Mobile Application (Design and) Development

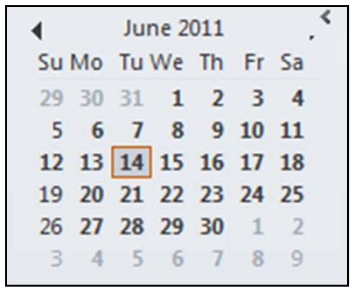
19th class

Prof. Stephen Intille
s.intille@neu.edu

Today

- Schedule code reviews / Q&A?
- Final project
  - Schedule
  - Checklist
    http://www.ccs.neu.edu/home/intille/teaching/MAD/FinalProjectChecklist.htm
- Optimization
- 4 presentations

Schedule



- Programming assignment 5 due today
- Project presentations: 22nd and 23rd
- App due: 23rd 8PM (+ available for other teams to review)
- Grade and feedback available: EOD 24th
- Contest voting: 28th – 29th
- Final download for revised grades: 29th

Optimization

- Mobiles have limited CPU and storage and battery life
  - Worth trying to be efficient
  - Battery life tracker will flag your app
- Two goals
  - Don't do work that you don't need to do.
  - Don't allocate memory if you can avoid it.

BUT...

- Optimize your DESIGN first
- Then optimize your choice of data structures and algorithms
- Then initially program for speed
- Only at a last stop optimize code

Before you start...

- ALWAYS measure; know you have a problem

- Google's recs based on Caliper microbenchmarking framework for Java
  - Google's open-source framework for writing, running and viewing the results of JavaMicrobenchmarks

Microbenchmark fallibility

- JIT compiler will likely compile your bytecode differently from real life
- Valid only for the particular hardware, OS and JRE run on; small change to any could lead to different results
- Less likely to have a cache miss
- Multithreading not considered
- Inputs may not be representative of what you get in real life

Challenge: hardware platforms

- Different versions of the VM running on different processors running at different speeds.
- Measurement on the emulator tells you very little about performance on any device.
- If you want to know how your app performs on a given device, you need to test on that device.

JIT

- Huge differences between devices with and without a JIT
  - “Best" code for a device with a JIT is not always the best code for a device without

Object creation not free

- Allocating memory is always more expensive than not allocating memory
  - 2.3 has concurrent GC
- Try to avoid creating GCs

Examples

- If you have a method returning a string, and you know that its result will always be appended to a StringBuffer anyway, change your signature and implementation so that the function does the append directly, instead of creating a short-lived temporary object.

Examples

- When extracting strings from a set of input data, try to return a substring of the original data, instead of creating a copy. You will create a new String object, but it will share the char[] with the data. (The trade-off being that if you're only using a small part of the original input, you'll be keeping it all around in memory anyway if you go this route.)

Examples

- An array of ints is a much better than an array of Integers, but this also generalizes to the fact that two parallel arrays of ints are also a **lot** more efficient than an array of (int,int) objects. The same goes for any combination of primitive types.

Examples

- If you need to implement a container that stores tuples of (Foo,Bar) objects, try to remember that two parallel Foo[] and Bar[] arrays are generally much better than a single array of custom (Foo,Bar) objects.

  (The exception to this, of course, is when you're designing an API for other code to access; in those cases, it's usually better to trade good API design for a small hit in speed. But in your own internal code, you should try and be as efficient as possible.)

Static

- If you don't need to access an object's fields, make your method static
- Invocations will be about 15%-20% faster
- Also good practice: can tell from the method signature that calling the method can't alter the object's state

Avoid internal getters/setters

- Virtual method calls are expensive, much more so than instance field lookups
- For public interface, use getters/setters
- Internally to class, access fields directly
  - i.e., don't do `i = getCount()`

- Without a JIT, 3x faster
- With a JIT, 7x faster

Use static final for constants

- No <clinit> method required
- Avoid field lookups

For-each loop syntax

```
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; }}
```

Zero slowest

One faster

Two faster for devices without JIT;
same as One otherwise

Limit use of floating point

- Floating-point is about 2x slower than integer on Android devices
  (True with and without FPU)
- No difference between float and double

Use libraries

- Might get lucky and be replaced with hand-coded assembler
  - Examples:
    - String.indexOf
    - System.arrayCopy (9x faster than hand-coded loop)

Native code

- Cost with transition
- Pain in the neck to compile for various resources
- GC issues

- Primarily useful for porting existing native codebase, not for "speeding up" parts of a Java app.

