

# ANDROID ALKALMAZÁSFEJLESZTÉS

**Optimalizálás**

Hatékony alkalmazás

Androidra



[sicz-mesziar.janos@nik.uni-obuda.hu](mailto:sicz-mesziar.janos@nik.uni-obuda.hu)

**Sicz-Mesziár János**

2013. december 3.

# Irányelvek

1. Első sorban arra kell törekedni, hogy jó programot írjunk, ne minden áron gyorsat!
2. Teljesítmény szempontjából fontoljuk meg az API-k tervezését, használatát.
3. Mérjük a teljesítményt az optimalizálás előtt és után.
4. Optimalizáljunk, ahol ésszerű, és lehetséges, de **ne rombolja a felhasználói élményt.**
5. Tartsuk be a "design pattern"-ek szabályait.

# Teljesítmény centrikus változások

Verzió	Gyorsítás	Hatása
Android 2.2 <i>api level: 8</i>	<u>JIT</u>	Just In Time fordító, elemzi az alkalmazás kódját, utána futási időben a byte kódot befordítja optimalizált natív kódra. Ez ~2x-5x gyorsítás.
Android 2.3 <i>api level: 9</i>	<u>GC optimization</u>	Régebben a GC meglehetősen sokáig futott : stop-the-world, full heap collection, >100ms Gingerbread óta: konkurens, részleges takarítás, <5ms
Android 3.0 <i>api level: 11</i>	<u>Hardware accelerate</u>	A 2D grafikában is GPU hardveres gyorsítás, korábban csak OpenGL-ként volt elérhető. A változással a hagyományos UI elemek rajzolása jelentősen gyorsult.
Android 3.0 <i>api level: 11</i>	<u>Multicore support</u>	Több magos készülékek rendszer szintű támogatása. Akár egy szálú programokat is gyorsíthat: egyik szál alkalmazás, másik szál a GC.
Android 4.1 <i>api level: 16</i>	<u>Project Butter</u>	Kimondottan gyorsítás volt a cél: <ul style="list-style-type: none"><li>• Vsync timing: 16ms heartbeat,</li><li>• Triple buffering: 3x bufferelés,</li><li>• Synchronizing touch: kisebb késleltetés,</li><li>• CPU input boost: gyorsabb CPU ébresztés</li></ul>
Android 4.4 <i>api level: 19</i>	<u>Memory-optimization</u> <i>ART első megjelenése</i>	Memória optimalizálás, kevesebb heap: <ul style="list-style-type: none"><li>• Low-memory device support, min 512MB</li><li>• Kernel samepage merging (ksm), swap to zRAM</li></ul>

# Java: objektumok használata

## Objektumok létrehozásának elkerülése:

- Például több dimenziós tömbök helyett, 2 párhuzamos egy dimenziós tömb használata.

```
Class X{  
    Foo a;  
    Bar b;  
}
```

```
Foo[];  
Bar[];
```

## Belső Getter/Setter használatának mellőzése:

- OOP elvek követése erősen ajánlott. Kifelé public Getter/Setter használata, de belső értékadás közvetlenül történjen!

```
Class X{  
    private int a;  
    public void do(){  
        setA(1027);  
    }  
}
```

```
Class X{  
    private int a;  
    public void do(){  
        this.a = 1027;  
    }  
}
```

Gyorsítás: **3x**  
JIT-el: **7x**

[Mi az a JIT compiler?](#)

# Java: típusok, megkötések

## ENUM használatának elkerülése

- ENUM használata kényelmes, de ne használjuk ha a sebesség számít! Helyette alkalmazzunk integer egészeket!

## Static használata

- Ha nem szükséges egy objektum mezőjéhez hozzáférni, akkor érdemes static megkötést használni.

