-t definiálni
    if(container == null) {
        return null;
    }

    View v = inflater.inflate(R.layout.testfragment,
        container, false);
    TextView textHello =
        (TextView) v.findViewById(R.id.textHello);
    textHello.setText(getResources().getString(R.string.
        hello));
    return v;
}

```

- *onActivityCreated(Bundle savedInstanceState)*: A függvény meghívódásakor már biztosak lehetünk abban, hogy a *Fragment* tulajdonos Activityjének *onCreate()* függvénye már befejeződött, és az abban definiált felhasználói felület, valamint a saját *Fragment*ünk felülete is elérhető már. Ez az a pont tehát, ahol a felhasználói felületet még manipulálhatjuk, mielőtt a felhasználó előtt valóban megjelenne. Szintén ebben a fázisban már biztosak lehetünk abban is, hogy a szükséges többi *Fragment* is hozzákapcsolódott az Activityhez.
- *onStart()*: A függvény meghívódásakor a felület már látszik a felhasználó előtt, ám még nem vezérelhető. A *Fragment onStart()*-ja szorosan kapcsolódik az Activity *onStart()*-jához, ezért a korábban ott elhelyezett logika áthelyezhető a megfelelő *Fragment onStart()* függvényébe.
- *onResume()*: Ez az utolsó callback függvény azelőtt, mielőtt a felhasználó vezérelhetné a felületet, és ugyancsak az Activity *onResume()* függvényéhez van szorosan csatolva.
- *onPause()*: Szorosan csatolódik az Activity *onPause()* függvényéhez, és hasonló célokat szolgál. Például, ha a *Fragment* egy médiaállományt játszik le, akkor azt itt érdemes leállítani, hiszen ha például bejövő hívás jön, akkor egy futó média nagyon zavaró lehet a felhasználók számára.
- *onSaveInstanceState(Bundle outState)*: Mielőtt a *Fragment* befejeződne, ebben a függvényben lehetőségünk van az állapot elmentésére. Az állapotot a paraméterül kapott *outState Bundle* objektumban menthetjük el, amelyet a *Fragment* egyébként majd megkap a korábban ismertetett *savedInstanceState Bundle* paraméterben, amikor újból előtérbe kerül.

Ügyeljünk arra, hogy mit mentünk el egy *Fragment* állapotáról, hiszen a nagy objektumok elmentése memóriagondokat okozhat. Ha például egy másik *Fragment* szeretnénk elmenteni, akkor ne azt próbáljuk eltárolni, hanem csak az azonosító Tagját, hiszen azzal később könnyedén elérhetjük.

- *onStop()*: Ez a függvény szorosan van csatolva az *Activity onStop()* *callback* függvényéhez, és hasonló célokat szolgál. Egy leállított *Fragment* újraaktiválása esetén azonnal az *onStart()* függvénye hívódhat meg, ha még nem semmisült meg teljesen.
- *onDestroyView()*: A *Fragment* leállítását jelző függvény azután hívódik meg, miután az *onCreateView()*-ban, a *Fragment*hez csatolt felület már lecsatolódtott.
- *onDestroy()*: Azután hívódik meg, miután a *Fragment* már nem használható, ám még az *Activity*hez van kapcsolva, és megkereshető a *FragmentManager*rel, de már nem használható.
- *onDetach()*: Az utolsó *callback* függvény a *Fragment* életciklusában. Akkor hívódik meg, amikor már nincs csatlakoztatva az *Activity*hez, és nincs felhasználói felülete. Ebben a függvényben már minden lefoglalt erőforrást fel kell szabadítani.

A *Fragmentek* életciklusához kapcsolódik még a *setRetainInstance(boolean retain)* függvény is, amelyben jelezhetjük, hogy az *Activity* újbóli létrehozásakor a *Fragment* ne jöjjön újra létre, hanem ugyanakkor a példánynak a felhasználása történjen meg. A *retain true* értéke esetén tehát az *onDestroy()* és az *onCreate()* függvényeket a rendszer átugorja a *Fragment* életciklusában, és ugyanazt a példányt használja. Természetesen az *onDetach()* és az *onAttach()* meghívódnak, hiszen az *Activity*-példány már más. Ekkor tehát az *onCreate()* és az *onDestroy()* függvényekben nem érdemes *Fragment*-logikát helyezni, hiszen nem biztos, hogy mindig meghívódik.

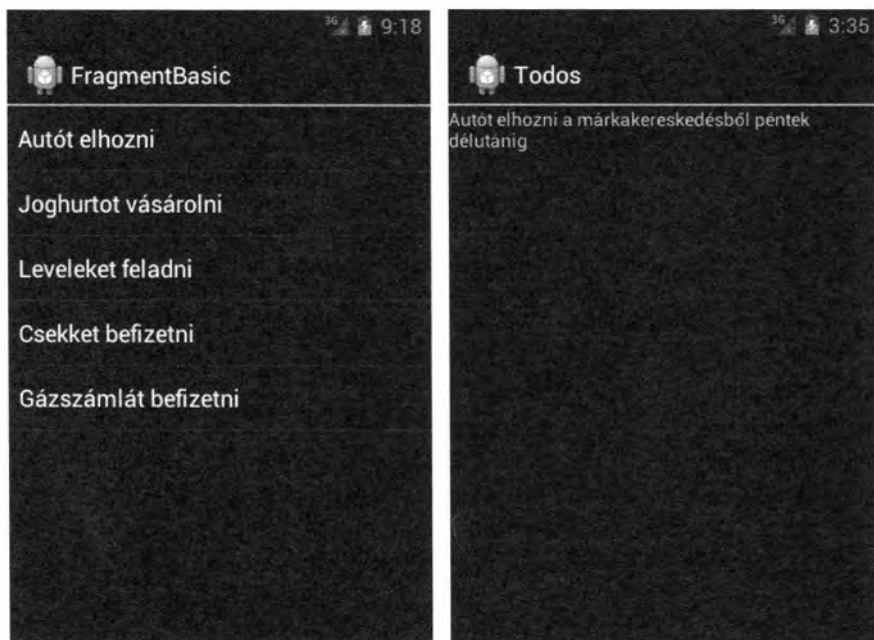
12.1.3. *Fragmentek* a gyakorlatban

A *Fragmentek* alaposabb megismeréséhez nézzünk meg egy példát, amelyben a bemutatott funkciókat működés közben ismertetjük. Példánkban egy tennivalólistát jelenítünk meg, ahol egy tennivalóra kattintva annak részletei jelennek meg. A megjelenítéshez azonban használjunk *Fragmentek*et, és biztosítsuk, hogy fekvő nézetben a tennivalók listája mellett jobb oldalt a tennivaló részletei jelenjenek meg. Az elkészítendő alkalmazás fekvő nézetét a következő ábra szemlélteti.



12.3. ábra. Tennivalók alkalmazásának fekvő nézete

Álló nézetben a kezdő nézetben csak a tennivalók listája látható, majd egy listaelemet kiválasztva a tennivaló részletei jelennek meg egy külön Activity-n, de a megjelenítéshez ugyanazokat a *Fragment*eket használjuk fel, mint a fekvő nézet esetében.



12.4. ábra. Tennivalók alkalmazásának álló nézete

A megvalósításhoz tehát két *Fragment*et használunk, egyet a tennivalók listájához, amely egy *ListFragment* viselkedését mutatja be, egyet pedig a tennivaló részleteinek a megjelenítéséhez. Látható lesz, hogy a *ListFragment* működése hasonló a *ListActivity*hez, továbbá hogy ha álló nézetben kattintunk egy tennivalóra, akkor egy új Activityben jelenik meg a tennivaló részleteit tartalmazó *Fragment*, míg fekvő nézetben egyszerűen csak a képernyő jobb oldalán egy *Fragment*-váltás hajtódik végre animáció kíséretében. Példánkban a tennivalókat és a tennivalók részleteit egyszerűen elérhető tömbök formájában tároljuk.

```
public class Todos {
    public static String TODOTITLES[] = {
        "Autót elhozni",
        "Joghurtot vásárolni",
        "Leveleket feladni",
        "Csekket befizetni",
        "Gázszámlát befizetni"
    };
    public static String TODODETAILS[] = {
        "Autót elhozni a márkakereskedésből péntek délutá-
nig",
        "Epres joghurtot kell venni",
        "A levelek a bal felső fiókban vannak",
        "Mindhárom csekket be kell fizetni",
        "Vasárnap estig be kell fizetni"
    };
}
```

A megoldás bemutatását kezdjük a kezdő felhasználói felület ismertetésével. Álló nézet esetén a *res/layout* könyvtárban az alábbi *main.xml*-t helyezzük el:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class=
        "hu.bute.daa1.amorg.examples.TodosFragment"
        android:id="@+id/titles"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

A fekvő nézetért a *res/layout-land* könyvtárban a következő *main.xml* a felelős:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#555555">
    <fragment class=
        "hu.bute.daai.amorg.examples.TodosFragment"
        android:id="@+id/todos"
        android:layout_weight="1"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:background="#00550033" />
    <FrameLayout
        android:id="@+id/tododetail"
        android:layout_weight="2"
        android:layout_width="0px"
        android:layout_height="match_parent" />
</LinearLayout>
```

Mindkét esetben a felület tartalmaz egy *<fragment>* elemet, amely a *TodosFragment*re hivatkozik. A *TodosFragment* egy *ListFragment*-leszármazott, feladata a tennivalók listájának megjelenítése és a listaelem kiválasztásának jelzése. Továbbá a fekvő nézet tartalmaz egy *FrameLayout*ot, amely a *DetailsFragment*nek biztosít helyet.

Mielőtt a *TodosFragment*et bemutatnánk, nézzük át a fő Activity forrását.

```
public class FragmentBasicActivity extends Activity {
    public static final String TAG = "Todo";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public boolean isLandscape() {
        return getResources().getConfiguration().
orientation ==
        Configuration.ORIENTATION_LANDSCAPE;
    }

    // Todo részletek megjelenítése
```



```

public void showTodo(int index) {
    // Fekvő mód esetén
    if (isLandscape()) {
        // FrameLayout-ban lévő Fragment keresése,
        // az eredmény null értékű, ha még nincs
        Fragment
        DetailsFragment details =
            (DetailsFragment)getFragmentManager().
                findFragmentById(R.id.tododetail);
        if (details == null ||
            details.getShownIndex() != index) {
            // Új fragment létrehozása
            details = DetailsFragment.newInstance(index);
            // Új fragment megjelenítése animáltan
            FragmentTransaction ft =
                getFragmentManager().beginTransaction();
            ft.setCustomAnimations(R.animator.todo_in,
                R.animator.todo_out);
            ft.replace(R.id.tododetail, details);
            ft.addToBackStack(TAG);
            ft.commit();
        }
    } else {
        // Egyéb esetben új Activity indítása
        // a tennivaló részletek megjelenítéséhez
        Intent intent = new Intent();
        intent.setClass(this, DetailsActivity.class);
        intent.putExtra("index", index);
        startActivity(intent);
    }
}
}

```

Az Activity forrásában az *isLandscape()* függvénnyel megállapítjuk, hogy álló vagy fekvő nézetben van-e aktuálisan a kijelző, míg a *showTodo(int index)* függvény feladata a paraméterül kapott indexű tennivaló megjelenítése. Fekvő nézet esetén a tennivaló részletei egy *Fragment*-ben jelennek meg, álló nézet esetén viszont egy új *Activity*-ben.

