Android Development: Lecture Notes

Joel Ross

2017-03-26

Contents

About this Book

This book compiles lecture notes and tutorials for the **INFO 448 Mobile Development: Android** course taught at the University of Washington Information School (most recently in Spring 2017). The goal of these notes is to provide learning materials for students in the course or anyone else who wishes to learn the basics of developing Android applications.

These notes are primarily adapted from the official Android developer documentation, compiling and synthesizing those guidelines for pedagogical purposes (and the author's own interpretation/biases). Please refer to that documentation for the latest information and official guidance.

This book is currently in **alpha** status, as pure lecture notes are converted into more generic formats.

Chapter 1

Introduction

Start off with a question: *What is Android?* Why did you sign up for a course called Android Development?

It's an Operating System! That is, it's software that connects hardware to software and provides general services. More than that, it's a *mobile specific* operating system.

"Android" also refers to "platform" (e.g., devices that use the OS) and the ecosystem that surrounds it. This includes the device manufacturers who use the platform, and the applications that can be built and run on this platform.

1.1 Android History

Some brief history:

- 2003: founded by "Android Inc" to build mobile OS operating system (similar to what Symbian (Nokia) was doing)
- 2005: acquired by Google, who was looking to get into mobile
- 2007: Google announces Open Handset Alliance, group of tech companies working together to develop "open standards"
  - Google; HTC, Samsung, Sony; T-Mobile, Sprint NTT DoCoMo; Broadcom, Nvidia, etc. Now 84 companies.
  - Note this is the same year the first iPhone came out!
- 2008: First Android device (HTC Dream / T-Mobile G1)
  - 528Mhz ARM chip; 256MB memory; 320x480 resolution capacitive touch; slide-out keyboard! A fun little device
- 2010: First Nexus device: Google-developed "flagship" devices
  - Nexus One: 1Gz Scorpion; 512MB memory; .37" 480x800 AMOLED capacitive touch

    - Comparison: iPhone 6S+ (1.85Ghz dual core ARMv8, 2GB RAM, 5.5" @ 1920x1080)
- 2014: Android One (low-cost for developing countries)
- 2015: Project Brillo, aiming for embedded devices

Android is incredibly popular! (see e.g., here, here, here)

- There are some questions about what is counted… but what we care about is that there are *a lot* of Android devices out there! More than that, there are a lot of **different** devices!

For version history, reference the Wiki chart - Different "versions" are named after desserts. But from developer perspective, care about API version (e.g., what set of *interfaces* are available). - Interactive version: http://www.android.com/history/ - Usage breakdown at: http://developer.android.com/about/dashboards/index.html

Updates to devices historically through purchasing new devices (every 18m on average in US). Otherwise updates–including security updates–come through carriers. This is a problem from a consumer perspective! But Google working to change that; moving some services out of OS into separate "App" (Google Play Services). - Android OS is "open source"; latest version at https://source.android.com/. Worth actually digging around in this sometime

Would be remiss if didn't mention some of the legal issues surrounding Android.

- Biggest is **Oracle v Google**: basics is that Oracle says the *Java API* is copyright, so because Google uses that API is Android, Google is violating the copyright
    - Claim: the method signatures themselves (and how they work) are protected!
    - CA court decided for Google in 2012 (can't copyright an API!); reversed by Federal Circuit in 2014. Supreme Court refused to hear in 2015.
    - https://www.eff.org/cases/oracle-v-google for more Also https://arstechnica.com/series/series-oracle-v-google/
    - Interesting development: latest version of Android (Nougat) uses the OpenJDK implementation of Java, instead of Google own see here. Shouldn't impact us, but keep an eye out for potential differences between Android and Java SE.
- Others as well:
    - Apple v Samsung: "You infringed our intellectual property!" (Supreme court sides with Samsung)
    - FairSearch v Google: "You're using predatory pricing!"

Take away: Android is a growing, evolving platform that is embedded in and affecting the social infrastructures around information technology.

1.2 Android Architecture and Code

Let's consider the basic architecture of the Android platform (More details: https://source.android.com/devices/)

- Runs on a linux kernel (for interacting with memory, processor, etc)
- On top of that is the hardware abstraction layer: interface to drivers that let us access the hardware (camera, storage, wifi, etc)
  - These drivers are generally written in C
- But on top of that is the Runtime and Android Framework, which provides the Java language that we all know and love!
- (We'll be developing applications that interact with the Framework, which runs on Java).

There are two languages we'll be working with in this class:

1. **Java**. Android code (the logic, control, data stuff) is written in Java.
   - This will feel a lot like every other Java Program: you write classes, define methods, instantiate objects, and call methods on those objects
   - But we're working within a **framework**: a set of code *already exists* to call specific methods: we'll just fill in what those methods do to run our code
     - More Angular (framework) than jQuery (library)
   - Important: This course expects you to have "journeyman"-level skills in Java (apprenticeship done, not yet master). We'll be using a number of intermediate concepts (like generics and inheritance) without much fanfare or explanation. And so you should be comfortable with that. If you're not... this class is going to be a challenge. Consider yourselves warned.
     - We will go over some of the topics that people reported being less familiar with tomorrow in lab... but that will be most our Java review. I'm assuming you're already comfortable with the language so that we can focus on the framework.
     - *By the way*: for reviewing; rather do a "work through a tutorial/explanation" or have me lecture/review topics? (e.g., do you want to drive or do you want me to?)
   - Any questions/concerns/issues about that
2. **XML**. Android interfaces and resources are specified in XML (eXtensible Markup Language). So we'll be doing a lot of XML writing.
   - XML is just like HTML, but you get to make up your own tags (though we're going to use the ones that Android made up).
   - If you've never worked with HTML... you'll pick it up fast (it's not complex).

If you think of web programming: the XML is going to contain what we would put in our HTML/CSS, and the Java will contain what would go in our JavaScript.

Building Apps: So we write code in Java & XML. How does that get run on the phone's hardware?

- Pre-Lollipop (5.0): used Dalvik, a virtual machine (similar to the JVM from Java SE)
  - Java code –compile–> JVM bytecode –translated–> DVM bytecode
    - stored in DEX or ODEX files ("[optimized] Dalvik executables")
    - process is called "dexing" (so code is "dexed")
  - For CS people: Dalvik register-based architecture (not stack-based!)
  - Dalvik Includes JIT compilation to native code (like the Java HotSpot)
    - *Why would "native code" be faster?* Because no translation step is needed to talk to the actual hardware (the OS)
- Post-Lollipop (5.0): uses Android Runtime (ART)
  - compile into native code on installation! ("Ahead of Time" AOT)
  - accepts DEX bytecode for backwards compatibility
  - faster execution, but longer install time

Android applications are packaged into APK files (basically zip files)

- Which are then "side-loaded" or cryptographically signed to be uploaded to the App Store.
  - these are the "executable" versions of your program!
- Note: application frameworks are pre-DEXed (compiled) on device; actually compiling against empty stubs!
  - But any other libraries you include are copied into the app code.

So when building an App...

1. Generate Java source files (e.g., from resource files, which are written XML used to generate Java code)
2. Compile Java code into JVM bytecode
3. "dex" the JVM bytecode into Dalvik bytecode
4. Pack in assets and graphics into an APK
5. Cryptographically sign the APK file to verify it

There are a lot of steps here, but there are tools that take care of it for us. We'll just write Java and XML code and run a "build" script to do all of the steps.

1.3 Development Tools

Since we're writing code for a virtual machine anyway, we can build Android apps on any computer OS (unlike some other mobile OS)

- Physical devices are the best–you'll need USB cable to be able to wire your device into your computer.
  - Any device should be fine; don't need cell service (just WiFi mostly)

  - If you don't normally use an Android phone, play around with it a bit to get use to the interaction language. E.g., how to click/swipe/drag/long-click things.
  - Turn on developer options!
- Also can use the Emulator (a "virtual" android device)
  - Represents a generic device with hardware you can specify… but has some limitations!
  - But the emulator not great on Windows; recommend you use Mac or a physical device
  - Make sure to use the HAXM (Intel acceleration manager!)
  - I'll be doing this for lecture demos (will keep me running slower than you!)

1.3.1 Software

The software used to develop Android applications include:

- Java 7 **SDK**
- Gradle or Apache ANT
  - These are automated build tools–in effect, they let you specify a command that will do a bunch of steps (e.g., compilation, moving, etc) at once. These are how we make the "build script" to do those 5 building steps from earlier.
  - ANT is the old way, but Gradle is the new hotness
  - You'll be using Gradle for first homework (Java review), and we'll be poking at it periodically through the course.
    * We'll be doing minor tweaks to build files, not learning the entirety of the gradle build system (though as you get into professional development systems you'll want to check them out)
- Finally, the Android Studio & SDK.
  - This is our IDE and build system (everything else goes to support this).
  - Make sure the SDK command-line tools are installed!
    * put `tools` and `platform-tools` on the `PATH`
    * run `adb` to check
- SDK Tools include:
  - *deprecated* `android`: does SDK/AVD (virtual device) work. Basically IDE commands, but from command-line
  - `emulator` runs the emulator
  - `adb` "Android Device Bridge"; connection between your computer and the device (physical *or* virtual). Used for console output!
  - all of these are built into IDE, but command-line is a good fall-back!

1.4 Hello World

In time remaining, lets get an App up and running and see what we actually will be working with! You will need to have Android Studio installed for this to work.