## Final static megkötés konstansoknál

- Fordító generál egy osztály inicializálót (<clinit>), ami első használatkor fut le.
- Ha static-ott használunk a továbbiakban nincs szüksége a <clinit>-re.

```
static int intVal = 42;  
static String strVal = "Hello, world!";
```

```
static final int intVal = 42;  
static final String strVal = "Hello, world!";
```

Gyorsítás:  
**15-20%**

***Ez az optimalizálás csak primitív típusokra és String konstansokra érvényes!***

# Java: for(each) előnyben részesítése

```
static class Foo { int mSplat; }
```

```
Foo[] mArray = ...  
public void zero() {  
    int sum = 0;  
    for (int i = 0; i < mArray.length; ++i)  
        sum += mArray[i].mSplat;  
}
```

```
public void one() {  
    int sum = 0;  
    Foo[] localArray = mArray;  
    int len = localArray.length;  
    for (int i = 0; i < len; ++i)  
        sum += localArray[i].mSplat;  
}
```

```
public void two() {  
    int sum = 0;  
    for (Foo a : mArray)  
        sum += a.mSplat;  
}
```

## Leglassabb:

Mert a JIT még nem tudja optimalizálni a tömb hosszának egyszeri számítását.

## Gyorsabb:

Mindent helyi változóba tesz → csökkenti a kereséseket. Tömb hosszának számítása gyorsabb.

## Leggyorsabb:

Gyorsulás a JIT nélküli készülékeken. De a JIT-el rendelkezőkön nincs észlelhető különbség az előző megoldással szemben.

# Java: rendszer API-k és szerkezetek

## StringBuilder

Látványos diagramok a különbségről!

- String: ha a szöveg nem változik
- StringBuffer: változik a szöveg – több szálon (thread safe)
- StringBuilder: változik a szöveg, **gyorsabb** – csak 1 szálon (ha a szöveg hosszát előre megadjuk **még gyorsabb**)

## System.arraycopy()

Gyorsítás: **9x**

- ~9x gyorsabb Nexus One készüléken - JIT-el, mintha kézzel írnánk meg.

## Listener objektumok elkerülése

Futási időben spórolunk: **1KB**

- Inkább használjuk a *this* kulcsszót, új Listener objektumok helyett!

## ArrayList vs. LinkedList vs. Vector vs. HashSet

- [http://balazs.pergamen.hu/?page\\_id=885](http://balazs.pergamen.hu/?page_id=885)
- <http://www.javacodegeeks.com/2010/08/java-best-practices-vector-arraylist.html>

ArrayList.isEmpty() vs. ArrayList.size() > 0

HashMap<Integer, String> vs. SparseArray()

# Java: munka a számokkal

## Lebegőpontos számokról jó tudni

- Android készülékeken szemmértékre a lebegőpontos ábrázolás 2x lassabb, mint az egészszámok esetén.  
Lásd.: **Location(double, double) vs. GeoPoint(int, int)**
- Sebességre a float és a double ~között nincs különbség. 😊  
De a double 2x nagyobb. → **ha lehet float-ot használjunk!**

## Shiftelés

- Ha kettő hatványaival végzünk osztást, vagy szorzást, akkor a biteltolás módszere sokkal gyorsabb.

```
int a = 4320;
int x = a / 2; // 2160
int x = a / 4; // 1080
int x = a / 8; // 540
int x = a * 2; // 8640
int x = a * 4; // 17280
```

```
int a = 4320;
int x = a >> 1;
int x = a >> 2;
int x = a >> 4;
int x = a << 1;
int x = a << 2;
```



# Android: teljesítmény mérése

Ajánlott optimalizálás előtt és után is mérni.

Így látni fogjuk, hogy a gyorsítás ért-e egyáltalán valamit.

Példakód az idő mérésére:



```
long start = System.currentTimeMillis();  
// System.nanoTime();
```

```
/* Kódok, amelyek teljesítményére kíváncsiak vagyunk. */
```

```
long end = System.currentTimeMillis();  
Log.i("M", String.valueOf(end - start));
```

# Android: memory leak

## Drawable, Bitmap resource-ok

- Telefon megdöntésekor az Activity újraindul és újratölti a forrásokat. → Képek esetén ez memória szivárgást jelent.
- Megoldás:

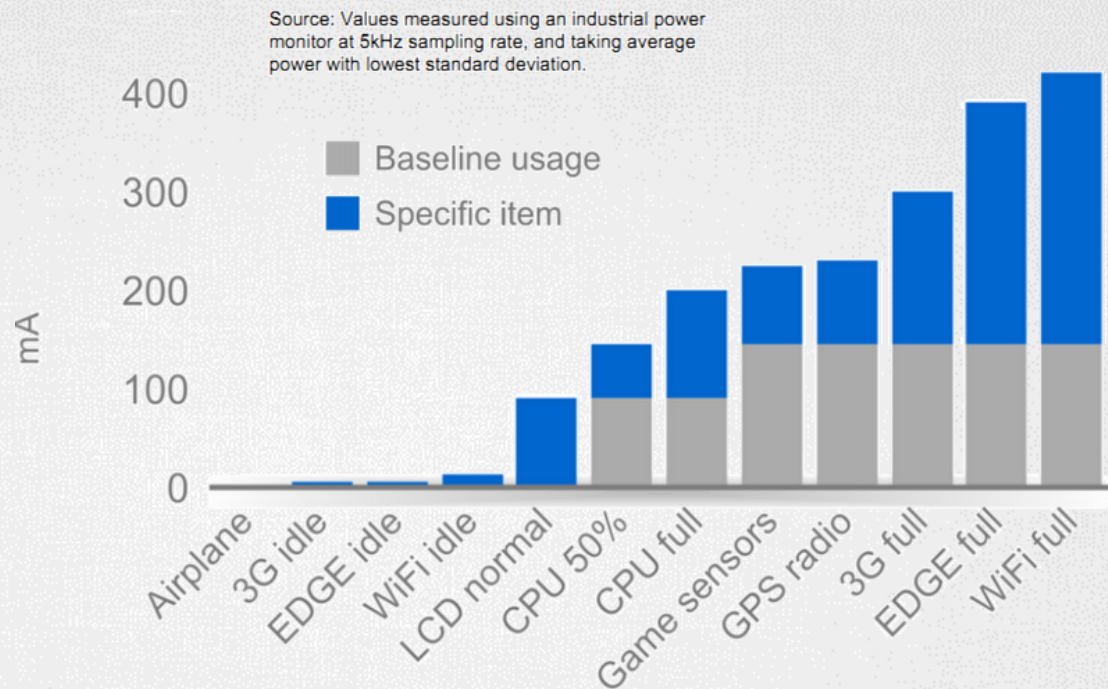
```
static Bitmap d;  
public void onCreate(Bundle ...){  
    if(d == null) d = Bitmap.decodeResource(...);  
}
```

## Erőforrás felszabadításokról ne feledkezzünk meg!

- DB.close();
- Input/OutputStream.close();
- Bitmap.recycle();
- Camera.release();
- System.GC(); // Csak ha szükségesnek látjuk

# Android: szenzorok fogyasztás

## Hálózati eszközök fogyasztása



## Szenzorok fogyasztása

*HTC Dream esetében*

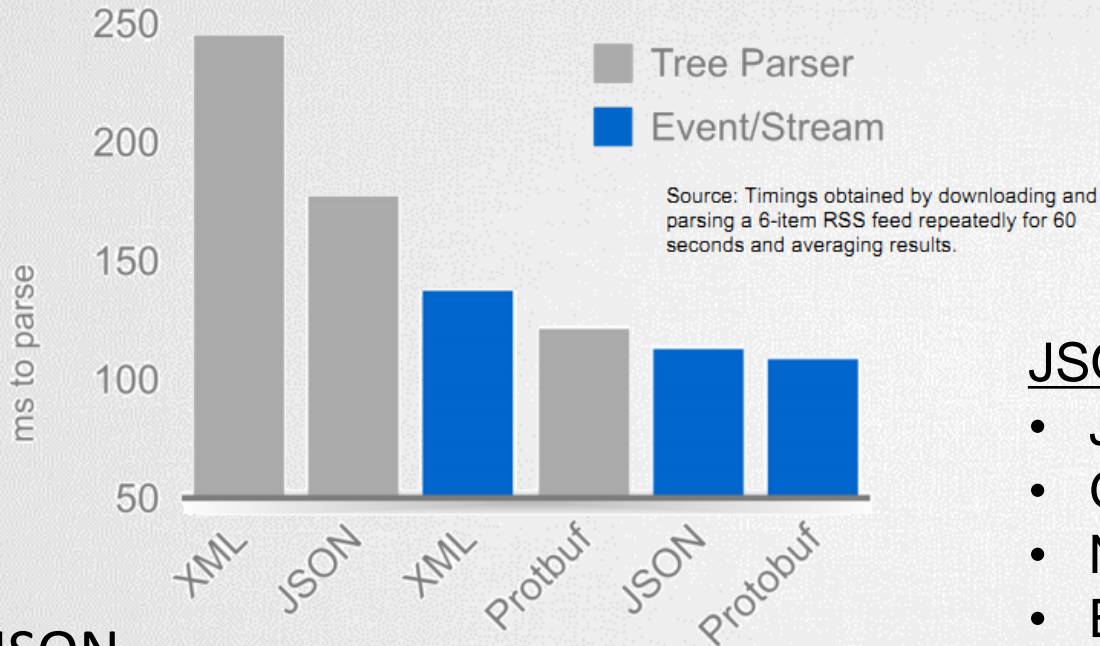
- Accelerometer/magnetic sensors
  - Normal: 10mA (used for orientation detection)
  - UI: 15mA (about 1 per second)
  - Game: 80mA
  - Fastest: 90mA