Responsiveness

- Want to avoid the Application Not Responding (ANR) dialog
  - No response to an input event within 5s
  - BroadcastReceiver fails to finish in 10s
- Danger points
  - Net access
  - Computationally intensive operations
  - File operations
  - DB operations

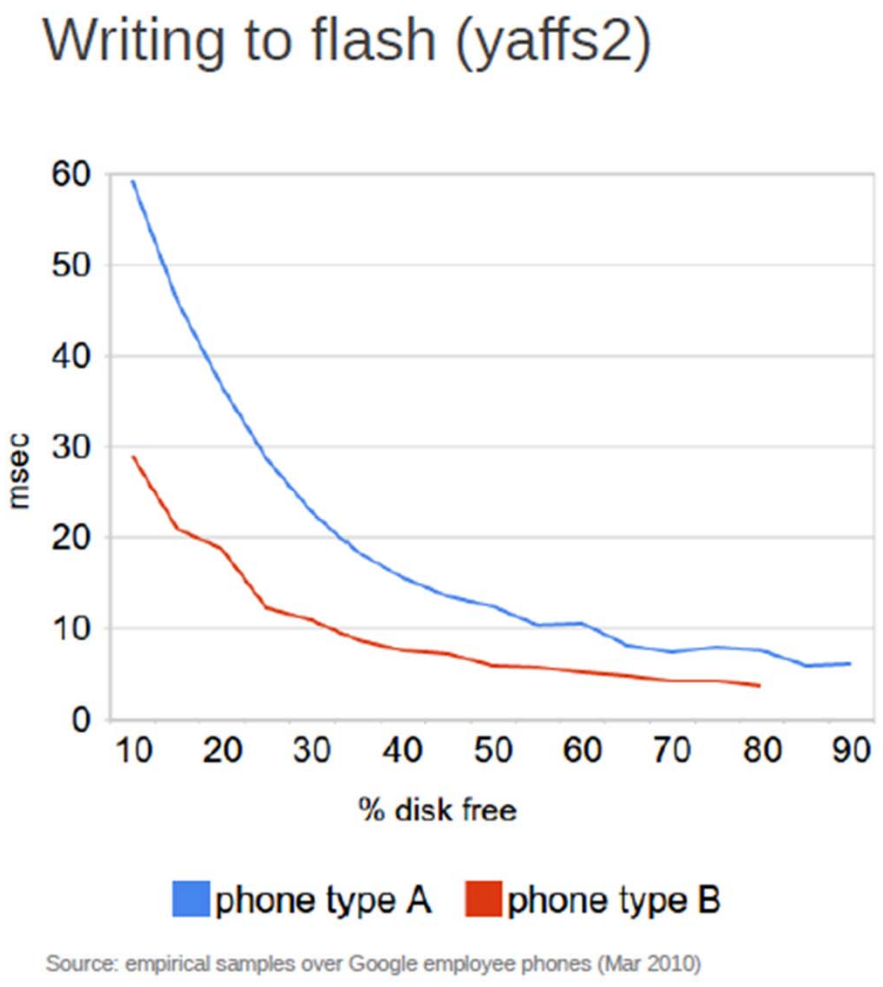
Responsiveness

- Method in the main thread should do as little work as possible
- Activities should do as little as possible to set up in key life-cycle methods such as onCreate() and onResume()
- Don't block waiting for a thread to complete ... Use the Handler or AsyncTask

What will feel slow?