1. Launch Android Studio if you have it (may take a few minutes to open)
2. Start a new project!
   - use uwnetid in domain
   - note project location! Desktop is good for now
   - Target: this is the "minimum" SDK you support. Going to target Ice Cream Sandwich (4.0.3) for most this class. This is the earliest version we'll support.
     - Note that this is different than the "target SDK", which is the version you tested on! We'll adjust this in a moment, and will test on API 21 (Lollipop).
     - If you're testing on a physical device running older version, you can target that. But we'll grade at the 5.0 target.
3. Select an Empty Activity
   - Activities are "Screens" in your application (things the user can do). We'll talk more about this on Wednesday.
4. And boom, have an Android app! Aren't frameworks lovely?

1.4.1 The Emulator

We can run our app by clicking the "Play" or "Run" button. We will need to define a device, so let's make an emulator!

- Nexus 5 is a good choice for supporting "older" devices. Pixel is also reasonable.
  - Use Lollipop, Google APIs, x86!
  - Snapshot to speed up loading
- Specify **hardware keyboard**
- Want to go in and edit it (`Tools > Android > AVD Manager`) so it accepts keyboard input!

Can slide to unlock, and there is our app!

1.4.2 Project Contents

So what does our app look like in code? What do we have?

Note that Android Studio by default shows the "Android" view, which organizes files thematically. If you got the "Project" view you can see what the actual file system looks like.

- `app/` folder contains our application

  - `manifests/` contains the **Android Manifest** files, which is sort of like a "config" file for the app
  - `java/` contains the Java source code for your project
    * And we can find the `MyActivity` file in here
  - `res/` contains resource files used in the app. These are where we're going to put layout/appearance information
- Also have the `Gradle` scripts. There are a lot of these:
  - `build.gradle`: Top-level Gradle build; project-level (for building!)
  - `app/build.gradle`: Gradle build specific to the app **use this one to customize project!**
    * we can change the target SDK here!
  - `proguard-rules.pro`: config for release version (minimization, obfuscation, etc).
  - `gradle.properties`: Gradle-specific build settings, shared
  - `local.properties`: settings local to this machine only
  - `settings.gradle`: Gradle-specific build settings, shared
  - ANT would give:
    * `build.xml`: Ant build script integrated with Android SDK
    * `build.properties`: settings used for build across all machines
    * `local.properties`: settings local to this machine only
    * We're using Gradle, but be aware of ANT stuff for legacy purposes
- `res` has resource files. These are **XML** files that specify details of the app–such as layout.
  - `res/drawable/` : graphics (PNG, JPEG, etc)
  - `res/layout/` : UI XML layout files
  - `res/mipmap/` : launcher icon files
    * fun fact: MIP comes from "multum in parvo", latin for "much in little". Map cause image mapping
  - `res/values/` : general constants
  - See also: http://developer.android.com/guide/topics/resources/available-resources.html
    * We'll talk about these more next week!

Let's look at what the application does. We'll revisit this on Wednesday, but this will let us start seeing how the framework is structured.

- We start with the **Activity** Java source
  - We extend `Activity` (actually a subclass that supports Material Design stuff), making our own customizations.
- Override `onCreate()` method that is called by the framework when the Activity starts (more on lifecycle tomorrow).
  - Call super, and then `setContentView` to specify what the content (appearance) of our activity is.
  - Passing in a value from something called `R`. What *type* of thing is this? (It's a class!)
  - `R` is a class that is **generated at compile time** and contains con-

stants that are defined by the "resource" files!
    * Those files are converted into Java variables, which we access through the `R` class.

- `R.layout` refers to the "layouts" resource, so can go there.
  - Can open then up in "design" view. This lets you use GUIs to lay out your application (like a powerpoint slide). But frowned upon–much cleaner/nicer to write it out in code.
    * Same difference between writing your own HTML and using FrontPage or DreamWeaver or Wix. Legit, but less nice.
  - We can see the XML: tags, attributes, values. Tags nested inside one another.
  - Defines a layout, and inside that is a TextView (view of some of text), which has a value–text!
    * Can change that, and re-run the app!
  - If time: can also define this in `values/strings` (e.g., as a constant, refer to as `@string/message`)
- We'll talk about the layout and using these resources *A LOT* more next week.
- If time: we can also set an icon! (`File > New > Image Asset`)

Chapter 2

Activities and Logging

Now that we've gotten a feel for the pattern/coding style of creating graphical applications, let's see how they play out in Android! - You'll need to create a new `Android` application to play with, with a single **Empty** Activity (e.g., `MainActivity`). In the future I'll have starter code for you to work from, but since we're starting from scratch it makes sense to start with something empty. Plus: it's good practice! - We can also take a **2-minute** stretch break while you get everything up and running.

2.1 What is an Activity? [10min]

What is an Activity? According to Google:

> An Activity is an application component that provides a screen with which users can interact in order to do something

You can think of an Activity as a single Screen in your app. The equivalent of a "Window" in a GUI system (or a `JFrame` in our Swing app). - Note that Activities don't **need** to be full screens: they can also be floating modal windows, embedded inside other Activities (like half a screen), etc. But we'll start by thinking of them as full screens. - In many ways, an Activity is a "bookkeeping mechanism": a place to hold *state* and *data*, and tell to Android what to show on the display. - It functions much like a Controller (in Model-View-Controller sense) in that regard! - We can have lots of Activities (screens) in an Application, and they are loosely connected so we can easily move between them.

Also to note from the documentation:

> An activity is a single, focused thing that the user can do.

which implies a design suggestion: Activities (screens) break up your App into "tasks": each one can represent what a user is doing at once! If the user does something else, that should be a different Activity and so probably a different screen.

2.1.1 Making Activities

How do we use them? We create our own activities by subclassing the provided `Activity` class. - This is **inheritance**: we're making a specialized type of `Activity`, similar to extending `JFrame` in Swing apps - All the methods that control how the OS interacts with Activities are provided for us.

If you look at the default Empty activity, it actually subclasses `AppCompatActivity`, which is a already specialized kind of Activity that provides an `ActionBar` (the toolbar at the top). If we change the class to just extend `Activity`, that bar disappears. - You'd need to import the `Activity` class! The keyboard shortcut inAndroid Studio is `alt+return`, or you can do it by hand (look up the package)!

There are a pile of other built-in Activity subclasses that we could subclass instead. We'll mention them as they become relevant. - Many on the books have been deprecated in favor of **Fragments**, which are sort of like "subactivities" that get nested in larger Activities. We'll talk about Fragments more in a couple weeks, once we've gotten the basics down.

Other important point to note: does this activity have a **constructor** that we call? (No!) - We never write code that **instantiates** our Activity. There is no `main` method. Activities are created and managed by the Android operating system!

2.2 The Activity Lifecycle [10min]

So if we never call the constructor or main, how do we start doing stuff?

Activities have an *incredibly* well-defined lifecycle—that is, a series of **events** that occur during usage (e.g., when created, stopped, etc). - When each of these events occur, Android executes a **callback method**, similar to how you called `actionPerformed` to react to a "button press" event. We can override these methods in order to do special actions (read: our own code) when these events occur.

What is the lifecycle?

(Or an alternative, simplified diagram here).

There are 7 "events" that occur in the Activity Lifecycle, which are designated by the callback function that they execute: - `onCreate()`: when **first** created/instantiated. This is where you initialize UI stuff (e.g., specify the layout