Forrás:

[http://dl.google.com/io/2009/pres/W\\_0300\\_CodingforLife-BatteryLifeThatIs.pdf](http://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf)

# Android: adatformátum és feldolgozás

## Feldolgozási idő



### JSON vs XML:

- JSON tömörebb
- Gyorsabb feldolgozás
- Natív API támogatás
- Egyszerű használat

### JSON

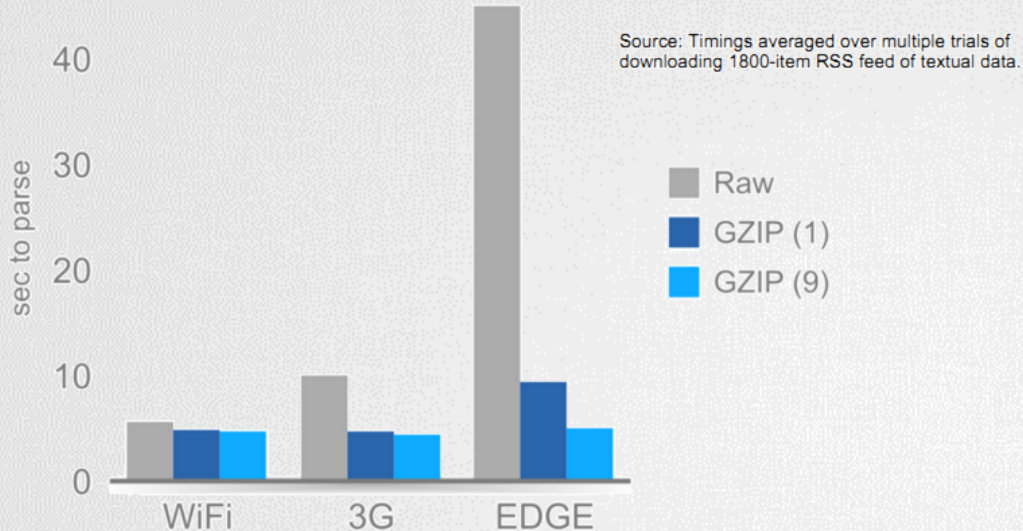
- <http://www.json.org/>
- <http://en.wikipedia.org/wiki/JSON>

### Protocol Buffers

- <http://code.google.com/p/protobuf/>

# Android:

## Nyers adat vs. GZIP (1) vs. GZIP (9)



Szöveges adatok  
tömörítése, ahol  
lehetséges!

- ◎ URLConnection használata HTTPClient helyett

- [http://www.innovation.ch/java/HttpClient/urlcon\\_vs\\_httpclient.html](http://www.innovation.ch/java/HttpClient/urlcon_vs_httpclient.html)

- ◎ Letöltés hatékonyságának fokozása

- <http://developer.android.com/training/efficient-downloads/efficient-network-access.html>

- ◎ ResponseCache, LastModify, If-Modify-Since

- <http://android-developers.blogspot.hu/2011/09/androids-http-clients.html>

# Android: UI gyorsítás

## Background drawable eltávolítása

- Alapértelmezett háttér eltávolítása gyorsít. (Csak ha nincs rá szükségünk, mert sajátot használunk)
- Gyorsulás oka a memória buszsebességéből ered.

```
<resources>  
  <style name="Theme.NoBackground" parent="android:Theme">  
    <item name="android:windowBackground">@null</item>  
  </style>  
</resources>
```

## Gyors orientáció váltás

- AndroidManifest.XML / adott Activity :  
configuration change = "orientation|screenSize"
- Következményei:
  - Döntéskor nem indul újra az életmodell ciklus.
  - Nem működik az alternatív minősítő az orientációra.