Nézzük át a fekvő nézet esetén futtatandó kódot. Elsőként a *FragmentManager* objektum segítségével felderítjük (a *getFragmentManager()* függvénnyel elérhető az *Activity*-ből), hogy a felhasználói felületen elhelyezett *tododetail* azonosítójú *FrameLayout*-ban melyik *Fragment* található. Ha ez az érték üres, vagy a nem megfelelő sorszámú tennivalóelemet jeleníti meg, akkor egy új *Fragment*-et hozunk létre. Az új *Fragment* felhelyezéséhez vagy az előző *Fragment* lecseréléséhez egy *FragmentTransaction* objektumot kell definiálnunk, amelyet a *FragmentManager beginTransaction()* függvényével

kérhetünk el használatra. Ezt követően beállítjuk a *Fragment* távozási és az új *Fragment* érkezési animációját a *setCustomAnimations()* függvénnyel. Végül azt definiáljuk, hogy a *Fragment* a *tododetail* azonosítójú *FrameLayout*-ba kerüljön be, vagy cserélje az abban lévő *Fragment*-et, majd helyezzük a váltáseseeményt a *Back Stack*-re, és a végén véglegesítjük a tranzakciót (*commit()*). A *Back Stack*-re való helyezés eredményeképpen, ha a *Vissza* gombot megnyomjuk, a *Fragment*-csere és a hozzátartozó animáció automatikusan visszafelé zajlana le.

A *FragmentManager* feladata tehát, hogy az *Activity*-hez tartozó *Fragment*-eket kezelje és felügyelje. Egyrészt a *FragmentManager* segítségével tudunk *FragmentTransaction* objektumot létrehozni, másrészt pedig az *Activity*-hez tartozó *Fragment*-eket kereshetjük meg a *findFragmentById()*, a *findFragmentByTag()* és a *getFragment()* függvényekkel.

Az előző forráskódban több állományra is hivatkoztunk, amelyeket még nem ismertettünk, ezért nézzük át ezeket. Elsőként vizsgáljuk meg a *DetailsActivity* kódját, amely tulajdonképpen csak a *DetailsFragment*-et jeleníti meg külön *Activity*-ben. A megoldás tehát nem új nézetet definiál, hanem ugyanazt a *Fragment*-et használja, amelyet majd fekvő nézetben a tennivalók listája mellé is elhelyeznénk.

```
public class DetailsActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Ha közben fekvő módra váltottunk volna
        if (getResources().getConfiguration().orientation
            == Configuration.ORIENTATION_LANDSCAPE) {
            finish();
            return;
        }
        if (getIntent() != null) {
            DetailsFragment details =
                DetailsFragment.newInstance(getIntent().
                    getExtras());
            getFragmentManager().beginTransaction()
                .add(android.R.id.content, details)
                .commit();
        }
    }
}
```

A *DetailsFragment* forrása a következő:

```
public class DetailsFragment extends Fragment {
    private int mIndex = 0;

    public static DetailsFragment newInstance(int index)
    {
        DetailsFragment df = new DetailsFragment();
        // index argumentum elmentése,
        // itt még hívhatunk setArguments()-et
        Bundle args = new Bundle();
        args.putInt("index", index);
        df.setArguments(args);
        return df;
    }

    public static DetailsFragment newInstance(Bundle
bundle) {
        int index = bundle.getInt("index", 0);
        return newInstance(index);
    }

    @Override
    public void onCreate(Bundle myBundle) {
        super.onCreate(myBundle);
        mIndex = getArguments().getInt("index", 0);
    }

    public int getShownIndex() {
        return mIndex;
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        if(container == null) {
            return null;
        }
        View v = inflater.inflate(R.layout.details,
            container, false);
        TextView textTodo =
            (TextView) v.findViewById(R.id.textTodo);
        textTodo.setText(Todos.TODODETAILS[ mIndex ] );
        return v;
    }
}
```

A *DetailsFragment*ben a *newInstance()* függvény szerepe az, hogy létrehozzon egy új *Fragment* példányt, és beállítsa a tennivaló sorszámparaméterét, amelynek alapján megállapíthatjuk, hogy melyik tennivaló részleteit kell megjeleníteni. Az *onCreate()* függvényben a tennivaló sorszámát elmentjük az *mIndex* tagváltozóba, és végül az *onCreateView()*-ban a *LayoutInflater* használatával összeállítjuk a *Fragment* felületét, és beállítjuk a tennivalók részleteit. A *DetailsFragment* felületéért felelős egy XML erőforrás (*layout/details.xml*).

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView android:id="@+id/textTodo"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</ScrollView>
```

A tennivalók listáját szolgáltató *TodosFragment* *ListFragment* megvalósítása a következő:

```
public class TodosFragment extends ListFragment {
    private FragmentBasicActivity myActivity = null;
    int mSelectedTodoPosition = 0;

    @Override
    public void onAttach(Activity myActivity) {
        super.onAttach(myActivity);
        this.myActivity = (FragmentBasicActivity)
myActivity;
    }

    @Override
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        if (bundle != null) {
            // Kiválasztott index
            mSelectedTodoPosition = bundle.
getInt("curIndex", 0);
        }
    }

    @Override
    public void onActivityCreated(Bundle bundle) {
        super.onActivityCreated(bundle);
```

```

// Lista feltöltése
setListAdapter(new ArrayAdapter<String>(getActivit
y(),
    android.R.layout.simple_list_item_1,
    Todos.TODOITITLES));

ListView lv = getListView();
lv.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
lv.setSelection(mSelectedTodoPosition);

myActivity.showTodo(mSelectedTodoPosition);
}

@Override
public void onSaveInstanceState(Bundle bundle) {
    super.onSaveInstanceState(bundle);
    bundle.putInt("curIndex", mSelectedTodoPosition);
}

@Override
public void onItemClick(ListView l, View v,
    int pos, long id) {
    myActivity.showTodo(pos);
    mSelectedTodoPosition = pos;
}

@Override
public void onDetach() {
    super.onDetach();
    myActivity = null;
}
}

```

A *TodosFragment* forráskódját áttekintve láthatjuk, hogy meglehetősen hasonlít egy általános *ListActivity* megvalósításhoz. Az *onActivityCreated()* függvényben beállítjuk a *ListAdapter*-t, amely a *Todos.TODOITITLES* tömbből inicializálja a listát, majd a *ListFragment*hez tartozó *ListView*-nak beállítjuk a megjelenítési módját, végül pedig megjelenítjük az első listaelemet a korábban ismertetett *showTodo()* függvénnyel.

Végezetül már csak az *R.animator.todo_in* és az *R.animator.todo_out* animációk bemutatásával tartozunk. A *todo_in* animáció az alábbi:

```

<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android=
    "http://schemas.android.com/apk/res/android">

```

```

<objectAnimator xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:interpolator="@android:interpolator/
bounce"
    android:valueFrom="-1280"
    android:valueTo="0"
    android:valueType="floatType"
    android:propertyName="X"
    android:duration="2000" />
<objectAnimator xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:interpolator="@android:interpolator/
bounce"
    android:valueFrom="-1280"
    android:valueTo="0"
    android:valueType="floatType"
    android:propertyName="Y"
    android:duration="2000" />
</set>

```

A *todo_out* forrása pedig a következő:

```

<?xml version="1.0" encoding="utf-8" ?>
<objectAnimator xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:interpolator=
        "@android:interpolator/accelerate_cubic"
    android:valueFrom="0"
    android:valueTo="1280"
    android:valueType="floatType"
    android:propertyName="X"
    android:duration="2000"
/>

```

Az animációk megvalósításához az *ObjectAnimator* megoldást használtuk, amely ugyancsak az újabb Android-verzióban jelent meg. Az *ObjectAnimator* hasonlóan működik a korábban bemutatott animációkhoz, ám itt tetszőleges objektumattribútumot változtathatunk, csupán azt kell biztosítanunk, hogy az attribútumnak legyen megfelelő *setter* metódusa. A változtatandó attribútum nevét az *android:propertyName* értékeként kell megadni.

Az alkalmazás kipróbálásakor azt tapasztalhatjuk, hogy álló nézetben nem a lista jelenik meg, hanem azonnal a tennivalórészletek nézete. Ennek kijavítását az olvasóra bízunk, de segítségként eláruljuk, hogy a jelenség okozója az, hogy a *TodosFragment.onActivityCreated()* függvényében feltétel nélkül meghívtuk a *showTodo()* metódust.

12.2. Fejlett felületi elemek: *ActionBar*, *ViewPager*, *ViewPagerIndicator*

A következőkben az új Android-verziókban gyakran használt *ActionBar*, *ViewPager* és *ViewPagerIndicator* komponenseket ismertetjük egy-egy példával.

12.2.1. Az *ActionBar* bemutatása

Az *ActionBar* tulajdonképpen egy dedikált felület az alkalmazáson belül, amelyen az alkalmazás logója, a főbb parancsok és a globális navigációs eszközök láthatók. Az *ActionBar* célja az, hogy a gyakran használt funkciókat egyszerűen elérhetővé tegye a felhasználók számára, és ne kelljen őket eldugott menükben, illetve különféle nézeteken keresgélteni.

Az *ActionBar* az Android 3.0-s verziójától kezdve érhető el, de egy ingyenesen elérhető osztálykönyvtár (*ActionBarSherlock*²⁶) segítségével a korábbi verziókon is használható. A komponens fő alkalmazási területei a következők:

- alkalmazás logó/ikon megjelenítése,
- konzisztens navigációs vezérlők,
- az adott Activity fő funkcióinak könnyű elérhetővé tétele (pl. keresés, létrehozás, megosztás stb.).



12.5. ábra. ActionBar az Android HoneyComb galériaalkalmazásából²⁷

Az *ActionBar* alapelemként jelentkezik azokban az alkalmazásokban, amelyek a *Theme.Holo* témát használják, amely egyébként az alapértelmezett téma a 3.0-s verziótól felfelé. Tehát alapesetben az *ActionBar* elérhető, ha a *manifest* állományban a *targetSdkVersion* vagy a *minSdkVersion* értékek 11-esre vagy nagyobbra vannak állítva:

```
<manifest ... >
  <uses-sdk android:minSdkVersion="8"
    android:targetSdkVersion="11" />
  ...
</manifest>
```

²⁶ ActionBarSherlock: <http://actionbarsherlock.com/>

²⁷ Forrás: <http://developer.android.com/guide/topics/ui/actionbar.html>

Ha az *ActionBar* API-ját el szeretnénk érni, a *minSdkVersion* értéket is 11-esre vagy nagyobbra kell állítani, ha nem az *ActionBarSherlock*ot használjuk. Viszont ha egy Activity-nem szeretnénk az *ActionBar*ot feltüntetni, akkor ezt a *Theme.Holo.NoActionBar* téma megadásával érhetjük el.