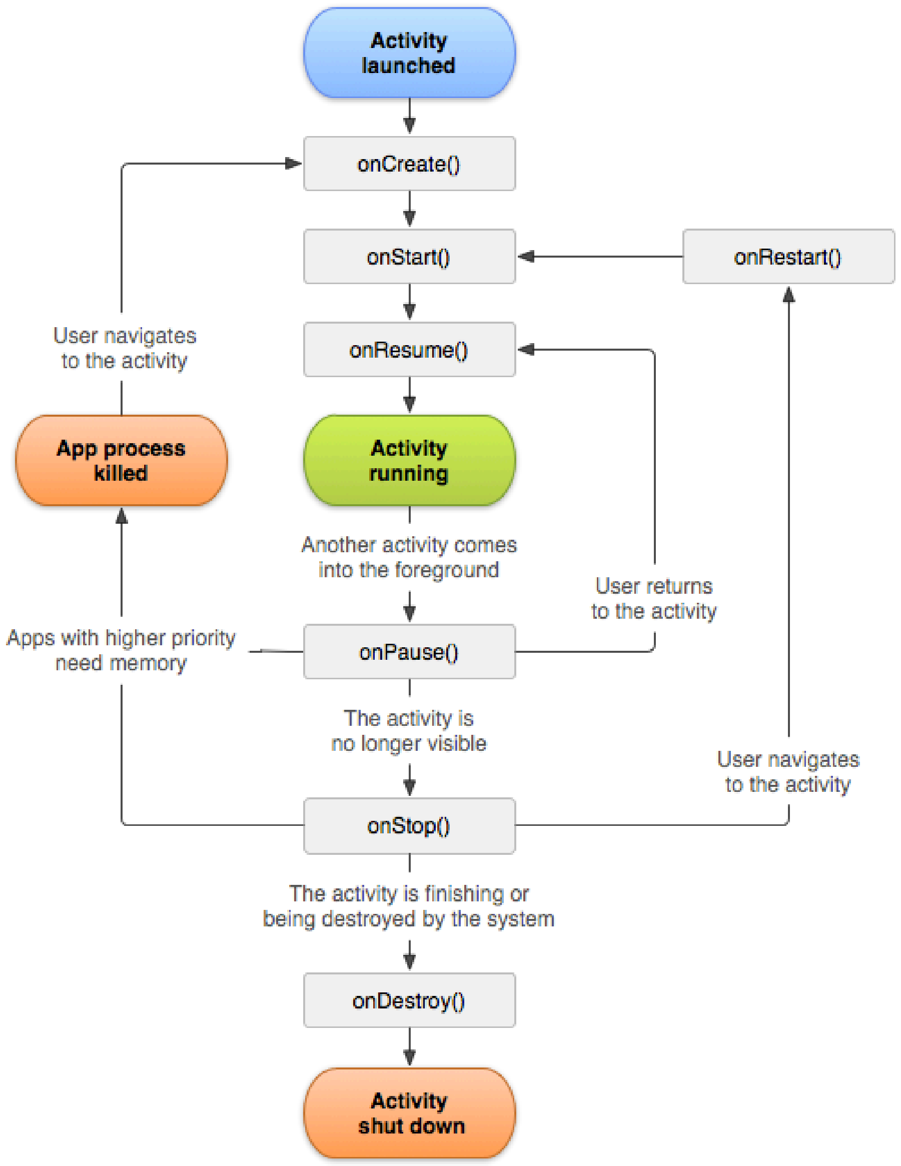

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.1: lifecycle state chart

to use) - `onStart()`: called just before the Activity becomes **visible** to the user. - Whats the difference between this and `onCreate`? `onStart` can be called more than once (e.g., if you leave the Activity and come back). - `onResume()`: just before **user interaction** starts–activity is ready to go! A little bit like when that Activity "has focus" - difference between `onStart()`? When start is called we're visible, but resume is called when interaction is ready. - We can be visible but not interacting (like if there is a modal in front of us!) - `onPause()`: when the system is about to start another activity (so about to lose focus). This is the "mirror" of `onResume()`. *The activity stays visible.* - This is where we tend to store (temporary) unsaved changes (like email drafts), stop animations, etc, since the activity might be on its way out. - `onStop()`: the activity is no longer visible. (e.g., another activity took over, but also because might be destroyed). A mirror of `onStart()`` - This is where we would want to persist any state information (e.g., save their game). -`onRestart(): called when the app is coming back from a "stopped" state -`onDestroy(): activity is about to be closed. This can happen because the user ended the application, **or** (and this is important!) because the OS is trying to save memory and so kills your app (if it hasn't been use in a while). - can do more final cleanup, but better to have cleanup in`onPause()`and`onStop()‘

Note that apps may not need to use all of these methods: for example, if there is no difference between starting from scratch and resuming from stop, then you don't need an onRestart (since onStart goes in the middle). Similarly onStart may not be needed if you just use onCreate and onResume; but these lifecycles allow for more granularity and the ability to avoid duplicate code.

2.2.1 Overriding the methods

Let's look at overriding these methods to see them in action!

We already have `onCreate()` overridden for us, since that's where the layout is specified (we'll cover how soon, I promise!) - Notice it takes a `Bundle` as a parameter. A `Bundle` is an object that stores key-value pairs, like a super-simple `HashMap` (or an Object in JavaScript). Bundles can only hold basic types (numbers, Strings) and so are used for temporarily "bunding" *small* amounts of information. - This `Bundle` in particular in stored information about the Activity's current state (e.g., what text they may have typed into a search box), so that if the App gets killed it can be restarted in the same state and the user won't notice that it was ever lost! - It stores current layout information in it by default (if Views have ids), and since its a Map (key-value pairs) you can store other data in it as well. - calls `onSaveInstanceState()` for each View, and those tend to save important bits already - see Saving Activity State for details. - Also note that we call `super.onCreate()`. **ALWAYS CALL UP THE INHERITANCE CHAIN**, so that you can do system-level stuff!

So we can add the others: how about `onStart()`? (see Implementing Lifecycle Callbacks for template).

2.3 Logging & ADB [10min]

But how can we know if the lifecycle events are getting called? - `System.out.println()`? we don't actually have a terminal! More specifically, our device (which is where the application is running) doesn't have access to `stdout` - Aside: we can get it with `adb shell stop; adb shell setprop log.redirect-stdio true; adb shell start` - Instead, Android provides a Logging system that we can use to write out debugging information, and which is automatically accessible over the `adb` (Android Debugging Bridge). - Logging can be filtered, categorized, sorted, etc. - Can be disabled in production builds, but often isn't :p - To perform logging, we'll use the `android.util.Log` class. This class includes a number of `static` methods, which all basically wrap around `println` to print to the device's log file, which is then accessible through the `adb`. - The device's log file is stored… sort of. It's a 16k file, but is shared across the *entire* system. So fills up fast. Hence filtering/searching becomes important, and you tend to watch it/debug in real time! - Remember to import the `Log`] class!

2.3.1 Log Methods

Methods correspond to different level of priority (importance) of the messages. From low to high: - `Log.v()`: VERBOSE output. The most detail. Everyday stuff. Often our go-to level. - Ideally, you should only compile into an application during development! - `Log.d()`: DEBUG output. lower-level, but a bit less detail (e.g., code-level) - Debug logs *can* be compiled but stripped at runtime using the [`BuildConfig`] generated class, which can be customized through Gradle - `Log.i()`: INFO output. High level info (e.g., user-level) - `Log.w()`: WARN output. Warnings - `Log.e()`: ERROR output. Errors - Also look at the API… `Log.wtf()`!

These are used to help "filter out the noise". So you can look just at errors, at errors and warnings, at err warn info… all the way down to *everything* with verbose. - A HUGE amount of information is logged, so filtering really helps!

Each `Log` method takes two `Strings` as parameters. The second is the message to print. The first is a "tag"–a String that's prepended to the output which you can search and filter on. - This is is usually the App or Class name (e.g., "AndroidDemo", "MainActivity") - A common practice is to declare a `TAG` constant you can use throughout the class:

```java
private static final String TAG = "MainActivity";
```

2.3.2 Logcat

You can view the logs via `adb` (the debugging bridge) and a service called `Logcat` (from "log" and "conCATenation", since it concats the logs). - The easiest way to check Logcat is to use Android Studio. The Logcat browser panel is usually found at the bottom of the screen after you launch an application. It "tails" the log, showing the latest output as it appears - You can use the dropdown box to filter by priority, and the search box to search (e.g., by tag if you want). - Android Studio also lets you filter to only show the current application, which is hugely awesome. - Note that you may see a lot of Logs that you didn't produce, including possibly Warnings (e.g., I see a lot of stuff about how OpenGL connects to the graphics card). *This is normal*! - It is also possible to view Logcat through the command-line using `adb`, and includes complex filtering arguments. See Reading and Writing Logs for more details.

Demo!

Let's log out some of the lifecycle! - implement `onResume()`. Note the wonders of tab completion! Have it log out at INFO level. Hit `menu` to send to background. - `onStop()` and switch out of the app. - `onDestroy()` can easily be called if you set phone to "Don't Keep Activities" (at bottom of developer settings) - Something else to test: Cause the app to throw a runtime `Exception` in one of the handlers. For example, you could make a new local array and try to access an item out of bounds. Or just `throw new RuntimeException()` (which is slightly less interesting). - *Can you see the **Stack Trace** in the logs?*

2.4 Basic Events [10min]

Now that can "output" some text (via log), let's add some "input" via an interface element: a Button we can click. - In **`res/layouts/activity_main.xml`** (the original Activity's layout), add the following code:

```
<Button
    android:id="@+id/my_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Start Activity"
    />
```

This should go inside the `<RelativeLayout>` element, **replace** the `<TextView>` element. - This defines a (anyone?) Button. The `android:text` attribute gives the text that is on the button. - We'll talk in a lot more detail about how exactly this XML works (and what's the deal with the id, layout_width/height) next week, but you should be able to make a pretty good educated guess based on the names. - Actually, to make the button not overlap, change the `<RelativeLayout>` to a `<LinearLayout>`. More on layout tomorrow! - Defining this in XML is basically the same process as creating the `JButton` and adding it to the `JFrame` in Java!

Now we have a button, but we want to click on it. So we need to register a "listener" for it (in Java).

```
Button button = (Button)findViewById(R.id.my_button);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Perform action on click
    }
});
```

- First we need to get access to a variable that represents that Button defined in the XML. The `findViewById` method "finds" the appropriate XML element with the given `id`. We'l talk about why we wrote the id as `R.id.my_button` tomorrow. Note that this method returns a `View`, so we want to cast into the more specific `Button`.
- Now we can assign a listener to that button by registering through the `.setOnClickListener()`, passing in an **anonymous class**
- Again, tab-completion is our friend!
- This is *just like* what we did with Swing!
- Finally, we can fill in the method to have it log out something when clicked.

This is an example of an Input Control. We'll talk about these in more detail next week.

2.5 Multiple Activities [15mins]

The whole point of considering the Activity Lifecycle is because Android applications can have multiple activities and interact with multiple other applications. So let's talk briefly about how we could have an app use multiple Activities (and so get a sense for how this lifecycle may affect us)

We can go ahead and create a New Activity through Android Student by using `File > New > Activity`. We could also just add a new .java file with the Activity class in it - Make a new Empty activity `SecondActivity` - Note that this Activity also gets a resource XML layout - You should edit the `<TextView>` element so this one can have a message as well. - *ALSO NOTE*: For every activity we make, it gets added to the **Manifest** file. This is sort of like the "Table of contents" for our application, telling the operating system information about what our app looks like so it can interact with it. - another `<activity>` element in the `<application>` element. We can also see the first Activity; we'll talk about its child elements later (a lot of that) - We can also add `android:label` attributes to these `<activity>` elements to give them nicer names.

In Android, we don't start new Activities by instantiating them (remember,

we never instantiate Activities!). Instead, we send the operating system a message requesting that the Activity do something (i.e., start). These messages are called **Intents**. - Intents are **messages** used to communicate between app components like Activities. This allows them to communicate, even though they don't have references to each other (so we can't just call a method on them). - I don't have a good justification for the name, other than it is an "intention" to do something that you announce to the OS - You can think of Intents as like envelopes: they are addressed to a particular target (e.g., another Activity–or more properly a `Context`), and contain a brief message about what to do.