A felhasználó kezeli  
az orientációt!



# Android: UI gyorsítás (2)

## Layout hierarchia csökkentése

- Sok View → lassabb mérés, indulás, rajzolás, ...
  - ImageView + TextView, helyett → TextView és drawableLeft
  - "layout\_weight" megoldás használata költséges!
- Mély hierarchiák elkerülése! → StackOverflowException
  - RelativeLayout előnyben részesítése (flat hierarchia)
  - Hierarchyviewer használata , lásd még: layoutopt!
  - ScrollView is lehet root az XML-ben!

## Touchscreen érintésének eseménygyakorisága

- A DOWN és az UP action jellemzően egy érintés alatt 1x-1x fut le, míg MOVE számtalanszor a mozgás alatt.

Ennek ismeretében **összehasonlítást** **spórolhatunk**, ha MOVE action-t előbb vizsgáljuk!

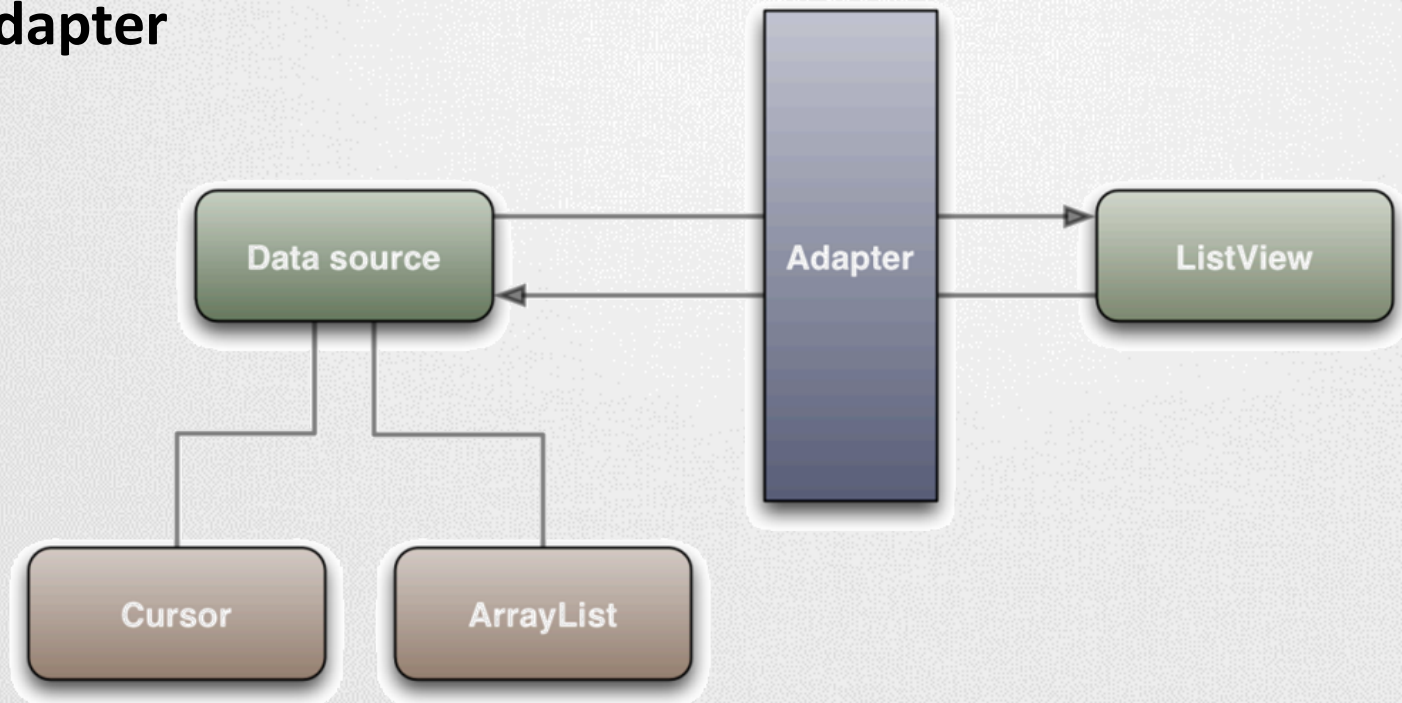
```
switch(event.getAction()){  
    case MotionEvent.ACTION_MOVE: break;  
    case MotionEvent.ACTION_DOWN: break;  
    case MotionEvent.ACTION_UP: break;  
}
```

# Android: UI gyorsítások (3) - Adapterek

## Adapter-ek:

- Sok elemszámú „listák” kiszolgálása hatékonyan.
- View példák ([AdapterView leszármazottak](#)):  
ListView, Gallery, GridView, Spinner, ViewPager, ...
- ArrayAdapter, CursorAdapter, SpinnerAdapter, ...