```
<activity android:theme=
    "@android:style/Theme.Holo.NoActionBar">
```

Forráskódon belül az *ActionBar* egy Activity-n belül a *getActionBar()* függvénnyel érhető el, letiltani pedig a *hide()* függvénnyel lehetséges, amikor is a rendszer átméretezi az aktuális *Layout*ot. Újbóli megjelenítésre a *show()* függvény használható.

```
ActionBar actionBar = getActionBar();
actionBar.hide();
```

Ha egy Activity-ben az *ActionBar* láthatóságát gyakran állítjuk, akkor érdemes az Activity számára egy külön témát készíteni, amelyben az *android.windowActionBarOverlay* paraméter *true* értékével megadható, hogy az *ActionBar* ne a *Layout* területéhez tartozzon, hanem fölötte helyezkedjen el átlapolódva.

Az *ActionBar* megjelenítése sokféleképpen testre szabható, példaként nézzünk meg egy tipikus elrendezést, amelyben az *ActionBar* négy fő részre oszlik. Az egyes részeket a következő ábra szemlélteti, de ettől eltérő elrendezés is kialakítható.



12.6. ábra. Az *ActionBar* felépítése²⁸

Az egyes részek a következők:

1. Alkalmazásikon: Az alkalmazás ikonja tetszőleges logóra is lecserélhető. Kattintásra legtöbbször egy kezdőnézetet szokás megjeleníteni. Az ikon melletti balra nyilat akkor érdemes engedélyezni a felhasználók számára, ha nem a kezdőnézeten vagyunk.

²⁸ Forrás: <http://developer.android.com/design/patterns/actionbar.html>

2. Nézetvezérlő: Ha az alkalmazás több nézetből áll, akkor itt érdemes megvalósítani a nézetek közti váltást. Ha nem, akkor statikus információ, például az alkalmazás címe jeleníthető meg itt.
3. Akciógombok: Az alkalmazás legfontosabb funkcióit érdemes itt elhelyezni. Ha nem fér ki valamelyik funkció a képernyő korlátos mérete miatt, akkor automatikusan átkerül az *ActionBar* menübe.
4. *ActionBar* menü: Az alkalmazás további funkcióit érdemes itt elhelyezni. Egyre inkább felváltja az Android korábban megszokott menüjét. Azokon a készülékeken automatikusan megjelenik, amelyeknek nincs dedikált menügombjuk.

12.2.1.1. *ActionBar* alkalmazásikonjának kezelése

Alapértelmezetten az alkalmazás ikonja az *ActionBar* bal felső részén jelenik meg, az ikonnak engedélyezhetjük az érintéseményét. Az *ActionBar* megjelenő logó a *manifest* állományban az *android:logo* attribútumon keresztül tetszőlegesen beállítható. Miután beállítottuk az értéket, az *ActionBar* objektumnak engedélyezni kell, hogy egyedi logót jelenítsen meg a *setDisplayUseLogoEnabled(true)* függvénnyel. Az alkalmazásikonra való kattintás eseményét tipikusan két dologra szokták felhasználni: kezdő Activityre ugrás vagy egy szinttel feljebb ugrás a navigációs hierarchiában.

Az *ActionBar* az alkalmazás ikonra való kattintás eseményét a *setHomeButtonEnabled()* függvénnyel engedélyeznünk kell az Android 4.0-tól felfelé.

```
ActionBar actionBar = getActionBar();
actionBar.setHomeButtonEnabled(true)
```

Amikor az alkalmazásikonra kattintottunk, az *Activity.onOptionsItemSelected()* függvénye hívódik meg az *android.R.id.home* azonosítóval, ekkor megvalósíthatjuk azt, hogy a kezdő Activity elinduljon, vagy átválthatunk például az előző szinten lévő Activityre. Ha a kezdő Activityre váltunk, az eseményt indító Intentben érdemes beállítani a *FLAG_ACTIVITY_CLEAR_TOP* jelzőt, hogy az eddig a *Back Stack*en lévő Activityk befejeződjenek, és valóban a kezdő Activityre kerüljünk, mintha most indítottuk volna el az alkalmazást.

Az *onOptionsItemSelected()* példa megvalósítása a következő:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            Intent intent = new Intent(this, HomeActivity.
            class);
```

```

        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
        startActivity(intent);
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}

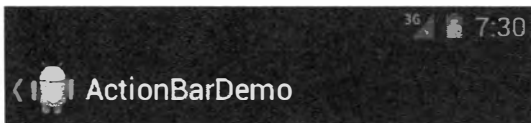
```

Ha az *ActionBar* lévő alkalmazásikon nem a kezdő Activityre navigál vissza, hanemegyszinttelfeljebb, lehetőségünk van ezt egy egyszerű balra nyíllal támogatni. Ennek megvalósításához az *ActionBar* *setDisplayHomeAsUpEnabled()* függvényét kell használnunk.

```

ActionBar actionBar = getActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);

```



12.7. ábra. *ActionBar* alkalmazásikonon visszalépési jelölővel

12.2.1.2. Egyszerű menüelemek elhelyezése

Az *ActionBar* lehetővé teszi, hogy a gyakran használt menüelemeket megjelenítse, és könnyen elérhetővé tegye a felhasználók számára. Ezeket az elemeket úgynevezett *ActionItem*eknek szokás nevezni, de valójában ugyanolyan *MenuItem*ekről van szó, mint amelyeket a hagyományos menüknél bemutatunk. Ha többet helyezünk el az *ActionBar*on, akkor amelyik nem fér ki, az a menüből továbbra is elérhető lesz. Illetve ha az adott eszközön nincs külön menügomb, akkor ezeket az elemeket az *ActionBar* jobb oldalán lévő felugró menüből érhetjük el.

Az *ActionItem*eket tulajdonképpen ugyanúgy kell felvennünk, mintha általános menük lennének, az *onCreateOptionsMenu()* függvényben a megszokott módon definiáljuk a menüt például a *MenuInflater* segítségével.

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    return true;
}

```

A különbség abban nyilvánul meg, hogy ha szeretnénk, hogy egy menüelem az *ActionBar* jelenjen meg *ActionItem*ként, akkor azt a menüelem `<item>` XML-leírásában az `android:showAsAction="ifRoom"` attribútummal kell jelölni. Az *ifRoom* helyett *always* is használható, ha kényszeríteni szeretnénk a menüelem megjelenítését. Fontos, hogy az adott menüelem `android:title` és `android:icon` attribútuma is ki legyen töltve. Ha szeretnénk, hogy az *ActionItem* szövege látható legyen, akkor ezt az `android:showAsAction` attribútum *withText* értékének hozzáadásával tehetjük meg.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android=
"http://schemas.android.com/apk/res/android">
  <item android:id="@+id/save_contact"
    android:icon="@drawable/ic_menu_save"
    android:title="@string/menu_save_contact"
    android:showAsAction="ifRoom|withText" />
</menu>
```

Ezt követően az *ActionItem* kezelése ugyanúgy történik, mint a menüelemeké. Egy *ActionItem* kiválasztásakor az `onOptionsItemSelected()` függvény hívódik meg, és paraméterében átkerül a kiválasztott *ActionItem* `android:id` attribútuma.

Az *ActionItem*ek kiválasztását érdemes alaposan átgondolni, ide valóban a fontos és gyakran használt elemeket kell elhelyezni, ilyen például a keresés, a frissítés, az új elem felvitele stb.



12.8. ábra. ActionItemek elhelyezése az ActionBaron

Android 4.0-tól felfelé támogatva van az úgynevezett osztott *ActionBar* is, amely az *ActionItem*eket külön sorban helyezi el a szűk képernyőn. Az osztott *ActionBar* engedélyezéséhez vagy az `<activity>`, vagy az `<application>` elemen belül be kell állítanunk az `uiOptions="splitActionBarWhenNarrow"` értéket.

12.2.1.3. Egyedi *ActionItem* nézet definiálása

Gyakran szükség lehet arra, hogy egy egyszerű nézetet, például egy keresőmezőt helyezünk el az *ActionBar*on. Ez tipikusan úgy szokott működni, hogy egy létező *ActionItem*re kattintva az *ActionBar*on egy előre definiált új nézet jelenik meg, ahova például beírhatjuk a keresendő kifejezést. Keresőmező esetén például használhatjuk az Android beépített `android.widget.SearchView` osztályát.



12.9. ábra. ActionBar SearchView-val

Ahhoz, hogy egy ilyen egyedi nézet egy *ActionItem*-re kattintva jelenjen meg, az *ActionItem*-et leíró menüelem definiálásakor az *android:showAsAction* attribútumnak a *collapseActionView* értéket is be kell állítanunk, és meg kell adnunk az *android:actionViewClass* attribútumban, hogy melyik osztály felelős az egyedi nézet megjelenítéséért.

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menuSearch"
        android:icon="@android:drawable/ic_menu_search"
        android:title="@string/menu_search"
        android:showAsAction="ifRoom|collapseActionView"
        android:actionViewClass="android.widget.
        SearchView" />
</menu>
```

Ha az egyedi nézetet manipulálni szeretnénk a forráskód oldaláról, akkor ezt megtehetjük például az *onCreateOptionsMenu()*-ben.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mainmenu, menu);
    SearchView searchView =
        (SearchView) menu.findItem(
            R.id.menuSearch).getActionView();
    return true;
}
```

Az egyedi nézet (példánkban a keresési nézet) akkor jelenik meg az *ActionBar*-on, ha a *menuSearch ActionItem*-et kiválasztotta a felhasználó. Ha valamilyen más esemény hatására szeretnénk megjeleníteni vagy eltüntetni valamelyik *ActionItem*-hez tartozó egyedi nézetet, akkor ezt az adott *MenuItem* objektum *expandActionView()/collapseActionView()* függvényeivel tehetjük meg.

Továbbá lehetőségünk van az adott *ActionItem* kinyitás/bezárás eseményére is feliratkozni.

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mainmenu, menu);
    MenuItem menuSave = menu.findItem(R.id.menuSearch);
    menuSave.setOnActionExpandListener(
        new OnActionExpandListener() {
            @Override
            public boolean onMenuItemActionCollapse(MenuItem
item) {
                // true esetén ActionView bezárása
                return true;
            }
            @Override
            public boolean onMenuItemActionExpand(MenuItem
item) {
                // true esetén ActionView kinyitása
                return true;
            }
        });
    return true;
}

```

12.2.1.4. Menüelemek kiterjesztése

Sokszor előfordul, hogy egy *ActionItem* összetett funkciót lát el, amelyhez egyedi felületet kell megjeleníteni a felhasználó számára, ahol további adatokat tud megadni. Gyakran szokás például egy „Megosztás” elemet elhelyezni az *ActionBar*-on, amely lehetővé teszi, hogy a listában kiválasztott elemet megosszuk e-mailben, Bluetoothon stb. Ilyenkor az *ActionItem*ekhez különféle *ActionProvider*eket definiálhatunk.

Ha valóban adatmegosztást szeretnénk megvalósítani, használhatjuk az Android beépített *android.widget.ShareActionProvider*²⁹ osztályát.