`Intents` are something we *can* instantiate, so let's do that in our event handler! There are lots of different constructors, but the one we'll start with is:

```
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
```

- The first parameter refers to the current **Context**, which is a superclass of `Activity`. Context is an **abstract class** that acts as a reference for information about the current running environment; it represents environmental data (stuff like "what OS is running? Is there a keyboard plugged in?"). You can *almost* think of it as representing the "Application", though it's broader than that (since `Application` is actually a subclass of `Context`!)
- The context is *used* to do "application-level" actions: mostly working with resources (accessing/loading), but also communicating between Activities like we're doing now. Effectively, it lets us refer to the state in which we are running: the "context" for our code (e.g., "where is this occurring?"). It's a kind of *reflection* or meta-programming, in a way.
- There are a couple of different kinds of Contexts we might use:
  - The Application context (e.g., the `Application`) references the state of the entire application. It's basically the Java object that is built out of the Manifest (and so contains that level of information)
  - The Activity context (e.g., the `Activity`) that references the state of that activity. Again, this would be the `<activity>` tags from the Manifest.

  Each of these `Context` objects exist for the life of their respective component: that is, an `Activity` Context is around as long as the Activity exists (disappearing after `onDestroy`), where as `Application` Contexts survive as long as the application does. We'll almost always use the `Activity` context, as it's safer and less likely to cause memory leaks.
- The second parameter is the class we want to send the Intent to (the `.class` property fetches a reference to the class type; this is metaprogramming!)

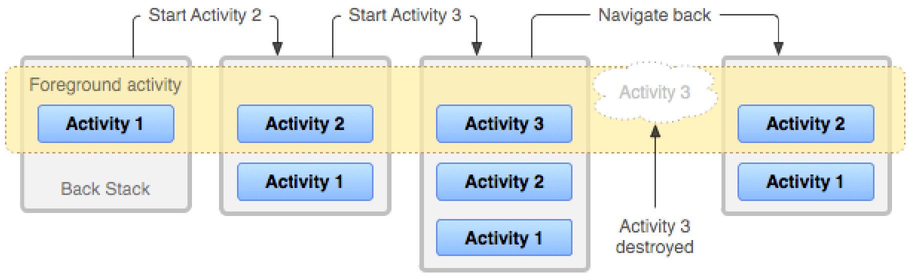
And now that we built the intent, we can use it to start an activity using the `startActivity` method (inherited from `Activity`), passing it the `Intent`!

- Voila! we can now start a second activity, and see how that impacts our Lifecycle calls (e.g., with visibility, etc). - And we can use the **back** button to go backwards!

There are actually a couple of different kinds of `Intents` (this is an **Explicit Intent**, because it is explicit about what Activity it's sent to), and a lot more we can do with them. We'll dive into Intents in more detail later; for now we're going to focus on mostly Single Activities. - e.g., if you look back at the Manifest, you can see that the MainActivity has an `<intent-filter>` that allows it to receive particular kinds of Intents–including ones to use it when launching the App!

2.6 Back & Tasks [5-10min]

So we can have lots of Activities (even across multiple apps!) running and move between them. How exactly is that "Back" button keeping track of where to go to? - Do you know what kind of data structure is associated with "back" or "undo"? A **stack**! - Every time you start a new Activity, Android creates it and puts it on the top of a stack. Then when you hit the back button, that activity is popped off the stack and you're taken to the new head.



However, you might have different "sequences" of actions you're working on: maybe you start writing an email, and then go to check your Twitter timeline through a different set of Activities. Android breaks up these sequences into groups called **Tasks**. A task is a collection of activities arranged in a Stack; and there can be multiple tasks in the background. - Tasks usually start from the Home Screen. E.g., when you launch an Application, that starts a new Task. - When you go back to home screen, that Task is moved to the background, so the "back" button won't let you navigate that Stack. - Thinking of them like different tabs/browsers and webpages is a pretty good analogy

Important caveat: Tasks are distinct from one another, so you can have different copies of the same Activity on multiple stacks (e.g., the Camera activity could be part of both Facebook and Twitter app Tasks if you are on a selfie binge) - Though it is possible to modify this, see Managing Tasks

Demo: switch to another app, then back to ours

2.6.1 Bonus: Up Navigation

We can make this "back" navigation a little more intuitive for users by providing explicit up navigation), rather than just forcing them to go back through Activities in the order they viewed them (e.g., if you're swiping through emails and want to go back to the home list). We just need a little bit of configuration for our Activities: - In the Java code, we want to add more functionality to the `ActionBar`. *Think*: what event handler should it be put in? `java getSupportActionBar().setHomeButtonEnabled(true);` - Then In the **Manifest**, add a `android:parentActivityName` attribute to the `SecondActivity`, with a value to set the full class name (including package **and** appname!) of your Main Activity. This will let you be able to use the "back" visual elements (e.g., of the ActionBar) to move back to the "parent" activity. See Up Navigation for details. - This is only supported for API 16+; since our min SDK is 15, we can include backwards support with the following child XML element: `xml <meta-data android:name="android.support.PARENT_ACTIVITY" android:value="{parent.activity.package.goes.here}" />`

2.7 Toasts [5min]

Logging is fantastic and one of the the best techniques we have for debugging, both in how Activities are being used or for any kind of bug (also RuntimeExceptions) - It harkens back to println debugging, which is totes legit. Android Studio does have a debugger if you're comfortable with those (can be handy!)

However, sometimes you want to check some output/interaction without Logging it. You just want to see some feedback while the app is running! Or you want to give a quick message to the user. Android provides a number of different classes for doing visual notifications, including alert-style and customizable Dialogs, which we'll talk about in a few weeks.

But a simple, quick way of giving some short visual feedback is to use what is called a Toast. This is a tiny little text box that pops up at the bottom of the screen for a moment. - Toast because it pops up :p

Toasts a pretty simple to implement, as with the following example (from the docs):

```
Context context = this; //getApplicationContext(); //for not disappearing if app clo
String text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

- But since `this` Activity *is* a context, and we can just use the Toast anonymous, we can shorten this to a one-liner:

```
Toast.makeText(this, "Hello world", Toast.LENGTH_SHORT).show();
```

- Boom, a quick visual alert method we can use for proof-of-concept stuff!
- Note that this uses a static `makeToast()` method, rather than a constructor. This is an example of a Factory method–a design pattern we'll see a lot.
- Toasts are intended to be ways to interact with the user (e.g., giving them quick feedback), but can possibly be useful for testing too! Though in the end, Logcat is going to be your best bet for debugging.

Chapter 3

Resources and Layouts

Today we're going to talk about **Resources** and how to use them to define **Layouts**. We talked about `Activities` last week which are the Java portion of an app, now we're going to talk about the XML.

3.1 Resources

Resources can be found in the `res` folder, and represent elements or data that are "external" to the code. You can think of them as "media content": often images, but also things like text clippings (or short String constants!). Textual resources are usually defined in XML files. - Why there? Because resources represent elements (e.g., content) that is *separate* from the code (the behavior of the app). This is about **Separation of Concerns** - By defining them in XML, they can be developed (worked on) *without* coding tools (e.g., with systems like the "layout design" tab). So you could have a Graphic Designer create these resources, which can then be integrated into the code without the Designer needing to do a lick of Java. - Similarly, you can choose what resources to include *dynamically*. You can choose to show different images based on device screen resolution. Or pick different Strings based on the language of the device (Internationalization!)–the behavior of the app is the same, but the "content" is different! - Web terms: same JavaScript, different HTML! - Architecture terms: keep the Model separate from the View!

What should be a resource? In general: - Layouts should **always** be resources - UI controls (buttons, etc) should mostly be defined as resources (part of layouts) - Any graphic images (drawables) should be resources - Any *user-facing* strings should be resources - Styles are should be resources

As we peeked at last week, there are a number of different resource types, many of which you can see in the `res` folder of a default Android project -

`res/drawable/` : graphics (PNG, JPEG, etc) - `res/layout/` : UI XML layout files (we're going to talk about these a lot today!) - `res/mipmap/` : launcher icon files - fun fact: MIP comes from "multum in parvo", latin for "much in little". Map cause image mapping - `res/values/` : general constants - `/strings` : short strings - `/colors` : color constants - `/dimen` : dimensions (like default margins) - `/styles` : style and theme details

All of the details about these is a bit scattered throughout the documentation, but Resource Types is a good place to start.

3.1.1 Alternate Resources

These aren't the only names for folders—as I said, part of the goal of resources is that they can be *localized* (changed depending on the device)! - You can specify folders for "alternative" resources (e.g., special handling for another language, or low-res phones). - At runtime, Android will check the config of the device, and try to find an alternative resource that matches that config. If it it *can't* find one, it will fall back to the "default" resource

So what kinds of configurations can we include? See this list. To highlight a few: - Language and region (two-letter ISO codes) - Screen size (small, normal, medium, large, xlarge) - Screen orientation (port, land) - Screen pixel density (dpi) (ldpi, mdpi, hdpi, xhdpi, …) - note: dpi is "dots per inch", so pixels across *relative* to the device size! - xxhdpi is pretty common for high-end devices - Platform version (v1, v4, v7… for each API number)

Configurations are indicated using the **directory name**, giving them the form `<resources_name>(-<config_qualifier>)+` - Demo: we can use the `New Resource` wizard to create a welcome message in another language, and then change the device's language settings to see it adjust! - Switch to `Package` view to see how the folder structure works

3.1.2 XML Details

Resources are usually defined as XML (which is similar in syntax to HTML). - The `strings.xml` resource is pretty simple, but more complex details can be seen in the `activity_main` layout. - Android-specific attributes are namespaced with a `android:` prefix, to avoid any potential conflicts (e.g., so we know we're talking about Android's `text`). - We can use the `@` symbol to reference one resource from another: `@[<package_name>:]<resource_type>/<resource_name>` - We can also use the `+` symbol to create a *new* resource that we can refer to (like declaring a variable inside an attribute) - usually used with `android:id="@+id/identifier"`; we'll see that later

3.1.3 R

So how does this XML get integrated into Android? How do we mix it into the Java? (See here)

When your application is compiled, the build tools (gradle!) **generate** an additional Java class called **`R`** (for "resource"). This class contains what is basically a ton of "constants"—one for each resource! - These constants are organized into subclasses, one for each resource type. This allows you to refer to `[(package_name).]R.resource_type.identifier`—syntax almost like a JSON object! - e.g., `R.string.hello`; `R.drawable.icon`, `R.layout.activity_main` - for most resources, the identifier is the identifier (`id` attribute) - for layouts, the "identifier" is the *filename* (without the xml) - that `@` symbol goes to `R` to look things up! - This class gets regenerated all the time; with Eclipse, often a lot of issues involved needing to regenerate this class so the IDE could find stuff. - Can find the file in `app/build/generated/source/r/debug/...` for fun! (Use IDE to get there)

Note that these static values are often just `int`s—they're pointers to element references!! - If you've done C work, it's like passing a `pointer*` around! - So in the Java, it's almost like we're working with `int` as the data type for a resource (because it's just a pointer TO a resource!). Think the "key" or "index" of that resource. - Android does the hard work of taking that `int`, looking it up in an internal resource table, finding the associated XML file, and then getting the right element out of it. (By hard work, I mean in terms of coding. It's looking stuff up directly in memory, so is really fast `O(1)`).

So how can we access these in Java? Well that `R` class is included, so we can call it as such! - example: `setContentView()` takes in a resource `int`. - Other common method will be `findViewById(int)`, which lets us grab a View element (e.g., a button) from the resource to talk about it in the Java. - We did this yesterday with our button!

3.2 Views

The most common resource type we'll be working with will be Views. `View` is the "superclass" for visual elements: an visual component on the screen is a `View`. - Examples: TextView, ImageView, Button, etc. - Why a super class? Because it means we can use **polymorphism** to be able to treat all these visual elements the same way! - We can lay them out, draw them, click on them, move them, etc. And all the behavior will be the same–though subclasses will have "extra" features

Here's the big magic trick: one subclass of `View` is `ViewGroup`. A `ViewGroup` can contain other "child" Views. But since `ViewGroup` is a `View`… it can contain more `ViewGroups` inside it! Thus we can **nest** Views within Views - This ends up working a lot of HTML! You can have elements (e.g, `<div>`) inside of other elements! - An example of the Composite pattern - This is how we'll do complex layouts!

Views are defined inside of Layouts—that is, inside a layout resource, which is an XML file describing Views. - These resources are "inflated" (rendered) into UI objects that are part of the application. - "inflate" like "unpacked/expanded" into a Java object

An important note: GUI design tools (like the "design tab" of Android Studio) are relatively *new*. - Early developers (hi!) only used XML; that's all we had ("back in my day…") - and early design tools were pathetic - Android Studio's tool is a lot less shabby. But because I'm old-school, I'm going to push you to write the layouts by hand in XML. Good for understanding the pieces, and you should be able to do it anyway. - a lot like how I push using git from the command-line (or my colleagues push Java from command-line). Grognards, all around.

Note that `Layouts` are `ViewGroups` that provide "ordering" and "positioning" information for the Views inside of them - They let the system lay out the Views intelligently/efficiently - **Views shouldn't know their position!** (good OOP design there)

3.2.1 View Components and Properties

But before we get into how to group Views, let's focus on the individual, basic View classes. - Example: look at the provided `TexView` - Example: make an `ImageView` that contains a picture! - note the XML attributes! - Then we can specify the content of that image in the code (e.g., if we wanted to dynamically pick a picture) `java ImageView imageView = (ImageView) findViewById(R.id.myimageview); imageView.setImageResource(R.drawable.myimage);`

Note that all views have 3 basic pieces: - They have **properties** which define the state - They have **methods** which the Java code can call to manipulate them (outside to in) - They can produce **events** which they can use to notify the Java code (inside to out)

We're mostly going to focus on the first one today: Properties. These are usually defined within the resource as XML attributes. So what are some properties? - `android:id` - a unique identifier; unique within the layout (and ideally the whole app) - legal java variable name (because turned into a variable name in `R`) - `lower_case` by convention - style: can prefix with type (`btn`, `edt`) for easy reference! - **note** give each `View` an id, and then it will be automatically "saved" in the `Bundle` when the Activity is destroyed! See here. - Size: `android:layout_width` and `android:layout_height` (see ViewGroup.LayoutParams for docs) - This can be either be a dimension/value (12dp), or more commonly one of two special values: - `wrap_content` (be as big as content, plus padding) - `match_parent` (be as big as parent, minus padding).

Used to be `fill_parent` - A note on dimensions and units! - `dp` is a “density-independent pixel”. On a 160dpi screen, `1dp` == `1px`. But as dpi increases, the #`px` per `dp` goes up - `px` actual screen pixels. **DO NOT USE**. - `sp` is a “scale-independent pixel”. Like `dp`, but scaled by font preference (think `px` vs. `pt` in CSS). **Use this for text (accessibility)** - `pt` is 1/72 of an inch of the physical screen. Also `mm` and `in`. *Not recommended* - Padding & Margin: `android:padding`, `android:paddingLeft`, `android:margin`, `android:marginLeft`, etc. - These work the same way they do in CSS: padding is between the content and the “edge” of the View; margin is between views. - Margin doesn't collapse! - Unlike CSS, styling properties are not inherited. It's like specifying the stuff using the inline `style` attribute. - Text-size, color, etc. - Lots of others as well! Check out the listing in `View`, or look at the options the “design tab” in Android Studio gives you!

Those are some of the main *visual properties*, since we're interesting in developing layouts and interfaces. Note that almost all of these properties can be modified with methods (e.g., `setPadding()`) from within the Java code; but you'd only do that to make things dynamic–specify the layout in the XML! - Other methods are things like `isVisible`, `hasFocus`, etc. We'll point at those as we need them

3.3 Layouts

As mentioned above, a Layout is a grouping of Views (specifically, a `ViewGroup`). They act as containers for other views, to help organize things. - Layouts are all subclasses of `ViewGroup`! So can get a list of them from there.

3.3.1 LinearLayout

Probably the simplest Layout to understand is the `LinearLayout`. This simply orders the children `Views` in a line (“linearly”) - All children are laid out in a single direction, but you can specify the orientation (`android:orientation`) - See LinearLayout.LayoutParams for a list of all options! - Remember, as `Views` you can also use all of the properties we talked about earlier; they are inherited!

The other piece you might want to control is how much of any remaining space the element should occupy (e.g., should it expand)? This is done with the `android:layout_weght` property. - after all sizes are calculated, the remaining space is divided up proportionally to the weight of each element (default 0) - use `0dp` for width and height and `1` for weight to make everything the same size - see also the example in the guide

You can also use `android:layout_gravity` to encourage more details on how elements might be put in the layout (e.g., alignment–where they “fall” to).

Important Point You can also nest `LinearLayouts` inside each other! So you make "grids" by putting a vertical layout containing "rows" of horizontal layouts (containing Views). - There are lots of ways to achieve any layout!

3.3.2 RelativeLayout

A `RelativeLayout` is the default layout given by Android Studio, but it's a bit more complex to use (so I often swap it back to a `LinearLayout`). In a `RelativeLayout`, children are positioned "relative" to the parent **OR** *to each other.* - All children default to the top-left of the View - Can instead give them properties from `RelativeLayout.LayoutParams` about where to go - ex: `android:layout_verticalCenter` centers vertically within parent - ex: `android:layout_toRightOf` centers to the right of View with resource id - Use the `@` symbol to refer to the resource, with the `+` after it (before the id) to define a new `id` `xml <TextView android:id="@+id/first" android:layout_width="match_parent" android:layout_height="wrap_content" android:text="FirstString" /> <TextView android:id="@+id/second" android:layout_height="wrap_content" android:layout_below="@id/first" android:layout_alignParentLeft="true" android:text="SecondString" />` - You don't need to specify "toRightOf" and "toLeftOf". Think about putting down one element, and then putting down the next relative to what came before. - This can get tricky; I'm honestly a fan of using `LinearLayouts` more (since you can always reproduce this kind of work by dividing into enough `Linears`) - [[practice: make an image with a textbox and submit button below it (aligned)]]

3.3.3 Other Layouts

There are other layouts as well, though we won't go over in depth (they work in similar ways; just check the documentation!) - FrameLayout is a sort of "placeholder" layout that holds a **single** child View; can think of it as a way of adding a simple container to use for padding, etc. - TableLayout acts like an HTML table: define `TableRow` layouts which you can fill with content - GridLayout puts things into a Grid (like Linear, but fills the grid first) - This is different than a GridView, which is a scrollable, adaptable list (like a `ListView`, which we'll talk about tomorrow).

3.3.4 Combining Layouts

Last piece: what if we want to combine layouts? Maybe we want to dynamically change what Views are included, or maybe we just want to refactor chunks of the layout into different XML files to improve the organization.

Well, we can statically include XML layouts inside other layouts by using the `<include>` tag:

```
<include layout="@layout/sub_layout">
```

But what if we want to change what View is shown dynamically? It's possible to dynamically load views "manually" (e.g., in code) using the `LayoutInflator`. This is a class that has the job of "inflating" (rendering) Views; it is implicitly used in that `setContentView()` method. It has the following syntax:

```
LayoutInflator inflator = getLayoutInflator(); //access the inflator
View myLayout = inflator.inflate(R.layout.my_layout, parentViewGroup, true); //to attach
```

- Note that we never instantiate the `LayoutInflator`, we just access the one that exists (Factory Pattern!)
- the `inflate` method takes a couple of arguments:
- The first is the resource to inflate (an int!)
- The second is a a group to act as the "parent" for this View–e.g., what layout should it go inside? Can be null for no context.
- The third (optional) is whether to actually attach it to that parent (if not, the parent just provides context/layout params to use). If not assigning to parent, can attach to view using method in `ViewGroup` (e.g., `addView(View)`).

This method works, but it tends to be messy and hard to maintain (UI should be specified in the XML!); so it isn't as common in modern development. A much cleaner solution is to use a `ViewStub`. A `ViewStub` is like an "on deck" (baseball) layout: it is written into the XML, but isn't actually shown until you reveal it with code. - Android inflates the `View` at runtime, but then removes it from the parent (leaving a "stub" in its place). When you call `inflate()` (or `setVisible(View.VISIBLE)`) on that stub, it is reattached to the View tree and displayed.