## BaseAdapter



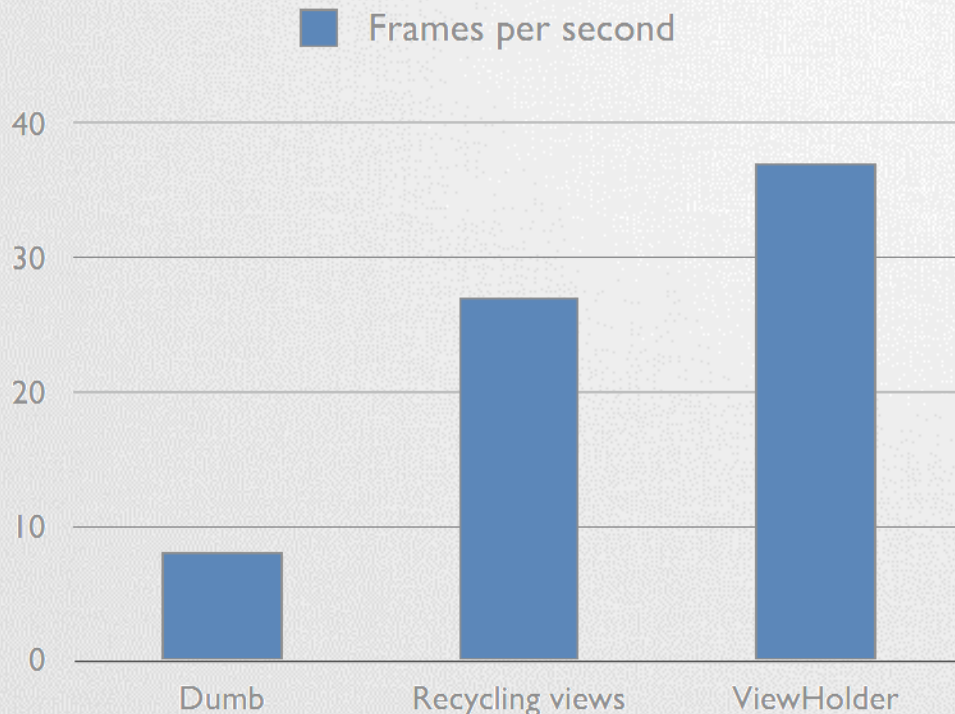


# Android: UI gyorsítások (4) - Adapterek

## Probléma:

- Minden pozícióban: `Adapter.getView()`;
- Minden esetben új View objektum költséges!
- Több ezer elem esetén?