```

<menu xmlns:android=
"http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menuShare"
        android:icon="@android:drawable/ic_menu_share"
        android:title="@string/menu_share"
        android:showAsAction="ifRoom|withText"
        android:actionProviderClass=
            "android.widget.ShareActionProvider" />
</menu>

```

²⁹ A *ShareActionProvider* használata: <http://developer.android.com/guide/topics/ui/actionbar.html#ActionProvider>

A *ShareActionProvider* használata azonban csak egy lehetőség. Ha további lehetőségekre van szükség, készíthetünk saját megoldást is, amikor az *ActionProvider* osztályból kell leszármaztatnunk, és felül kell definiálnunk a megfelelő függvényeket.

További lehetőségként, hogyha például füleket szeretnénk használni az alkalmazásunkban, akkor egyrészt használhatjuk a hagyományos *TabWidget* megoldást, másrészt az *ActionBar*hoz is lehet hasonló navigációs füleket rendelni,³⁰ amellyel akár *Fragment*ek közti váltást is megvalósíthatunk. Az *ActionBar*on való fülek alkalmazásának további előnye az, hogy az *ActionBar* a fülek méretét rugalmasan méretezi, és jobban a kijelzőhöz igazítja.

Az *ActionBar*on való navigációs fülek használatához elsőként egy saját osztályban meg kell valósítani az *ActionBar.TabListener* interfészt, ahol meg kell adnunk, hogy mi történjen a fülek váltásakor az *onTabSelected()* függvény felüldefiníálásával. Ezt követően az *ActionBar* navigációs módjának meg kell adnunk az *ActionBar.NAVIGATION_MODE_TABS* értéket, majd az *addTab()* függvényekkel felvehetjük a megfelelő füleket az *ActionBar*ra.

12.2.2. A *ViewPager* és a *ViewPagerIndicator* komponensek bemutatása

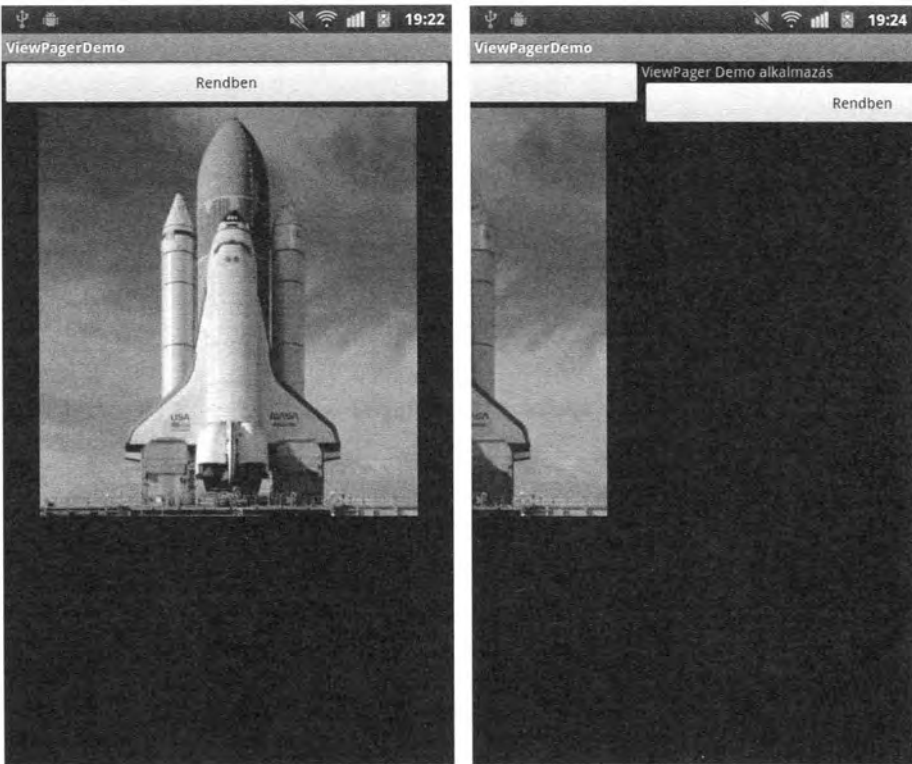
A mobilalkalmazások fejlődésével egyre fontosabbá válnak a látványos animációkkal kísért, egyszerűen használható felhasználói felületi elemek. Az érintőképernyők dominanciájának eredményeképpen a felhasználóknak olyan felületre van legtöbbször igényük, amelyeket egyszerű készmozdulatokkal, könnyedén és gyorsan irányíthatnak. Az egyik ilyen legnépszerűbb megoldás az, amikor a kezünket a képernyőn oldalra húzva válthatunk a különféle nézetek között egy folyamatos átúszási animáció kíséretében. Android platformon ezt a funkcionalitást hivatott megvalósítani a *ViewPager* komponens.

A *ViewPager* jelenleg a korábban bemutatott *Compatibility Pack*kel együtt érhető el, így korábbi, 2.x alapú Android-eszközökön is egyszerűen alkalmazhatjuk. A *ViewPager* feladata az, hogy egy *Activity*n belül lehetővé tegye több nézet között az egyszerű váltást. Az egyes nézetek akár külön *Fragment*ek is lehetnek, így egy valóban rugalmasan működő felhasználói felületet hozhatunk létre.

A *ViewPager* használatához a felhasználói felületen el kell helyoznunk egy *ViewPager* elemet, majd létre kell hoznunk egy saját osztályt, amely a *PagerAdapter* osztályból származik le. Ennek az osztálynak az a feladata, hogy megadja azt, hogy az adott *ViewPager* mennyi különböző nézetet (*View*-t) támogat, illetve a megfelelő nézetre váltás esetén meghatározza az aktuális nézettartalmat, amelyeket tipikusan a *LayoutInflater* segítségével XML-erőforrásból szokás létrehozni. A *PagerAdapter* további feladata a nézetek megfelelő felszabadítása, valamint esetlegesen az állapotmentés.

A *ViewPager* használatához nézzünk meg egy egyszerű példát. Készítsünk egy olyan alkalmazást, amely három nézetet definiál, és ezek között egy *ViewPager* segítségével tudunk „lapozni”.

³⁰ *ActionBar* navigációs fülek kezelése: <http://developer.android.com/guide/topics/ui/actionbar.html#Tabs>



12.10. ábra. ViewPager lapozási példa

Az alkalmazás fő felhasználói felülete (*main.xml*) a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <android.support.v4.view.ViewPager
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/viewPagerMain"/>
</LinearLayout>
```

Az egyes nézeteket megvalósító XML-erőforrások a következők.

A *layoutone.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnOk" />
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/atlantis"
        android:contentDescription="@string/imgPlace" />
</LinearLayout>
```

A *layouttwo.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnOk" />
</LinearLayout>
```

Végül készítsünk egy harmadik nézetet is (*layoutthree.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
```



```

    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnOk" />
</LinearLayout>

```

A *ViewPager* használatához szükség van egy saját *PagerAdapter* megvalósításra, amely példánkban a következő:

```

public class DemoPagerAdapter extends PagerAdapter {

    public static final int DEMOVIEWSNUM = 3;

    public int getCount() {
        return DEMOVIEWSNUM;
    }

    public Object instantiateItem(View collection,
        int position) {
        LayoutInflater inflater =
            (LayoutInflater) collection.getContext().
                getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        int resId = 0;
        switch (position) {
            case 0:
                resId = R.layout.layoutone;
                break;
            case 1:
                resId = R.layout.layouttwo;
                break;
            case 2:
                resId = R.layout.layoutthree;
                break;
        }
        View view = inflater.inflate(resId, null);
        ((ViewPager) collection).addView(view, 0);
        return view;
    }

    @Override
    public void destroyItem(View viewPager, int arg1,
        Object currView) {

```

```

        ((ViewPager) viewPager).removeView((View)
currView);
    }

    @Override
    public boolean isViewFromObject(View arg0, Object
arg1) {
        return arg0 == ((View) arg1);
    }

    @Override
    public Parcelable saveState() {
        return null;
    }
}

```

A *getCount()* függvény egy konstansként megadott 3-as értékkel tér vissza, hiszen az alkalmazásunkban 3 nézetet támogatunk a *ViewPager*-en keresztül. Emellett figyeljük még meg az *instantiateItem()* függvényt, amely a megfelelő nézeteket hozza létre a különböző indexek esetében egy *LayoutInflater* segítségével. Továbbá a *destroyItem()* függvény feladata a nézetek eltávolítása, amikor nincs rájuk szükség. Ha a *ViewPager* nézet váltási eseményére van szükségünk, egyszerűen csak be kell állítanunk egy *OnPageChangeListener* implementációt a *ViewPager*-nek a *setOnPageChangeListener()* függvényével.

Végül az Activity kódja a következő:

```

public class ViewPagerDemoActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        DemoPagerAdapter vpAdapter = new
DemoPagerAdapter();
        ViewPager mainPager =
            (ViewPager) findViewById(R.id.viewPagerMain);
        mainPager.setAdapter(vpAdapter);
        mainPager.setCurrentItem(1);
    }
}

```

A *ViewPager* megfelelő használatához csupán annyi dolgunk van, hogy megadjuk a *PagerAdapter* implementációt, és beállítsuk a kezdőnézetet.

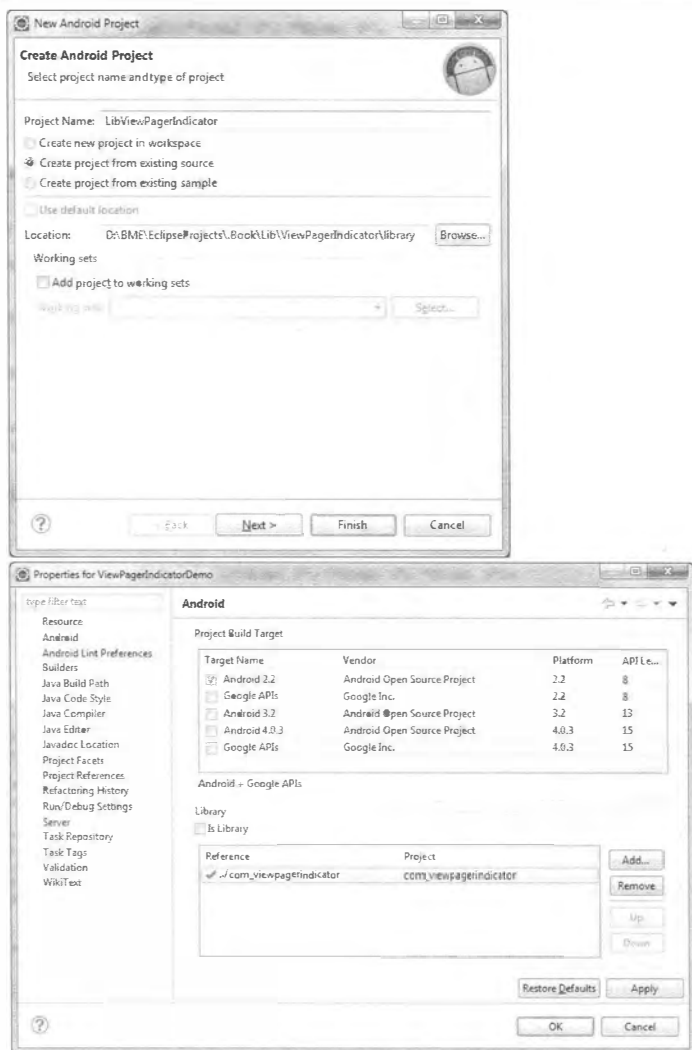
ViewPager alkalmazásakor sokszor szükség lehet arra, hogy a felhasználót valahogyan tájékoztassuk arról, hogy éppen melyik oldalon jár. Ennek megvalósításához több nyílt osztálykönyvtárat is használhatunk, amelyek közül az egyik legnépszerűbbet, a *ViewPagerIndicator*-t mutatjuk be.³¹ A *ViewPagerIndicator* három megjelenítési formát támogat (*TitlePageIndicator*, *TabPageIndicator* és *CirclePageIndicator*), amelyeket a következő ábra szemléltet. Mindhárom típus további stílusok definiálásával tetszőlegesen testre szabható.



12.11. ábra. *ViewPagerIndicator* stílusok

A *ViewPagerIndicator* használatához elsőként le kell töltenünk az osztálykönyvtárat, majd új Andorid-projektként fel kell vennünk az Eclipse *workspace*-be. Ezt követően abban a projektben, ahol használni szeretnénk, fel kell venni a létrehozott *ViewPagerIndicator* projektet Android-könyvtárprojektként. Ügyeljünk arra, hogy ez a *ViewPagerIndicator* projekt már tartalmazza a korábban említett *Compatibility Packet*, így ezt a saját projektünkbe nem kell külön felvenni.

³¹ *ViewPagerIndicator* komponens: <http://viewpagerindicator.com/>



12.12. ábra. A ViewPagerIndicator könyvtár beállítása

A *ViewPagerIndicator* bemutatásához egészítsük ki az előző projektet. A feladatunk csupán annyi, hogy a felhasználói felületen definiáljuk, hogy hol jelenjen meg a jelölősor, valamint hogy a *PagerAdapter* osztály implementálja a *TitleProvider* interfészt, és a *getTitle(int position)* függvényben visszatérjünk az adott oldal nevével. Végül pedig a *ViewPagerIndicator*-nak be kell állítani, hogy melyik *ViewPager*-hez tartozik.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <com.viewpagerindicator.TitlePageIndicator
        android:id="@+id/titles"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent" />
    <android.support.v4.view.ViewPager
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/viewPagerMain"/>
</LinearLayout>
```