- 200+ms lag
  - If your application is doing work in the background in response to user input, show that progress is being made (ProgressBar and ProgressDialog are useful for this)
  - In games, calculate moves in child thread
  - Use a splash screen during setup, or render main view and fill in info asynchronously
  - Always indicate progress being made

Watch out for writing...



- Create file, 512 byte write, delete
  - ala sqlite .journal in transaction
- Flash is … *different* than disks you're likely used to
  - read, write, *erase*, wear-leveling, GC, …
- nutshell: write performance varies a lot

What will feel broken?

- App can be snappy but feel broken with sensors

- Provide feedback on
  - Sensor state
  - What sensors know
  - Sensor noise

Responsiveness in BR

- Don’t use child threads because life of BroadcastReceiver is short
- Use a Service instead

Good citizen

- Avoid starting an Activity from an Intent Receiver
  - Spawns a new screen that will steal focus from whatever application the user is currently has running.
  - If your application has something to show the user in response to an Intent broadcast, it should do so using the Notification Manager

Testing responsiveness

- Use StrictMode to help find potentially long running operations such as network or database operations that you might accidentally be doing your main thread

Seamlessness

- Beware of popping up dialogues
  - During testing may make sense
  - But may conflict with other apps (Use Notification instead)
- App losing state because onPause and onResume not working properly

Think unpredictable

- Another app can pop up at any time (E.g. phone app)
  - Fires the onSaveInstanceState() and onPause() methods
  - Will likely result in your app being killed
- Beware if user was editing data

Share

- "Android Way" if data to expose is to use a ContentProvider

Be polite

- Don't spawn Activities except in response to user action
  - Could become a "keystroke bandit"
  - I.e., don't call startActivity from BroadcastReceivers or Services

Activities created equal

- Use multiple Activity object instances
- Don't think of Activity as single entry point to app
- Think of your application as a "federation of Activity objects"
  - Helps with history and "backstack" model
  - Makes code a bit more modular

Respect Themes

- When designing your UIs, you should try and avoid rolling your own
  - Jarring
  - Confusing
- Use a theme so you start with the same basic components
  - See Applying Styles and Themes

Respect diversity (of hardware)

- Many screen resolutions and dimensions
  - Aria: 320 x 480 pixels (1.5 ratio)
  - Droid X: 480 x 854 pixels (1.8 ratio)
- Variety of data connection speeds
  - GPRS (33kb/s in practice)
  - 3G (about 4x faster GPRS)
  - WiFi (about 120x faster GPRS)