**Megoldás:** Látható UI elemek újrahaznosítása!



**Forrás:**  
[Google I/O - 2009](#)

**Példa-kód:**  
[ListView](#)

# Android: UI gyorsítások (5)

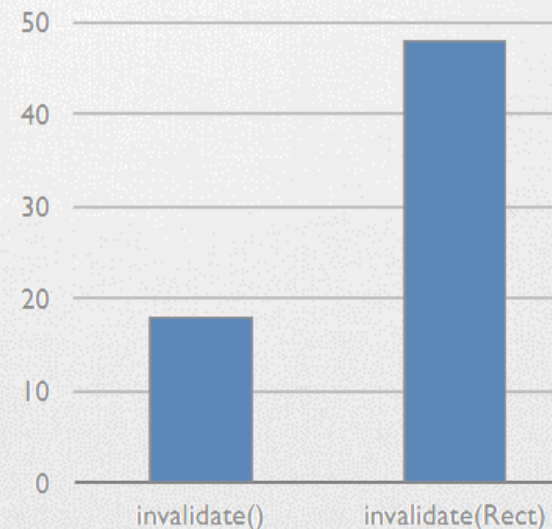
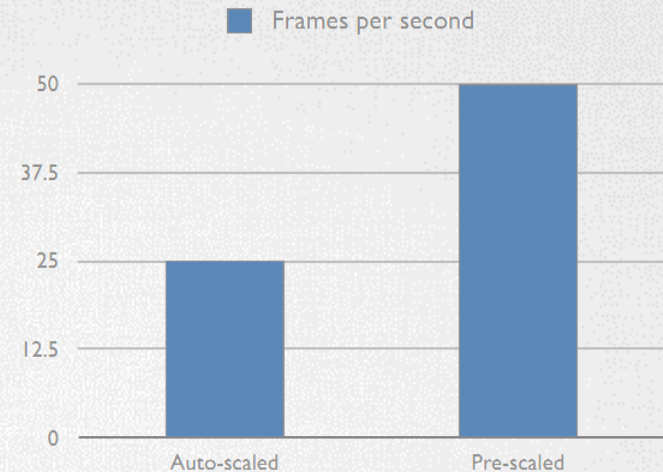
## Futtás idejű méretezés költséges

Könnyen orvosolható: **Pre-Scale**

```
originalImage = Bitmap.createScaledBitmap(  
    originalImage,    // bitmap to resize  
    view.getWidth(), // new width  
    view.getHeight(), // new height  
    true);           // bilinear filtering
```

## Hatékony újrarajzolás

- invalidate();
  - **Könnyű, kényelmes, de költséges**
- invalidate(Rect)
- invalidate(left, top, right, bottom)



# Resource optimalizálások

## PNG képek optimalizálása

- Vannak jó kis programok (☺), melyek újratömörítik a képet kisebb fájlméretbe információ veszteség nélkül.
- PNGOut optimization  
<http://advsys.net/ken/utils.htm>
- Csökkenti az APK fájlunk méretét

## Fontok optimalizálása

- Web-font optimalizálás csökkenti a fontok méretét, gyorsítja azok betöltését: <http://www.fontsquirrel.com/tools/webfont-generator>
- Singleton osztály használata a memory-leak megelőzésére

## Android Resource Tracker

- <http://code.google.com/p/android-unused-resources/>
- OpenSource, nem hivatalos Google eszköz
- Fel nem használt „resources”-ok felkutatása
- Csökkenti az APK méretét (*Google Play-re max. 50MB apk mehet*)

# Adatbázis optimalizálások

## Lekérdezések átgondolása

- Luxus a \* alkalmazása → felesleges adatmozgatás!
- Előre rendezett tárolás: megspóroljuk a lekérdezéskor a rendezést!

## Elsődleges kulcs használata

- Mindig használjunk elsődleges kulcsot! (ID) Gyorsabb egy sor elérése.

## Egy tábla sorainak száma

Hallgató kódja

```
Cursor c = adatb.rawQuery("Select * from fotabla", null);  
Log.d("NIK", "Count c: " + String.valueOf(c.getCount()));
```

```
Cursor c = adatb.rawQuery("Select count(1) from fotabla", null);  
int count = c.getInt(1);
```

## Tömeges adatbeszúrás

**Számokban**

*LG O2X, ~100E sor esetén*

Transaction nélkül: ~8 perc

Transaction-nel: ~20 mp

**~20-25x gyorsítás, [részletek itt.](#)**

```
db.beginTransaction();  
for (entry : listOfEntries) {  
    db.insert(entry);  
}  
db.setTransactionSuccessful();  
db.endTransaction();
```

# További optimalizálások

## UI folyamatosságának fenntartása

- Időigényes feladatokat háttérszálon dolgozunk fel!
- Biztosítsuk a háttér folyamat alacsonyabb prioritását!
  - Nem rontjuk le az UI szál teljesítményét
  - `imageLoaderThread.setPriority(Thread.NORM_PRIORITY-1);`

Android UI rendereléséről egy érdekes bejegyzés

## Nagy méretű képek használatának csökkentése

- Kisebb kép, kevesebb adatforgalom az adatbuszon.
- Például háttérként egy ismétlődő mintát használjunk!