A *PagerAdapter* megvalósítása a következő:

```
public class DemoPagerAdapter extends PagerAdapter
implements TitleProvider {
    private String[] TITLES = new String[]{"Oldal 1",
"Oldal 2", "Oldal 3"};

    // ...

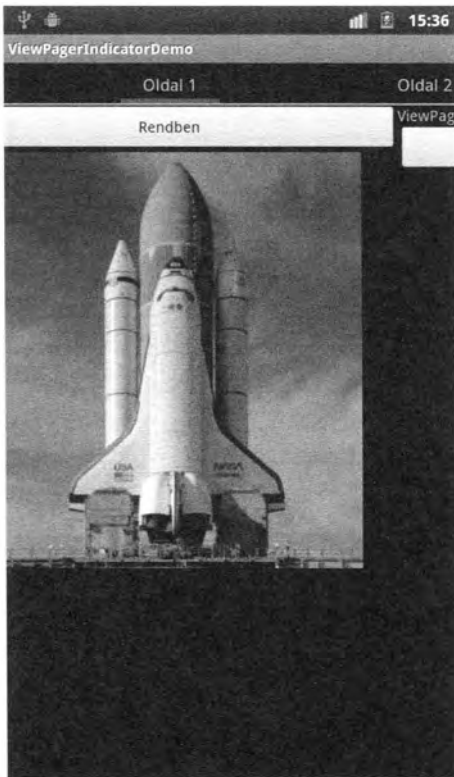
    @Override
    public String getTitle(int position) {
        return TITLES[position];
    }
}
```

Végül az Activityben a *ViewPager* a következőképpen kötjük a *ViewPagerIndicator*hoz:

```
public class ViewPagerIndicatorDemoActivity extends
Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        DemoPagerAdapter vpAdapter = new
DemoPagerAdapter();
        ViewPager mainPager =
            (ViewPager) findViewById(R.id.viewPagerMain);
```

```
mainPager.setAdapter(vpAdapter);
mainPager.setCurrentItem(0);
// ViewPager hozzákötése a ViewPager Indicatorhoz
TitlePageIndicator titleIndicator =
    (TitlePageIndicator)findViewById(R.id.titles);
titleIndicator.setViewPager(mainPager);
    }
}
```

Az elkészült *ViewPagerIndicator*-t működés közben a következő ábra szemlélteti.



12.13. ábra. A ViewPagerIndicator működése

12.3. Az Android natív programozása

Az Android egy Linux-alapú operációs rendszer, és a nyíltságának köszönhetően lehetőség van arra, hogy a fejlesztők közvetlenül natív alkalmazásokat fejlesszenek. Továbbá, mivel az Android nyílt forráskódú, ezért lehetőség van a teljes forráskód újrafordítására és kiegészítésére is, akár saját Android rendszert is készíthetünk, és azt tetszőlegesen testre szabhatjuk. Egyik ilyen népszerű kezdeményezés például a Cyanogenmod,³² de több hasonló is megtalálható és felhasználható.

12.3.1. A natív programozás jellemzői

Az Android natív programozásakor a legnagyobb különbség az eddig ismertekkel ellentétben, hogy nem Java nyelven, hanem C, illetve C++ nyelven történik az alkalmazások fejlesztése, hiszen a natív programokat már nem a Dalvik virtuális gép futtatja, hanem azok közvetlenül az Android operációs rendszeren futnak. Továbbá az Android natív alkalmazások fejlesztésekor a natív kódot nem az SDK segítségével fordítjuk le, hanem a szintén ingyenesen elérhető NDK (Native Development Kit) segítségével. A fordításhoz Linux-környezetre van szükség, illetve Windows operációs rendszer esetén egy Linux „emulációs” környezet is használható, például a Cygwin.³³

A natív alkalmazásfejlesztés legnagyobb előnye az, hogy a C-ben és a C++-ban megírt natív kódrészek, bonyolultabb algoritmusok és számítások sokkal gyorsabbak, hiszen többek között nem kell a virtuális gép overheadjével számolni. További előny az, hogy a számítástechnika világában népszerű C és C++ nyelven elkészített modulok és algoritmusok könnyedén felhasználhatók Android platformon is, amellyel rengeteg idő spórolható meg bizonyos esetekben.

Az NDK különféle eszközöket biztosít a C- és a C++-osztálykönyvtárak fordításához, és lehetővé teszi, hogy a natív osztálykönyvtárakat standard Android-alkalmazásokba beágyazzuk, és elkészítsük a telepítőcsomagot (*apk* állományt). Emellett az NDK-n keresztül Android-specifikus natív osztályheaderek és könyvtárak is elérhetők, amelyekkel például natív Activity fejlesztésére is lehetőségünk van. Továbbá az NDK telepítése után érdemes szétnézni a telepítési könyvtárban, hiszen számos dokumentációt, példakódot és tutorialt is nyújt a csomag. Az Android NDK telepítési könyvtárán belül, a *docs* könyvtárban találhatók a segédletek, amelyek közül a legfontosabbak a következők:

- *INSTALL.HTML*: NDK telepítése és beállítása,
- *OVERVIEW.HTML*: NDK képességeinek a leírása,
- *ANDROID-MK.HTML*: az *Android.mk* fájl szerkezete (megadja, hogy mely állományokat akarjuk fordítani),

³² Cyanogenmod – kernelfordítás: http://wiki.cyanogenmod.com/wiki/Building_Kernel_from_source

³³ Cygwin: <http://www.cygwin.com/>

- *APPLICATION-MK.HTML*: az *Application.mk* fájlszerkezete,
- *CPLUSPLUS-SUPPORT.HTML*: C++-támogatás leírása,
- *CPU-ARCH-ABIS.HTML*: CPU-architektúrák és az azonosítók,
- *CPU-FEATURES.HTML*: bemutatja a *cpufeatures* statikus könyvtárat, amely lehetővé teszi, hogy futási időben detektáljuk a CPU-típust és a képességeket,
- *CPU-ARM-NEON.HTML*: az ARM NEON-ra és a VFPv3-D32-re való fordítás lépései,
- *CHANGES.HTML*: változások az aktuális verzióban,
- *DEVELOPMENT.HTML*: NDK verziócsomag készítésének lépései,
- *HOWTO.HTML*: általános információk az NDK használatához,
- *IMPORT-MODULE.HTML*: modulok megosztása és újrafelhasználása,
- *LICENSES.HTML*: kapcsolódó open source license-ek,
- *NATIVE-ACTIVITY.HTML*: natív Activity bemutatása,
- *NDK-BUILD.HTML*: az ndk-build script bemutatása,
- *NDK-GDB.HTML*: natív kóddebugger leírása,
- *PREBUILTS.HTML*: statikus és osztott könyvtárak használata,
- *STANDALONE-TOOLCHAIN.HTML*: standalone compiler (béta) használatának bemutatása,
- *SYSTEM-ISSUES.HTML*: ismert bugok leírása,
- *STABLE-APIS.HTML*: API-k (headerök) bemutatása.

Mielőtt bemutatnánk a natív fejlesztés lépéseit, vizsgáljuk meg, hogy mikor van szükség natív modulok fejlesztésére. Az NDK használata nem növeli meg automatikusan a teljesítményt, hiszen nem biztos, hogy az adott algoritmus eléri azt a bonyolultságot, amely csak jelentős overhead mellett tud a virtuális gépen futni. Egyszerűbb számítási műveleteknél gyakran nincs érezhető különbség, ugyanakkor egy feleslegesen natív oldalra helyezett funkcionalitás minden esetben növeli az alkalmazás komplexitását. Csak akkor érdemes tehát natív modult fejleszteni, ha az valóban indokolt, és nem csak azért, mert például az adott algoritmust szívesebben írnánk meg C-ben vagy C++-ban.

Tipikusan a következő esetekben érdemes kihasználni az NDK nyújtotta lehetőségeket:

- nagyobb C/C++ kódok átemelése, portolás,
- önálló és kellően bonyolult feladat/algoritmus fejlesztése,
- CPU-intenzív műveletek, amelyekre nem a memóriaintenzivitás jellemző,

- jelfeldolgozási feladatok,
- grafikai algoritmusok,
- fizikai szimuláció.

Mielőtt egy adott funkcionalitást vagy algoritmust natív oldalon elkészítenénk, érdemes megnézni, hogy nem létezik-e hivatalos Android API. Ha létezik, akkor meg kell vizsgálni, hogy kellően gyors-e, hiszen jó eséllyel optimális implementáció áll az API mögött, ugyanakkor előfordulhat, hogy a valóban számításigényes algoritmusoknál gyorsabb lesz a natív megvalósításunk.