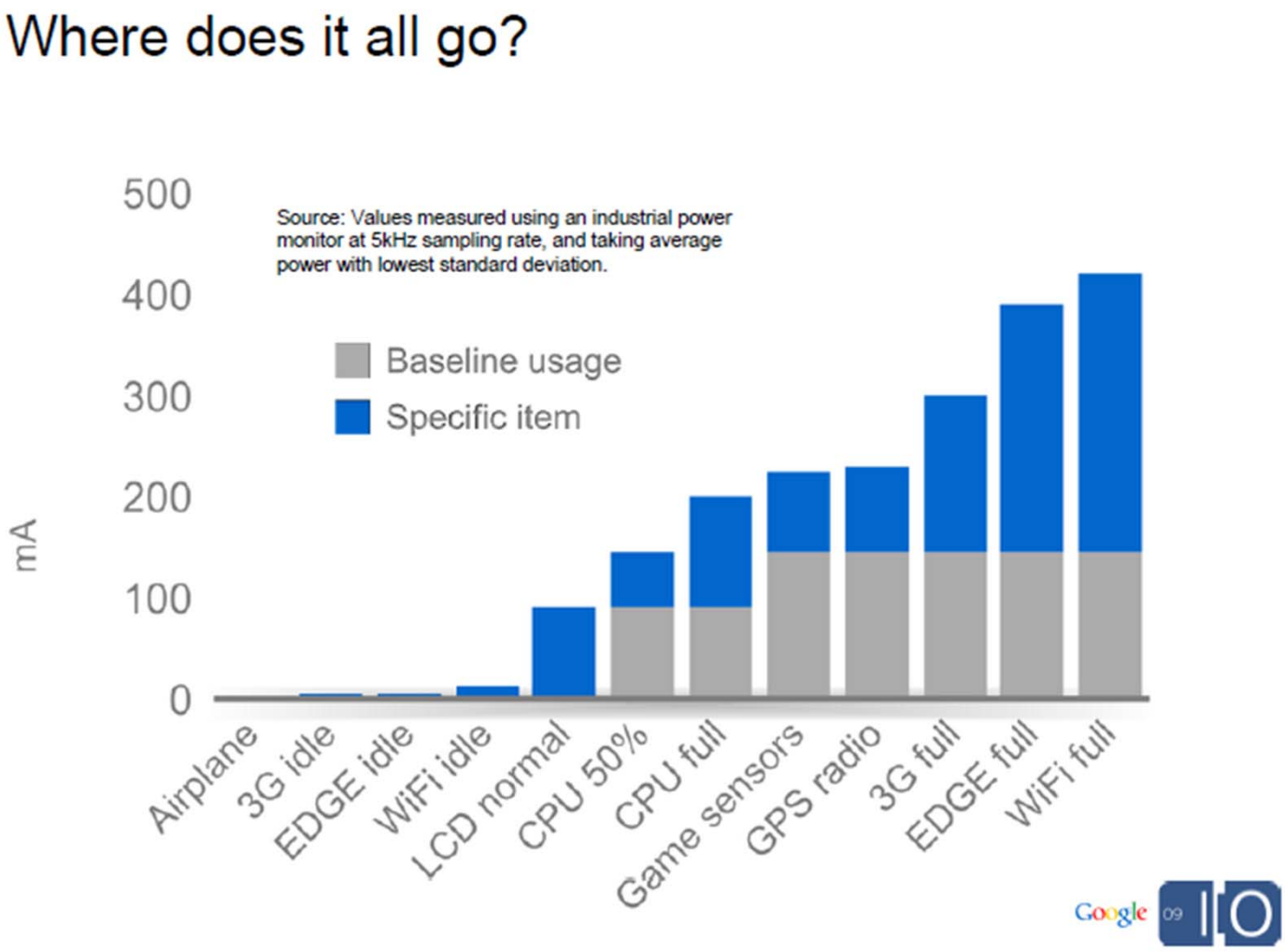
Respect diversity (of hardware)

- Tip: design for
  - Smallest screen
  - Slowest phone CPU
  - Slowest phone Internet (GPRS) (Change emulator setting)
  - Worst battery life
- MUCH Easier to scale up than down

Save battery

- Great differences
  - HTC Dream: **1150mAh**
  - HTC Magic: **1350mAh**
  - Samsung I7500: **1500mAh**
  - Asus Eee PC: **5800mAh**
- Write efficient code
- Don't repeat failed operations
  - Don't get Internet connection? Wait

Relative use of features



Real life use

- Watching YouTube: 340mA = **3.4 hours**
- Browsing 3G web: 225mA = **5 hours**
- Typical usage: 42mA average = **32 hours**
- EDGE completely idle: 5mA = **9.5 days**
- Airplane mode idle: 2mA = **24 days**