```
<ViewStub android:id="@+id/stub"
          android:inflatedId="@+id/subTree"
          android:layout="@layout/mySubTree"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content" />
```

```
ViewStub stub = (ViewStub) findViewById(R.id.stub);
View inflated = stub.inflate();
```

Handy!

Chapter 4

Interactive Views

4.1 Inputs [20min]

Yesterday we talked about **Views**. (*What's a View?*). We discussed `ViewGroups` (Layouts), and we've used some basic `Views` like `TextView`, `ImageView`, and `Button`. That last one is an example of an Input Control - These are "simple" (read: single purpose) widgets that allow for use input. - We've already seen Button, but there are others! Mostly in the `android.view` package

[[Show off a demo activity with them all!]] - Button (afford clicking). Can have text, images, or both - EditText (text entry) - can control `android:inputType` (just like inputs in HTML!) - Checkbox (on-off state) - RadioButton (select from a set of choices) - put these into a `RadioGroup` to make them mutually exclusive - ToggleButton (on-off state) - Switch (on-off state; ToggleButton with a slider UI). Introduced in API 14. - Spinner (pick from an array of choices) - define choices in resources (e.g., `strings.xml`)! - Pickers: compound controls around some specific input (dates, times, etc) - typically put in pop-up dialogs, which we'll talk about next week - …and more! (see `android.widget`)

These all mostly work the same way: you define them in the layout resource, and then can access them in Java in order to interact with them.

There are two ways of interacting with controls (and Views in general) from Java: (1) calling methods on them and (2) listening for events from them - Can think of methods as "outside to in" (wrt the control) - Can think of events as "inside to out"

We've already done the event-driven approach to respond to buttons. - We register a **Listener** and specify a **callback method** to get "called back to" when the event occurs. - Find the button, set a listener, etc. - Note that we can also specify a callback method in the resource using the `android:onClick` attribute. The value is the *name* of the method. `xml`

`<Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:onClick="handleButtonClick" />` We declare the callback method in Java as taking in a `View` parameter and returning `void`: `java public void handleButtonClick(View view) { }` The `View` passed to the method will be whatever View caused the event to occur.

- We'll actually use a mix of both of these strategies in this class.
- *Opinion*: arguable about which is better–the resource is easier/faster, but starts mixing the "behavior" and the logic (though since buttons are made to be pressed, it's not unreasonable to give an "name" to what they should do; this can always just be a "launcher" method that calls something else)

We can use listeners like this to respond to all kinds of controls. - Checkboxes use `onClick`, ToggleButtons use `onCheckedChanged`, etc. - Other common events in the View documentation - Include `OnDragListener` (for drags), `OnHoverListener` ("hover" events), `OnKeyListener` (when user types), `OnLayoutChangeListener` (when layout changes display), etc.

In addition to listening for events, we can call methods directly on these Views to access their state. Methods such as `isVisible`, `hasFocus`, etc. give us information about state, but we can also inquire directly about the inputs - For example, the `isChecked()` method lets us look up if a checkbox is ticked whenever we want to know

This is also a good way of getting inputs: for example, we can call `.getText()` on an `EditText` view to fetch the contents of that view - [[demo: log contents when button pressed]]

Between listening for events and querying for state, we can interact with input controls however we want. Check the documentation for more details on how to use specific widgets!

4.2 ListView and Adapters [30min]

((Switch to `list_layout` Layout))!

Now that we've covered basic controls, let's look at some more advanced Views. In particular, the ListView. - It's is a *ViewGroup* that displays a scrollable list of items! - This is basically a `LinearLayout` inside a `ScrollView` (a `ViewGroup` that can be scrolled). - Each item in the linear layout is another `View` (usually a layout) for that particular item - However, The `ListView` does extra work: it keeps track of what items are already laid on the screen, inflating only the visible items plus a few extra on the top and bottom as buffers. Then as the user scrolls, it takes the disappearing views and recycles them (altering the content) to reuse for the new Views that are appearing. This lets it save on memory and performance and work smoothly. See here for some diagrams. - A more advanced and flexible version of this behavior is offered by the RecyclerView class. See this guide.

The `ListView` control uses a **Model-View-Controller** architecture. What's that? - A design pattern common in UI systems - **Model** is the data, **View** is the representation/display of that data, and **Controller** hooks them together! - Originally developed as part of the Smalltalk language - You'll actually find this all over Android. The resources are models and views (separately), with the Java Activities act as controllers - So with a `ListView`, we'll have some data to be displayed, the Views (layouts) to be shown, with the `ListView` control itself acting as a controller

Specifically, the `ListView` is a subclass of `AdapterView`, which is a view backed by a data source–the `AdapterView` hooks the two together (the controller)! - There are other `AdapterViews` as well; `GridView` works exactly the same way, but lays out items in a scrollable grid rather than a scrollable list.

Okay, so let's get our pieces in place 1. First we need the **model**. Some raw data. Let's use a String[] (and we can fill it with whatever data): `java String[] data = new String[99];     for(int i=99; i>0; i--){ data[99-i] = i+ " bottles of beer on the wall";     }`

2. Next we want the **view**. A `View` to show. Let's make a layout resource for that (`list_item` is a good name and a common idiom)

- Put a basic `TextView` in there; no layout needed! (width: `match_parent`, height: `wrap_content`)
- can give `android:minHeight=?android:attr/listPreferredItemHeight` (framework's preferred height), and some `center_vertical` gravity
- `android:lines` if we need more space
- **don't forget an `id`!**

3. Finally, we want the **controller**. The `ListView` itself; let's add that element to our Activity's layout. How big should it be?

To fill in the controller `ListView`, we need to provide it an `Adapter` to use to connect the model to the view. The Adapter does the "translation" work of converting models to views (and getting the model for particular views to show) - We're going to use an `ArrayAdapter`, because we have an array and because it's the simplest to use. - An `ArrayAdapter` creates views by calling `.toString()` on each item in the array and putting that String inside a `TextView`! - Check the parameters of the constructor (and note the generics!)

`java   ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, R.layout.list_item_layout, R.layout.list_item_txtView, myStringArray);` - We then get a reference to the `ListView` with `findViewById`, and call `.setAdapter()` to attach the adapter to that controller.

```
ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

And voila! We have a scrollable list of data! What else can we do with it? - Each item in this list is selectable (can have an onClick), so we can click on them e.g., to go to a "detail" Activity - use `AdapterView#..setOnItemClickListener(ItemClickListener)` to register! - The `position` is where in the list they clicked, use `(Type)parent.getItemAtPostion(position)` to fetch the value. - Each item does have an individual layout, so we can customize these appearances (e.g., if our layout also wanted to include pictures) - See this guide for an example on making a custom adapter to fill in multiple `Views` with data from a list! You will *not* need to do this for your homework, though you can if you want.

And remember, a `GridView` is basically the same thing (in fact, we can just change over that and have everything work, if we use *polymorphism*!)

4.3 Network Data [40min]

We have our lovely `ListView` hooked up to an adapter so it can show a list of Strings. But during lab we were using some code that fetched data from the Internet and gave us a list of Strings… can we combine them?! - You betchya!

Let's go ahead and add in the downloader method (copy and paste into this class, or can just access it in the other), and then we can call it to fill our `String[]` (instead of doing it by hand) - Does it work? What went wrong?! (look at the logs!) `NetworkOnMainThreadException`

Android apps run by default on the ***Main Thread*** (also called the ***UI Thread***). This thread is in charge of all user interactions–handling button presses, scrolls, drags, etc.–but also ui *output* like drawing/displaying text! See Android Threads for more details. - *What's a thread?* (You played with these in lab!) A thread is a piece of a program that is independently scheduled by the processor. Computers do exactly one thing at a time, but make it look like you're doing lots of stuff by switching between them (between processes) really fast. Threads are a way that we can break up a single application or process into little "sub-process" that can be run simultaneously—by switching back and forth periodically so everyone has a chance to work - Within a single thread, all method calls are **synchronous**—that is, one has to finish before the next occurs. We can't get to step 4 without going past step 3. With an event-driven system like Android, each method call is fast enough that this isn't a problem. - But for long, drawn-out processes like network access (or processing bitmaps, or accessing a database), this could cause all of the other stuff to have to wait. It's like a traffic jam! - Things like network access are **blocking** method calls, which stop the Thread from continuing. - A blocked Main Thread will lead to the infamous **"Application not responding" (ANR)** error

We need to move the network code *off* the Main Thread, onto a **background thread**. To do this, we're going to use a class called `ASyncTask` to let us perform a task (like network connecting) asynchronously (without waiting for other people). - How did we find this class? Look at the Processes and Threads Guide! - Note that this background thread will be *tied to the lifecycle of the Activity*—if I close the Activity, the network connection will die as well. - A better, if *much* more complex solution would be to use a `Service`, which we will cover later in the course. But since we're just fetching a small amount of data, we don't really care if the network connection gets dropped. - This class can be pretty complicated to use, but let's look at the documentation and walk through it.