Az NDK-val fordított natív könyvtárakat Android-alkalmazásból tipikusan JNI-n (Java Native Interface) keresztül érjük el. A natív fejlesztés Android 1.5-től felfelé van támogatva, ám a natív Activity csak Android 2.3-tól érhető el a *NativeActivity* C++-osztályon keresztül, amely az életciklus kezeléséhez is támogatást nyújt. A további Android-alkalmazáskomponensek, mint a *Service*, a *ContentProvider* és a *BroadcastReceiver* nem fejleszthetők natívan.

A támogatott főbb natív headerök/könyvtárak a következők:

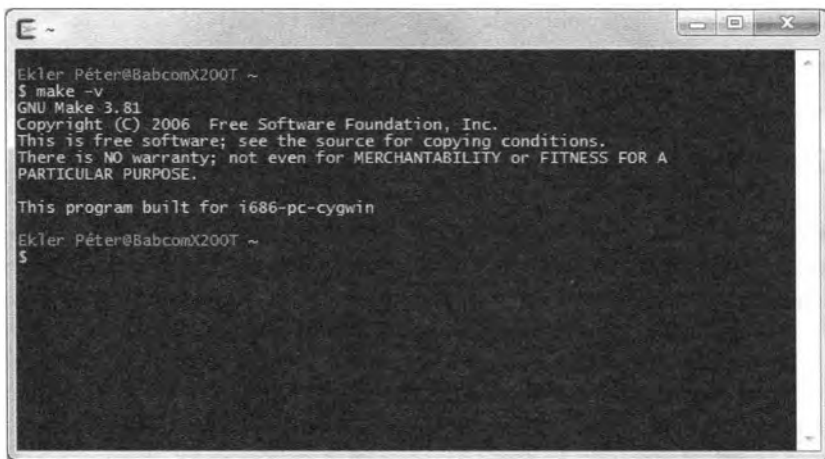
- *libc* headerök: C-könyvtár,
- *libm* headerök: *math* könyvtár,
- *JNI interface* headerök,
- *libz* headerök: *Zlib* tömörítő,
- *liblog* (Android *logging*) headerök: Android-naplózás,
- *OpenGL ES 1.1* és *OpenGL ES 2.0* headerök: 3D-s grafikus könyvtár,
- *libjnigraphics* headerök: pixelbuffer-elérés Android 2.2-től,
- C++-támogató headerök: korlátozott támogatás,
- *OpenSL ES*: natív audiokönyvtár,
- Android natív alkalmazáskönyvtár.

12.3.2. A fejlesztési környezet és az első natív modul

A natív Android-fejlesztés első lépése Eclipse környezet feltételezése esetén az Eclipse C/C++ plugin telepítése (CDT³⁴ – C/C++ Development Tooling). Emellett egy Linux-környezetre van szükség, amelyet Windows platform esetén a Cygwinnel válthatunk ki. A Cygwin telepítésekor elegendő a „Devel” ágat feltelepíteni, és ügyeljünk arra, hogy a telepítési útban ne legyen szóköz. A sikeres telepítés a következőképpen ellenőrizhető:

```
make -v
```

³⁴ Eclipse CDT plugin: <http://www.eclipse.org/cdt/>



12.14. ábra. Cygwin devel mód sikeres telepítésének ellenőrzése

Ezt követően töltsük le a legújabb NDK-t,³⁵ és tömörítsük ki például az SDK mellé (pl. c: \Android\android-ndk-r7\).

Miután a fejlesztési környezetet összeállítottuk, készítsük el az első Android-alkalmazásunkat, amely natív modult használ. A natív modulban helyezzünk el egy *sayHello()* függvényt és egy *add()* függvényt, előbbi szöveges értékkel tér vissza, míg utóbbi a számok összeadására használható. Valósítjuk meg, hogy a *sayHello()* függvény meghívódásakor a *LogCat*re is írjunk üzenetet natív oldalról.

Első lépésként hozzunk létre egy új Android-projektet egy szóköz nélküli elérési úton. Ezután a projekten belül hozzunk létre egy *jni* könyvtárat, ahova majd a natív kódrészek és a *make* állományok kerülnek. Első példánkban hozzuk létre a következő állományokat a *jni* könyvtáron belül:

- *Android.mk*: a források leírása,
- *Application.mk*: a használt natív modulok és a fordítási hardver megadása,
- *hello.c*: C/C++-forrás.

Az *Android.mk* tartalma legyen a következő:

```

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
# modul név, forrás állományok és használt könyvtárak
LOCAL_MODULE      := hello

```

³⁵ Android NDK: <http://developer.android.com/sdk/ndk/index.html>

```
LOCAL_SRC_FILES := hello.c
LOCAL_LDLIBS     := -llog

include $(BUILD_SHARED_LIBRARY)
```

Az *Application.mk* tartalma (az *APP_ABI*-hez a készülékünknek megfelelő értéket kell beírni) a következő:

```
APP_PLATFORM := android-9
APP_ABI := armeabi-v7a

Végül pedig a hello.c tartalma az alábbi:

#include <string.h>
#include <jni.h>

#include <android/log.h>

#define LOGI(...) ((void)__android_log_print(ANDROID_
LOG_INFO,
    "native-module", __VA_ARGS__))

jstring Java_hu_bute_daai_amorg_examples_NativeLib_
sayHello(
    JNIEnv* env, jobject javaThis) {
    LOGI("sayHello() meghívodott");
    return (*env)->NewStringUTF(env, "Hello Android -
nativ!");
}

jint Java_hu_bute_daai_amorg_examples_NativeLib_add(
    JNIEnv *env, jobject javaThis, jint value1, jint
value2) {
    return (value1 + value2);
}
```

Az *Android.mk* állományban látható, hogy a modul neve *hello* lesz, és egyetlen *hello.c* forrásból építkezünk, valamint használjuk a *log* könyvtárat. Az *Application.mk*-ban jelezzük, hogy a 9-es Android-verziót használjuk, és ARM processzor támogatással kívánjuk a natív modult fordítani.

A *hello.c* forrás az előzőeknél kicsit bonyolultabb. A forráskód elején a használt *header*állományokat látjuk. A naplózást például az *android/log.h*-n keresztül valósítjuk meg, és definiálunk egy *LOGI(...)* makrót, amellyel egyszerűen el tudjuk érni az *android_log_print(...)* függvényt, amely ténylegesen

a *LogCatre* ír. Ezt követi a *sayHello()* és az *add()* függvények definiálása, amelyeket JNI-n keresztül Java-oldalról hívunk meg. Ennek folytán ezek a függvények nevükben hordozzák a Java *package*-et és azt az osztályt, amelyen belül megvalósul a használatuk. Esetünkben ez a *NativeLib* osztály a *hu.bute.daa.amorg.examples* csomagban, így a függvények nevei *Java_hu_bute_daa_amorg_examples_NativeLib_sayHello(..)* és *Java_hu_bute_daa_amorg_examples_NativeLib_add(...)*. Mindkét függvény két kötelező paraméterrel rendelkezik, amely a hívási környezetet írja le, ezek után következhetnek a függvény saját paraméterei. A függvényparaméterek és a visszatérési értékek típusainál érdemes megfigyelni a 'j' előtagot, amely jelzi, hogy „Java-oldalról érkezik” a paraméter, vagy visszatérési értéként szerepel. Végül figyeljük meg a függvények implementációját, azt, hogy hogyan használjuk a LOGI makrót, továbbá hogyan kell szöveget alkotni.

Miután áttekintettük a natív forrást, a következő lépés, hogy lefordítsuk. Ehhez lépünk be az alkalmazásunk projektkönyvtárába, és adjuk ki az NDK könyvtárban szereplő *ndk-build* parancsot.

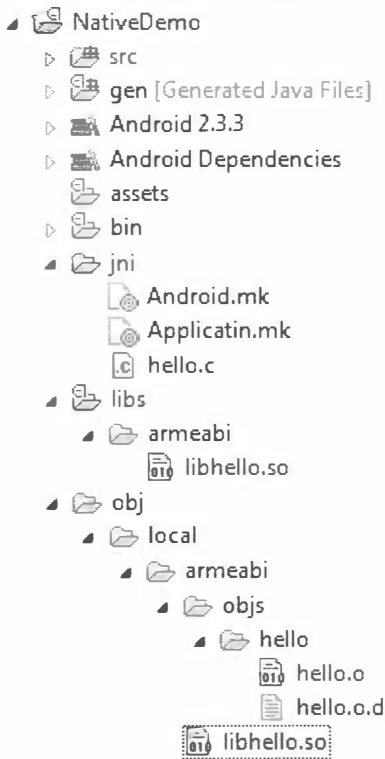
Cygwinnel való fordításkor körülbelül a következő eredményt láthatjuk:

```
user@hostmachine ~
$ cd /cygdrive/d/EclipseProjects/NativeDemo/

user@hostmachine /cygdrive/d/EclipseProjects/
NativeDemo
$ /cygdrive/c/Android/android-ndk-r7/ndk-build
Compile thumb   : hello <= hello.c
SharedLibrary   : libhello.so
Install         : libhello.so => libs/armeabi/libhello.
so

user@hostmachine /cygdrive/d/EclipseProjects/
NativeDemo
$
```

Miután sikerült a fordítás, az Eclipse-en belül frissítsük a projektet, hiszen a fordítás eredményeképpen új állományok kerültek a projekt könyvtárába, ilyenek például a *.so* (*shared object*) állományok. A következő ábra szemlélteti a fordítás eredményét az Eclipse-ben a frissítés után.



12.15. ábra. Natív fordítás eredménye Eclipse-ben, projektfrissítés után

A *libs* és az *obj* könyvtárakban az *Application.mk* állományban megadott *targetek*re lefordított állományok találhatók.

A natív kód fordítása után semmiképp se feledkezzünk meg a projekt frissítéséről az Eclipse-en belül, hiszen anélkül nem biztos, hogy fordításkor az Eclipse-környezet érzékeli a változást. Ennek eredménye lehet, hogy nem a frissen lefordított natív modul csomagolódik az *apk*-ba, mert a környezet feltételezheti, hogy nincs változás a natív oldalon.