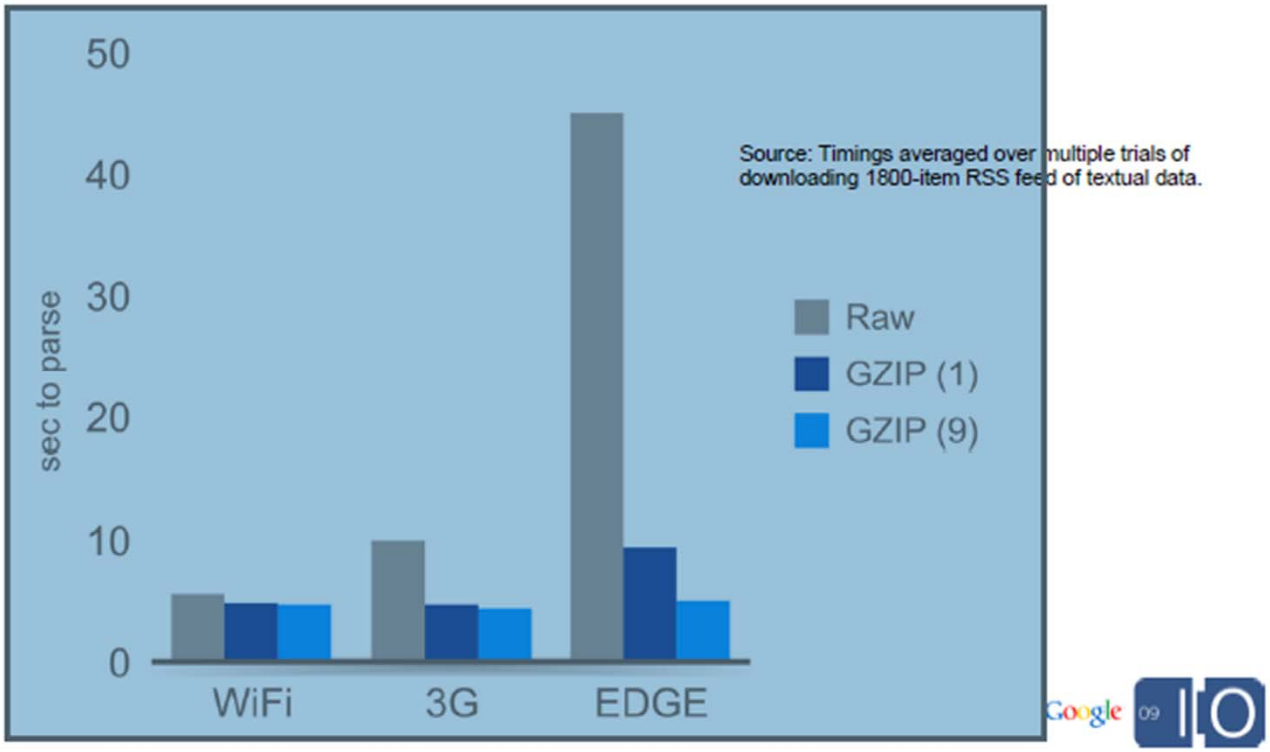
Eating the battery life

- **E.g., Waking up in the background** when the phone would otherwise be sleeping
  - App wakes up every 10 minutes to update
  - Takes about 8 seconds to update, 350mA
- Cost during a given hour:
  - 3600 seconds * 5mA = **5mAh resting**
  - 6 times * 8 sec * 350 mA = **4.6mAh updating**
- Just *one app* waking up can trigger cascade

Eating the battery life

- **Bulk data transfer** such as a 6MB song:
  - EDGE (90kbps): 300mA * 9.1 min = **45 mAh**
  - 3G (300kbps): 210mA * 2.7 min = **9.5 mAh**
  - WiFi (1Mbps): 330mA * 48 sec = **4.4 mAh**
- Moving between cells/networks
  - Radio ramps up to associate with new cell
  - BroadcastIntents fired across system
- Parsing textual data, regex without JIT

Use gzip library for text transfers



Eating the battery life

- **Use coarse network location**, it's much cheaper
  - GPS: 25 seconds * 140mA = **1mAh**
  - Network: 2 seconds * 180mA = **0.1mAh**
- 1.5 uses AGPS when network available

Eating the battery life

- GPS time-to-fix varies wildly based on environment, and desired accuracy, and might outright fail
  - Just like wake-locks, location updates can continue after onPause(), so make sure to unregister
  - If all apps unregister correctly, user can leave GPS enabled in Settings

Eating the battery life

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

Service

- Services should be short-lived; these aren't daemons
  - Each process costs 2MB and risks being killed/restarted as foreground apps need memory
  - Otherwise, keep memory usage low so you're not the first target