The first thing we should notice (if the API was a little more readable) is that `ASyncTask` is **`abstract`** What does that mean? (We need to subclass it to use it!) - We can also notice that it's a generic class with three (3)! parameters: the type of the Params to the task, the type of the Progress measurement, and the type of the Result. We can start these as `Void` if we don't know them, but we should be able to guess what kind of params/returns we want from our method (take in a `String`, return a `String[]`) - So we can make an *inner* class (non-static, because we want to be tied to *this* Activity) that extends `ASyncTask`. `MovieDownloadTask` is a good name

But what goes inside this class? When we "run" an AsyncTask, it will do four (4) things: 1. `onPreExecute()` is called *on the UI thread* before we run the task. We can do setup here 2. `doInBackground(Params...)` is called *on the background thread* to do that work - we *must* override this (it's `abstract`!) - note it's params and return type needs to match the `ASyncTask` generic types! 3. `onProgressUpdate()` is called *on the UI thread* if we want to update our progress (e.g., update a progress bar) 4. `onPostExecute(Result)` is called *on the UI thread* to process any results (which are passed in when `doInBackground` finished)

The `doInBackground()` is the heart of the method, so let's move our networking code in there!

We can then *instantiate* a new one of these classes in `onCreate()` and call `.execute(params)` to start it running. - Does it work? What went wrong? `SecurityException`!

As a security feature, Android apps by default have very limited access to the system (e.g., to do anything other than show a layout). An app can't use the Internet (which might eat at people's data plans!) without explicit permission from the user. This permission is given at *install time*. - So in order to get permission we need to ask for it ("Mother may I..."). We do that by declaring that the app uses the Internet in the `Manifest` (which has all the details of our app!)

```xml
<uses-permission android:name="android.permission.INTERNET"/>
<!-- put this ABOVE the <application> tag -->
```

- Note that Marshmallow introduced a new security model in which users

grant permissions at *run-time*, not install time, and can revoke permissions whenever they want. To handle this, you basically need to add code to request the permission each time you use it (for "dangerous" permissions, like location, phone or SMS. Internet is *not* dangerous).
- For "normal" permissions (Internet), you declare in the Manifest
- For "dangerous" permissions (Location), you declare in the Manifest *AND* request in code each time you want to use it.
- Since we're targeting Lollipop, won't affect us yet, but something to keep in mind (and we'll look at exactly how to handle this later)

Finally, we can connect to the bloody internet (Log out the results to prove it)!

But how do we get this stuff back into our ListView? - Remember that `doPostExecute()` function? That happens on the *UI Thread* so we can use it to update the View (we can *only* change the View on the UI Thread, to avoid collisions). It also gets the results returned by `doInBackground()` passed to it! - So we can take that String[] and put it into our `ListView`. Specifically, we'll feed it into the `Adapter` - make the Adapter an instance variable so we can access it in this other class - We'll first clear out any previous entries in the adapter using `adapter.clear()` And then we use `.addAll()` to put all the items into the Adapter (or can loop through the data and `.add()` each item). - We can call `.notifyDataSetChanged()` on the `ArrayAdapter` to make sure that the View knows the data has changed, but this is already called by the `.add()` method so isn't necessary here.

How do we search for a different movie?! - Well we can take our `EditText` field and its button from the other Layout, and use it to get a String. We can then pass that String into the `execute` function, since we've declare that the Generic `ASyncTask` takes that type as the first param. - We can actually pass in a lot of Strings, using the `String... params` syntax (arbitrary number of items of that type). See here. The value we *actually* get is an array.

Whew! We've now got some data downloading off the net and showing up on our screen! Spiffy! - We've done a whirl-wind tour of Android in this process: Layouts in the XML, Adapters in the Activity, Threading in a new class, Security in the Manifest… bringing lots of parts together to make a set of functionality. - Note that this is the same process you'll need to do for your homework this week!

Appendix A

Java Review

Android applications are written primarily in the Java Language. This appendix contains a review of some Java fundamentals needed when developing for Android, presented as a set of practice exercises. The code for these exercises can be found at https://github.com/info448-s17/lab-java-review.

A.1 Building Apps with Gradle

Consider the included `Dog` class found in the `src/main/java/edu/info448/review/` folder. This is a very basic class representing a Dog. You can instantiate and call methods on this class by building and running the `Tester` class found in the same folder. - You can just use any text editor, like *VS Code*, *Atom*, or *Sublime Text* to view and edit these files.

You've probably run Java programs using an IDE, but let's consider what is involved in building this app "by hand", or just using the JDK tools. There are two main steps to running a Java program:

1. **Compiling** This converts the Java source code (in `.java` files) into JVM bytecode that can be understood by the virtual machine (in `.class`) files.
2. **Running** This actually loads the bytecode into the virtual machine and executes the `main()` method.

Compiling is done with the `javac` ("java compile") command. For example, from inside the code repo's directory, you can compile both the `.java` files with:

```
# Compile all .java files
javac src/main/java/edu/info498/review/*.java
```

Running is then done with the `java` command: you specify the full package name of the class you wish to run, as well as the classpath so that Java knows where to go find classes it depends on:

```
# Runs the Tester#main() method with the `src/main/java` folder as the classpath
java -classpath ./src/main/java edu.info498.review.Tester
```

Practice: Compile and run this application now.

Practice: Modify the `Dog` class so that it's `.bark()` method barks twice (`"Bark Bark!"`). What do you have to do to test that your change worked?

You may notice that this development cycle can get pretty tedious: there are two commands we need to execute to run our code, and both are complex enough that they are a pain to retype.

Enter **Gradle**. Gradle is a build automation system: a "script" that you can run that will automatically perform the multiple steps required to build and run an application. This script is defined by the `build.gradle` configuration file. ***Practice: open that file and look through its contents***. The `task run()` is where the "run" task is defined: do you see how it defines the same arguments we otherwise passed to the `java` command?

You can run the version of Gradle included in the repo with the `gradlew <task>` command, specifying what task you want to the build system to perform. For example:

```
# on Mac/Linux
./gradlew tasks

# on Windows
gradlew tasks
```

Will give you a list of available tasks. Use `gradlew classes` to compile the code, and `gradlew run` to compile *and* run the code.

- **Helpful hint**: you can specify the "quite" flag with `gradlew -q <task>` to not have Gradle output its build status (handy for the `run` task)

Practice: Use gradle to build and run your Dog program. See how much easier that is?

We will be using Gradle to build our Android applications (which are much more complex than this simple Java demo)!

A.2 Class Basics

Now consider the `Dog` class in more detail. Like all classes, it has two parts:

1. **Attributes** (a.k.a., instance variables, fields, or member variables). For example, `String name`.

    - Notice that all of these attributes are `private`, meaning they are not accessible to members of another class! This is important for **encapsulation**: it means we can change how the `Dog` class is implemented without changing any other class that depends on it (for example, if we want to store `breed` as a number instead of a `String`).

2. **Methods** (a.k.a., functions). For example `bark()`

    - Note the *method declaration* `public void wagTail(int)`. This combination of access modifier (`public`), return type (`void`), method name (`wagTail`) and parameters (`int`) is called the **method signature**: it is the "autograph" of that particular method. When we call a method (e.g., `myDog.wagTail(3)`), Java will look for a method definition that *matches* that signature.

    - Method signatures are very important! They tell us what the inputs and outputs of a method will be. We should be able to understand how the method works *just* from its signature.

Notice that one of the methods, `.createPuppies()` is a `static` method. This means that the method belongs to the ***class***, not to individual object instances of the class! ***Practice: try running the following code (by placing it in the `main()` method of the `Tester` class)***:

```
Dog[] pups = Dog.createPuppies(3);
System.out.println(Arrays.toString(pups));
```

Notice that to call the `createPuppies()` method you didn't need to have a `Dog` object (you didn't need to use the `new` keyword): instead you went to the "template" for a `Dog` and told that template to do some work. *Non-static* methods (ones without the `static` keyword, also called "instance methods") need to be called on an object.

Practice: Try to run the code `Dog.bark()`. What happens? This is because you can't tell the "template" for a `Dog` to bark, only an actual `Dog` object!

In general, in 98% of cases, your methods should **not** be `static`, because you want to call them on a specific object rather than on a general "template" for objects. Variables should **never** be static, unless they are **also** `final` constants (like the `BEST_BREED` variable).

- In Android, `static` variables cause significant memory leaks, as well as just being generally poor design.

A.3 Inheritance

Practice: Create a new file `Husky.java` that declares a new `Husky` class:

```java
package edu.info448.review; //package declaration (needed)

public class Husky extends Dog {
  /* class body goes here */
}
```

The `extends` keyword means that `Husky` is a **subclass** of `Dog`, inheriting all of its methods and attributes. It also means that that a `Husky` instance **is a** `Dog` instance.

Practice: In the Tester, instantiate a new `Husky` and call `bark()` on it. What happens?

- Because we've inherited from `Dog`, the `Husky` class gets all of the methods defined in `Dog` for free!
- Try adding a constructor that takes in a single parameter (`name`) and calls the appropriate `super()` constructor so that the breed is `"Husky"`, which makes this a little more sensible.

We can also add more methods to the **subclass** that the **parent class** doesn't have. ***Practice: add a method called `.pullSled()` to the `Husky` class.***

- Try calling `.pullSled()` on your `Husky` *object*. What happens? Then try calling `.pullSled()` on a `Dog` *object*. What happens?