Miután sikerült a natív modul fordítása, készítsük el az Android-alkalmazást, amely a natív modult felhasználja. Elsőként valósítsuk meg a *NativeLib* osztályt a *hu.bute.daai.amorg.examples* csomagban pontosan ilyen névvel, ez a natív függvényekhez nyújt interfészt (hiszen a natív függvények nevében szerepelt is ez a csomag- és osztálynév). Figyeljük meg a függvények előtt a *native* kulcsszót.

```
public class NativeLib {
    static {
        System.loadLibrary("hello");
    }
}
```

```
// native függvények - megfelelő névvel
public native String sayHello();
public native int add(int a, int b);
}
```

A natív modult használó Activity forrása, amely két gombnyomás hatására hívja meg a két natív függvényt, a következőképpen implementálható:

```
public class NativeDemoActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        final NativeLib nativeLib = new NativeLib();

        Button btnSayHello =
            (Button)findViewById(R.id.btnSayHello);
        btnSayHello.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String nativeHello = nativeLib.sayHello();
                Toast toast = Toast.makeText(NativeDemoActivity.
this,
                nativeHello, Toast.LENGTH_LONG);
                toast.setGravity(Gravity.TOP|Gravity.LEFT, 5,
200);
                toast.show();
            }
        });

        Button btnAdd = (Button)findViewById(R.id.btnAdd);
        btnAdd.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                int res = nativeLib.add(20, 12);
                Toast toast = Toast.makeText(NativeDemoActivity.
this,
                "20 + 12 = "+res, Toast.LENGTH_LONG);
                toast.setGravity(Gravity.TOP|Gravity.LEFT, 5,
200);
                toast.show();
            }
        });
    }
}
```

Az Activity forrását áttekintve láthatjuk, hogy itt egyszerűen csak használjuk a *nativeLib* objektum függvényeit. Látható továbbá, hogy a *Toast*ot nem az alapértelmezett helyen, hanem a bal felső saroktól x irányban 5, y irányban pedig 200 képpontnyira behúzva jelenítjük meg. Végül az Activityhez tartozó felhasználói felület a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
  http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical" >
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
  <Button
    android:id="@+id/btnSayHello"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/btnSayHello" />
  <Button
    android:id="@+id/btnAdd"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/btnAdd" />
</LinearLayout>
```

Az elkészült alkalmazást a következő ábra szemlélteti.



12.16. ábra. Natív függvény hívási eredménye

A kitömörített NDK könyvtárában szereplő példák közül érdemes még megnézni a *two-libs* példát,³⁶ azt, hogy hogyan kell a natív kódrészt strukturálni.

³⁶ Android NDK *two-libs* példa: <http://developer.android.com/sdk/ndk/overview.html#samples>

Nevezetesen: hogyan tudunk több forrásállományt és saját *header*állományt kezelni.

Tételezzük fel, hogy létezik egy *first.h*, egy *first.c* és egy *second.c* forrásállományokból álló natív modulunk.

A *first.h* tartalma, amely deklarálja a *first(...)* függvényt a következő:

```
#ifndef FIRST_H
#define FIRST_H

extern int first(int x, int y);

#endif /* FIRST_H */

A first.c tartalma az alábbi:

#include "first.h"

int first(int x, int y)
{
    return x + y;
}
```

A *second.c* tartalma, amely használja a *first.c*-ben definiált *first(...)* függvényt, a következő:

```
#include "first.h"
#include <jni.h>

jint
Java_com_example_twolibs_TwoLibs_add(JNIEnv* env,
    jobject this, jint x, jint y)
{
    return first(x, y);
}
```

Végül a legfontosabb, a forrásokat összefogó *Android.mk* állomány a következő:

```
LOCAL_PATH:= $(call my-dir)

# a first statikusan könyvtárként kerül fordításra
#
include $(CLEAR_VARS)

LOCAL_MODULE      := libtwolib-first
```



```

LOCAL_SRC_FILES := first.c

include $(BUILD_STATIC_LIBRARY)

# a second a first-től függ és tartalmazza azt
#
include $(CLEAR_VARS)

LOCAL_MODULE     := libtwolib-second
LOCAL_SRC_FILES  := second.c

LOCAL_STATIC_LIBRARIES := libtwolib-first

include $(BUILD_SHARED_LIBRARY)

```

A *first* modul fordítása statikus könyvtárként történik, míg a *second* lesz az osztott könyvtár *libtwolib-second* néven, amelyet Java-oldalról érünk el.

Az így elkészített natív komponenst felhasználó Activity a következő:

```

public class TwoLibs extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        int x = 1000;
        int y = 42;
        // dinamikus könyvtárbetöltés még a natív hívás
        előtt
        System.loadLibrary("libtwolib-second");
        int z = add(x, y);
        tv.setText("x+y="+z);
        setContentView(tv);
    }

    public native int add(int x, int y);
}

```

12.3.3. A készülék érzékelőinek használata natív oldalról

Az Android-alapú eszközök általában többféle érzékelővel rendelkeznek, ilyen például a magnetométer, a gyorsulásérzékelő, a fényérzékelő stb., amelyeket a Java-oldali Sensor API segítségével könnyen elérhetünk. A Sensor API alapját a *SENSOR_SERVICE* és a *SensorManager* adja, amely a korábban bemutatott rendszerszolgáltatásokhoz hasonlóan működik.

Ebben a részben az érzékelők natív oldalról történő elérését mutatjuk be, hiszen erre sokszor szükség lehet, például, ha egy érzékelő értékeire sűrűbben van szükség, mint ahogy az a hagyományos Sensor API-n keresztül elérhető.

Mielőtt a natív érzékelő használatát bemutatnánk, egy rövid példát ismer-tetünk, ahol egy Activityben érjük el a gyorsulásmérő adatait.

```
public class SensorDemoActivity extends Activity
    implements SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mAccelerometer;
    TextView tvAccX;
    TextView tvAccY;
    TextView tvAccZ;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        tvAccX = (TextView) findViewById(R.id.tvAccX);
        tvAccY = (TextView) findViewById(R.id.tvAccY);
        tvAccZ = (TextView) findViewById(R.id.tvAccZ);

        mSensorManager =
            (SensorManager) getSystemService(SENSOR_SERVI-
CE);
        mAccelerometer =
            mSensorManager.getDefaultSensor(
                Sensor.TYPE_ACCELEROMETER);
    }

    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener(this,
mAccelerometer,
            SensorManager.SENSOR_DELAY_NORMAL);
    }

    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }

    public void onAccuracyChanged(Sensor sensor, int
accuracy) {
    }
}
```

```

    public void onSensorChanged(SensorEvent event) {
        if (event.sensor.getType() == Sensor.TYPE_
ACCELEROMETER) {
            tvAccX.setText("X: " + event.values[0]);
            tvAccY.setText("Y: " + event.values[1]);
            tvAccZ.setText("Z: " + event.values[2]);
        }
    }
}

```

Az Activityt áttekintve látható, hogy nagyon egyszerűen elérhetők a gyorsulásmérő x, y és z tengely irányú összetevői. Elsőként egy *SensorManager* példányra van szükség, majd annak segítségével a gyorsulásérzékelőt jelképező *Sensor* objektumra. Ezt követően az *onResume()* függvényben a *SensorManager registerListener()* függvényében elindítjuk az érzékelést, és az eredmények az *onSensorChanged()* függvényen keresztül érhetők el folyamatosan a *SensorEventListener* interfészen keresztül. Az érzékelést az *onPause()*-ban az *unregisterListener()* függvény segítségével felfüggesztjük, így nem pazaroljuk feleslegesen az erőforrásokat, amikor háttérbe kerül az alkalmazás.



12.17. ábra. Gyorsulásmérő értékeinek a megjelenítése

Miután megismerkedtünk a *Sensor* API-val, vizsgáljuk meg, hogy natív oldalról hogyan érhetők el az érzékelők értékei. Példánkban a fényérzékelőt mutatjuk be, de a többi érzékelő is hasonlóképpen használható. Natív oldalról az érzékelők eléréséhez a *sensor.h* header beillesztésére (include) lesz szükség, amely csak a 2.3-as (9-es) Android-verziótól érhető el.

Készítsünk egy alkalmazást, amely a fényérzékelő értékét jeleníti meg egy Activityben úgy, hogy a fényérzékelő kezelését natív oldalon valósítjuk meg. Elsőként készítsünk egy *sensor.c* állományt, amelynek tartalma a következő:

```

#include <string.h>
#include <jni.h>

#include <android/log.h>
#include <android/sensor.h>
#include <android/looper.h>

#define LOGI(...) ((void)__android_log_print(ANDROID_

```

```

LOG_INFO,
    "native-module", __VA_ARGS__)

struct engine {
    ASensorManager* sensorManager;
    const ASensor* lightSensor;
    ASensorEventQueue* sensorEventQueue;
};

struct engine engine;
float lightValue;

static int get_sensor_data(int fd, int events, void*
data);

jfloat Java_hu_bute_daai_amorg_examples_
NativeSensorLib_getLightSensorValue() {
    return lightValue;
}

void Java_hu_bute_daai_amorg_examples_NativeSensorLib_
startLightSensor(JNIEnv* env, jobject thiz) {
    LOGI("Light sensor inditasa");
    void* sensor_data = malloc(1000);
    ALooper* looper = ALooper_forThread();
    if (looper == NULL) {
        looper =
            ALooper_prepare(ALOOPER_PREPARE_ALLOW_NON_
CALLBACKS);
    }

    engine.sensorManager = ASensorManager_getInstance();
    engine.lightSensor = ASensorManager_
getDefaultSensor(
    engine.sensorManager, ASENSOR_TYPE_LIGHT);
    engine.sensorEventQueue = ASensorManager_
createEventQueue(
    engine.sensorManager, looper, 1, get_sensor_data,
    sensor_data);

    ASensorEventQueue_enableSensor(engine.
sensorEventQueue,
    engine.lightSensor);
    // Mintavételi gyakoriság: 100Hz,
    // utolsó paraméter microsec-ben
    ASensorEventQueue_setEventRate(engine.
sensorEventQueue,
    engine.lightSensor, 100000);
}

```

```

void Java_hu_bute_daai_amorg_examples_NativeSensorLib_
stopLightSensor() {
    LOGI("Light sensor leallitva");
    if (engine.lightSensor != NULL) {
        ASensorEventQueue_disableSensor(engine.
sensorEventQueue,
        engine.lightSensor);
    }
}

int get_sensor_data(int fd, int events, void* data) {
    ASensorEvent event;
    while (ASensorEventQueue_getEvents(engine.
sensorEventQueue,
        &event, 1) > 0) {
        if (event.type == ASENSOR_TYPE_LIGHT) {
            lightValue = event.light;
            LOGI("Light sensor value: %f", lightValue);
        }
    }
    // 1-el való visszatérés esetén újból meghívódnak a
    // callback-ek, 0 esetén nem (unregister)
    return 1;
}

```

Nézzük át a natív forrást. Látható, hogy három függvény lesz majd elérhető a későbbi *NativeSensorLib* osztályban, a *getLightSensorValue()*, a *startLightSensor()*, és a *stopLightSensor()*. A *getLightSensorValue()* függvény feladata csupán annyi, hogy az aktuális fényértéket tároló *lightValue* változó értékét visszaadja. A fényérzékelés során mindig ebbe mentjük el az aktuális értéket.

A *startLightSensor()* függvény felelős a fényérzékelés indításáért. Elsőként egy ütemezőre van szükség (*ALooper*), amely felkészíti a rendszert a *callback*ek kezelésére. Ezt követően értéket adunk az *engine* struktúrában tárolt *sensorManager*, *lightSensor* és *sensorEventQueue* elemeknek. Az *ASensorManager_createEventQueue()* függvényben hozzuk létre az érzékelő olvasásáért felelős „sort”, és itt kell megadni, hogy melyik callback függvénybe érkezzenek az adatok. Esetünkben ez a *get_sensor_data()* függvény lesz, amelyben a fényérzékelő értékét kiolvassuk, eltároljuk a *lightValue* változóban, és egyben naplózzuk is az értéket. A függvény 1 visszatérési értéke jelzi, hogy továbbra is kíváncsiak vagyunk az újabb értékekre. A 0-val való visszatérés az érzékelő eseményéről való leiratkozást jelentené.

Végül a *stopLightSensor()* függvény szerepe csupán annyi, hogy az *ASensorEventQueue_disableSensor()* függvény segítségével leállítja az érzékelő figyelését.

A megoldáshoz tartozó *Android.mk* állomány tartalma a következő:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
# modul név és forrás állományok
LOCAL_MODULE     := sensor
LOCAL_SRC_FILES  := sensor.c
LOCAL_LDLIBS     := -llog -landroid

include $(BUILD_SHARED_LIBRARY)
```