## LogCat-be írás visszafogja a teljesítményt!

## Android Lint - teljesítmény javító ajánlások

- <http://tools.android.com/tips/lint>

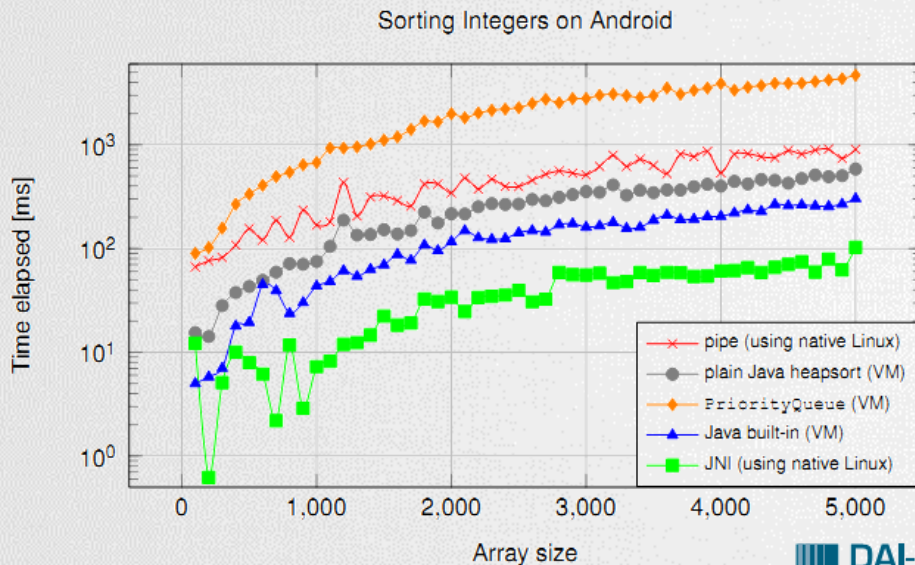
## Beépített drawable-k felhasználása, kisebb APK méret

- `android:icon="@android:drawable/ic_menu_save"`

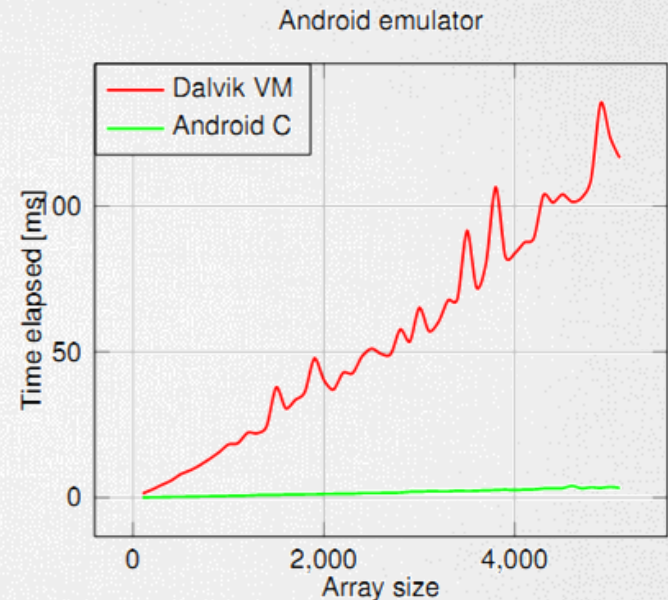
# További gyorsítások (2)

## Natív fejlesztés JNI-n keresztül

- Java kódból hívhatunk C/C++ kódot, memóriára mi ügyelünk!
- Mit jelent ez? – néhány példa



DAI-Labor  
TU Berlin



## OpenGL

- Komoly grafikát igénylő alkalmazásoknál (pl.: játék) erősen ajánlott OpenGL használata a hardveres gyorsítás miatt.
- 2D / 3D egyaránt.

De a natív kód meghívása némi többlet költséggel jár!

## Hardveres gyorsítás megjelenése a View kirajzolásánál

- <http://developer.android.com/guide/topics/graphics/hardware-accel.html>

# Teljesítményt javító eszközök

## Zipalign tool

- A forráskezelő akkor a leghatékonyabb, ha a forrás 4 byte-os egységekhez van igazítva. (32 bit) → Zipalign erre jó!
- ADT 0.9.3-as óta, projekt exportálásánál automatikus:  
Projekten jobb klikk / Andorid tools / Export Signed Application Package...
- Manuálisan:  
tools/zipalign -v 4 source.apk destination.apk

## DDMS memórafoglalás figyelése

- DDMS perspektívában lehetőségünk van a memória foglalásokat követni.