A natív modul használatához tehát szükség van egy *NativeSensorLib* Java-osztályra a *hu.bute.daa.iamorg.examples* csomagban, ahogy az a natív függvények nevéből már előre látható volt.

```
public class NativeSensorLib {
    static {
        System.loadLibrary("sensor");
    }

    // native függvények
    public native void startLightSensor();
    public native void stopLightSensor();
    public native float getLightSensorValue();
}
```

A natív modult használó Activity a következő:

```
public class NativeSensorDemoActivity extends Activity
{
    private NativeSensorLib nativeLib;
    private TextView tvStatus;
    private Handler mHandler = new Handler();
    private boolean lightSensorEnabled = true;

    private final Runnable mRunnable = new Runnable() {
        public void run() {
            if (lightSensorEnabled) {
                float lightValue = nativeLib.
getLightSensorValue();
                tvStatus.setText("Light sensor value:
"+lightValue);
                mHandler.postDelayed(mRunnable, 200);
            }
        }
    }
}
```



```

};

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    nativeLib = new NativeSensorLib();

    tvStatus = (TextView)findViewById(R.id.tvStatus);

    Button btnStartLight =
        (Button)findViewById(R.id.btnStartLight);
    btnStartLight.setOnClickListener(new
OnClickListener() {
        @Override
        public void onClick(View v) {
            nativeLib.startLightSensor();
            lightSensorEnabled = true;
            mHandler.postDelayed(mRunnable, 200);
        }
    });
    Button btnStopLight =
        (Button)findViewById(R.id.btnStopLight);
    btnStopLight.setOnClickListener(new
OnClickListener() {
        @Override
        public void onClick(View v) {
            tvStatus.setText("");
            lightSensorEnabled = false;
            nativeLib.stopLightSensor();
        }
    });
}

@Override
protected void onStop() {
    super.onStop();
    if (lightSensorEnabled) {
        lightSensorEnabled = false;
        nativeLib.stopLightSensor();
    }
}
}

```

Az Activity az *onCreate()* függvényében példányosítja a *NativeSensorLib* objektumot és a *Start*, illetve a *Stop* gombok hatására a *NativeSensorLib*en keresztül elindítja/leállítja a fényérzékelést. A fényérzékelő adatainak kiolvasásáért egy ütemezett feladat a felelős, amely példánkban 200 milliszekundumonként frissíti az értéket a natív modulon keresztül, és jeleníti meg egy *TextView*-n.



12.18. ábra. Fényérzékelő adatainak megjelenítése natív modulon keresztül

TÁRGYMUTATÓ

A, Á

adb, 19, 20, 22
ActionBar, 331, 349, 351, 354
ADT, 19
AIDL, 326
ALooper, 376, 377
ANDROID_SDK_HOME, 19
AndroidHttpClient, 229-244
ANR, 310
Apache, 6, 229, 239
API Demos, 82
apk, 15, 16, 18, 34, 75, 282, 339, 369, 371
APN, 214
application, 15, 34, 64, 99, 107, 145, 152, 202, 251, 309, 351, 369
ARM, 364, 367
ASensorEventQueue, 376, 377
ASensorManager, 376, 377
AsyncTask, 235, 239, 312
AudioManager, 298, 301
AutoCompleteTextView, 88, 89, 90
AVD, 19, 20, 71, 72, 202
AVD Manager, 20, 24

B

BasicResponseHandler, 233, 244
BitmapFactory, 246, 283, 294
Bound Service, 309-315
BroadcastReceiver, 28, 32, 35, 52, 130, 152, 199, 200, 215, 251, 267, 277, 319, 365
BSSID, 182
Build Target, 202, 203

C

C2DM, 250, 251
cache, 133, 153, 157, 159

Camera, 280-296
CDMA, 186, 262, 264
CDT, 365
ConnectivityManager, 213, 214
ContentProvider, 28, 32, 129, 171, 176, 280, 365
ContentValues, 170, 175
Criteria, 198
CRUD, 174
Cupcake, 6, 7, 12
Cursor, 170, 177, 179
CursorAdapter, 177
CursorFactory, 173
CSS, 222
Cygwin, 365, 368

D

Dalvik, 4, 15, 17, 24, 129
DDMS, 20, 24, 25, 197, 296
DefaultHttpClient, 229, 239
Deprecated, 47
DocumentBuilder, 245, 246
DocumentBuilderFactory, 245, 246
DOM, 245
Donut, 7, 12
DownloadManager, 241
dp, 66, 68, 71, 86
dpi, 66, 86

E, É

Eclair, 7, 12
Eclipse, 19, 23, 24, 34, 64, 101, 365, 368, 369
élő háttérkép, 102, 103, 106
Emulátor, 19, 22, 24, 71, 102, 197, 202, 284
ESN, 263
Explicit Intent, 138, 139
External Storage, 155, 158, 281, 303

F

FaceDetector, 285
FaceView, 285, 288
FAT, 158
FileInputStream, 156, 157, 283, 287
FileOutputStream, 156
finalizer, 41, 42
Fragment, 11, 74, 165
frame-by-frame, 92
Froyo, 8, 9, 12

G

Geocoding, 200, 201, 243
GeoPoint, 204, 209, 210
GET, 107, 109, 229, 232, 233, 240, 248
Gingerbread, 9, 12
GPS, 2, 102, 164, 181, 194, 198, 307
GSM, 2, 186, 189, 262, 274
GsmCellLocation, 184, 191, 259

Gy

gyorsítótárak, 157

H

Handler, 233, 235, 252, 315, 322, 378
Honeycomb, 10, 12, 165
hotkey, 23, 24
HttpClient, 229-239, 244
HttpEntity, 231, 237
HttpPost, 239, 240
HttpResponse, 231, 233, 240
HttpStatus, 231, 237
HttpURLConnection, 241

I, í

IBinder, 318, 325, 326
Ice Cream Sandwich, 11, 12
IMEI, 184, 185, 192, 263
Implicit Intent, 138, 146, 152, 266, 276

IMSI, 185, 192, 264
InputStream, 156, 157, 231, 248, 249, 321
Intent Flags, 137, 138, 219, 313, 318
Intent objektum, 58, 110, 130, 146, 281
Intent-kategóriák, 135
IntentService, 308, 320
Intent-szűrő, 135, 146, 152
Internal Storage, 155
ItemizedOverlay, 207

J

JavaDoc, 42, 47
JavaScript, 8, 222, 228
JNI, 17, 365, 368, 375
JSON, 213, 242, 244,
JSONArray, 242, 244
JSONObject, 242, 244

L

LAC, 184
launchMode, 57
LayoutInflater, 114, 115, 177
Linkify, 148
Linux kernel, 4, 9, 14
ListActivity, 110, 116, 128
LocationListener, 193, 195
LocationManager, 192, 195, 200
LogCat, 24, 48, 64, 238, 366, 368

M

MAC, 182
Manifest, 16, 19, 29, 34, 64, 70, 107, 121, 152, 185, 203, 223, 251, 280, 303, 308, 329, 347, 349
MapActivity, 204, 205
MapView, 82, 202-210
MediaPlayer, 297, 299, 305
MediaRecorder, 288, 303
MediaScanner, 159
MediaStore, 140, 281, 283

MEID, 263
 MenuItem, 61, 91, 119, 150
 MEO, 183
 Messenger, 314, 321, 323
 MIME, 134, 146, 159, 276
 MMS, 255, 277
 Mobile Country Code (MCC), 264
 mobilhálózat-alapú helymeghatározás,
 181, 182

N

Naplózás, 48, 49
 natív broadcast-Intent, 153
 NDK, 17, 33
 Nexus, 1, 2, 9
 nomedia, 160
 Notification, 30, 147, 184, 219, 221,
 297, 298, 327
 NotificationManager, 218, 219, 221
 NotificationBar, 218

O,Ó

ObjectAnimator, 346
 Open Handset Alliance, 4
 openBmap, 182, 183
 OpenCellId, 183
 OutputStream, 248, 249
 Overlay, 203, 208, 210

P

Parcelable, 136, 143, 321, 326
 PDU, 275, 277
 PendingIntent, 109, 110, 137, 147,
 199, 200, 219, 221, 273, 275,
 328
 PhoneStateListener, 186-191, 255,
 263
 PictureCallback, 292, 293
 POST, 229, 239, 250
 PreferenceActivity, 162, 165, 166
 PreferenceCategory, 163, 164
 PreferenceScreen, 163, 164

preview, 280, 288, 291
 ProximityAlert, 199, 200
 Push, 2, 8, 213, 250, 291

R

Reverse Geocoding, 200
 RingtoneManager, 297-299
 runOnUiThread, 233, 235, 248

S

sandbox, 27, 129
 screen density, 66
 screen size, 66
 SDK, 17, 24, 35, 71, 82, 202, 326, 363,
 366
 SDK Manager, 20
 Sepia, 293
 Serializable, 136, 143
 ServiceConnection, 324, 326
 SharedPreferences, 160, 162, 166
 ShareActionProvider, 353, 354
 ShutterCallback, 293
 SignalStrength, 187, 189, 262, 263
 SIM, 192, 255, 265, 273
 SMS, 22, 24, 250, 255, 277
 SmsMessage, 277
 SNR, 262
 Socket, 213, 248, 249
 SPN, 265
 SQLite, 14, 169, 171, 174
 SQLiteOpenHelper, 170, 173
 Started Service, 309, 311, 324
 style, 64, 85, 121, 210, 286
 SuppressWarnings, 47, 48
 SurfaceHolder, 105, 288, 291, 292
 SurfaceView, 288, 291
 sűrűségfüggetlen, 66, 68

T

táblagép, 3, 10, 11, 66, 74, 255, 264
 TAC, 185
 taskAffinity, 57

TCP/IP, 213, 247
TelephonyManager, 184, 190, 255,
270
theme, 64, 87, 121
Toast, 61, 62, 87, 116, 117, 141, 187,
232, 247, 268, 270, 305, 322
ToneGenerator, 297
TrafficStats, 252, 253
tweened animáció, 92

U

unmount, 158
URI, 32, 133, 143, 176, 281, 300
URLConnection, 229
URLEncoder, 230, 238

V

VideoView, 296, 297
ViewGroup, 73, 114, 177

W

WakeLock, 301, 302
WebChromeClient, 226, 227
WebKit, 14, 222
WebView, 80, 81, 213, 225, 227
WebViewClient, 224, 225
widget, 10, 58, 74, 96, 106, 110, 147,
167
ViewPager, 347, 362
WifiManager, 214, 217, 302

Y

yaffs, 9, 158