Finally, we can **override** methods from the parent class. ***Practice: add a `bark()` method to `Husky` (with the same signature), but that has the `Husky` "woof" instead of "bark".*** Test out your code by calling the method in the `Tester`.

A.4 Interfaces

Practice: Create a new file `Huggable.java` with the following code:

```java
package edu.info448.review;

public interface Huggable {
  public void hug();
}
```

This is an example of an **interface**. An **interface** is a list of methods that a class *promises* to provide. By *implementing* the interface (with the `interface` keyword in the class declaration), the class promises to include any methods listed in the interface.

- This is a lot like hanging a sign outside your business that says *"Accepts Visa"*. It means that if someone comes to you and tries to pay with a Visa card, you'll be able to do that!
- Implementing an interface makes no promise about *what* those methods do, just that the class will include methods with those signatures. ***Practice: change the `Husky` class declaration***:

`java public class Husky extends Dog implements Huggable {...}`

Now the the `Husky` class needs to have a `public void hug()` method, but what that method *does* is up to you!

- A class can still have a `.hug()` method even without implementing the `Huggable` interface (see `TeddyBear`), but we gain more benefits by announcing that we support that method.
  - Just like how hanging an "Accepts Visa" sign will bring in more people who would be willing to pay with a credit card, rather than just having that option available if someone asks about it.

Why not just make `Huggable` a superclass, and have the `Husky` extend that?

- Because `Husky` extends `Dog`, and you can only have one parent in Java!
- And because not all dogs are `Huggable`, and not all `Huggable` things are `Dogs`, there isn't a clear hierarchy for where to include the interface.
- In addition, we can implement multiple interfaces (`Husky implements Huggable, Pettable`), but we can't inherit from multiple classes
  - This is great for when we have other classes of different types but similar behavior: e.g., a `TeddyBear` can be `Huggable` but can't `bark()` like a `Dog`!
  - ***Practice: Make the class `TeddyBear` implement `Huggable`. Do you need to add any new methods?***

What's the difference between inheritance and interfaces? The main rule of thumb: use *inheritance* (`extends`) when you want classes to share **code** (implementation). Use *interfaces* (`implements`) when you want classes to share **behaviors** (method signatures). In the end, *interfaces* are more important for doing good Object-Oriented design. Favor interfaces over inheritance!

A.5 Polymorphism

Implementing an interface also establishes an **is a** relationship: so a `Husky` object **is a** `Huggable` object. This allows the greatest benefit of interfaces and inheritance: **polymorphism**, or the ability to treat one object as the type of another!

Consider the standard variable declaration:

```
Dog myDog; //= new Dog();
```

The variable type of `myDog` is `Dog`, which means that variable can refer to any value (object) that **is a** `Dog`.

Practice: Try the following declarations (note that some will not compile!)

```
Dog v1 = new Husky();
Husky v2 = new Dog();
Huggable v2 = new Husky();
Huggable v3 = new TeddyBear();
Husky v4 = new TeddyBear();
```

If the **value** (the thing on the right side) *is an* instance of the **variable type** (the type on the left side), then you have a valid declaration.

Even if you declare a variable `Dog v1 = new Husky()`, the **value** in that object *is* a `Husky`. If you call `.bark()` on it, you'll get the `Husky` version of the method (***Practice: try overriding the method to print out*** ***`"barks like a Husky"`*** ***to see***).

You can **cast** between types if you need to convert from one to another. As long as the **value** *is a* instance of the type you're casting to, the operation will work fine.

```
Dog v1 = new Husky();
Husky v2 = (Husky)v1; //legal casting
```

The biggest benefit from polymorphism is abstraction. Consider:

```
ArrayList<Huggable> hugList = new ArrayList<Huggable>(); //a list of huggable things
hugList.add(new Husky()); //a Husky is Huggable
hugList.add(new TeddyBear()); //so are Teddybears!

//enhanced for loop ("foreach" loop)
//read: "for each Huggable in the hugList"
for(Huggabble thing : hugList) {
    thing.hug();
}
```

Practice: What happens if you run the above code? Because Huskies and Teddy Bears share the same behavior (`interface`), we can treat them as a single "type", and so put them both in a list. And because everything in the list supports the `Huggable` interface, we can call `.hug()` on each item in the list and we know they'll have that method—they promised by `implementing` the interface after all!

A.6 Abstract Methods and Classes

Take another look at the `Huggable` interface you created. It contains a single method declaration… followed by a semicolon instead of a method body. This is an **abstract method**: in fact, you can add the `abstract` keyword to this method declaration without changing anything (all methods are interfaces are implicitly `abstract`, so it isn't required):

```java
public abstract void hug();
```

An **abstract method** is one that does not (yet) have a method body: it's just the signature, but no actual implementation. It is "unfinished." In order to instantiate a class (using the `new` keyword), that class needs to be "finished" and provide implementations for *all* abstract methods—e.g., all the ones you've inherited from an interface. This is exactly how you've used `interfaces` so far: it's just another way of thinking about why you need to provide those methods.

If the `abstract` keyword is implied for interfaces, what's the point? Consider the `Animal` class (which is a parent class for `Dog`). The `.speak()` method is "empty"; in order for it to do anything, the subclass needs to override it. And currently there is nothing to stop someone who is subclassing `Animal` from forgetting to implement that method!

We can *force* the subclass to override this method by making the method `abstract`: effectively, leaving it unfinished so that if the subclass (e.g., `Dog`) wants to do anything, it must finish up the method. ***Practice: Make the `Animal#speak()` method `abstract`. What happens when you try and build the code?***

If the `Animal` class contains an unfinished (`abstract`) method… then that class itself is unfinished, and Java requires us to mark it as such. We do this by declaring the *class* as `abstract` in the class declaration :

```java
public abstract class MyAbstractClass {...}
```

Practice: Make the `Animal` class `abstract`. You will need to provide an implementation of the `.speak()` method in the `Dog` class: try just having it call the `.bark()` method (method composition for-the-win!).

Only abstract classes and `interfaces` can contain `abstract` methods. In addition, an `abstract` class is unfinished, meaning it can't be instantiated. ***Practice: Try to instantiate a `new Animal()`. What happens?*** Abstract classes are great for containing "most" of a class, but making sure that it isn't used without all the details provided. And if you think about it, we'd never want to ever instantiate a generic `Animal` anyway—we'd instead make a `Dog` or a `Cat` or a `Turtle` or something. All that the `Animal` class is doing is acting as an **abstraction** for these other classes to allow them to share implementations (e.g., of a `walk()` method).

- Abstract classes are a bit like "templates" for classes... which are themselves "templates" for objects.

A.7 Generics

Speaking of templates: think back to the `ArrayList` class you've used in the past, and how you specified the "type" inside that List by using angle brackets (e.g., `ArrayList<Dog>`). Those angle brackets indicate that `ArrayList` is a generic class: a template for a class where a *data type* for that class is itself a variable.

Consider the `GiftBox` class, representing a box containing a `TeddyBear`. ***What changes would you need to make to this class so that it contains a*** **`Husky`** ***instead of a*** **`TeddyBear`*****? What about if it contained a*** **`String`** ***instead?***

You should notice that the only difference between `TeddyGiftBox` and `HuskyGiftBox` and `StringGiftBox` would be the **variable type** of the contents. So rather than needing to duplicate work and write the same code for every different type of gift we might want to give... we can use **generics**.

Generics let us specify a data type (e.g., what is currently `TeddyBear` or `String`) as a *variable*, which is set when we instantiate the class using the angle brackets (e.g., `new GiftBox<TeddyBear>()` would create an object of the class with that type variable set to be `TeddyBear`).

We specify generics by declaring the data type variable in the class declaration:

```java
public class GiftBox<T> {...}
```

(`T` is a common variable name, short for "Type". Other options include `E` for Elements in lists, `K` for Keys and `V` for Values in maps).

And then everywhere you would have put a datatype (e.g., `TeddyBear`), you can just put the `T` variable instead. This will be replace by an *actual* type **at compile time**.

- Warning: *always* use single-letter variable names for generic types! If you try to name it something like `String` (e.g., `public class GiftBox<String>`), then Java will interpret the word `String` to be that variable type, rather than refering to the `java.lang.String` class. This a lot like declaring a variable `int Dog = 448`, and then calling `Dog.createPuppies()`.

Practice: Try to make the **`GiftBox`** ***class generic and instantiate a new*** **`GiftBox<Husky>`**

A.8 Nested Classes

One last piece: we've been putting *attributes* and *methods* into classes... but we can also define additional *classes* inside a class! These are called **nested** or **inner classes**.

We'll often nest "helper classes" inside a bigger class: for example, you may have put a `Node` class inside a `LinkedList` class:

```java
public class LinkedList {
  //nested class
  public class Node {
    private int data;

    public Node(int data) {
      this.data = data;
    }
  }

  private Node start;

  public LinkedList() {
    this.start = new Node(448);
  }
}
```

Or maybe we want to define a `Smell` class inside the `Dog` class to represent different smells, allowing us to talk about different `Dog.Smell` objects. (And of course, the `Dog.Smell` class would implement the `Sniffable` interface...)

Nested classes we define are usually `static`: meaning they belong to the *class* not to object instances of that class. This means that there is only one copy of that nested blueprint class in memory; it's the equivalent to putting the class in a separate file, but nesting lets us keep them in the same place and provides a "namespacing" function (e.g., `Dog.Smell` rather than just `Smell`).

Non-static nested classes (or **inner classes**) on the other hand are defined for each object. This is important only if the behavior of that class is going to depend on the object in which it lives. This is a subtle point that we'll see as we provide inner classes required by the Android framework.

Appendix B

Swing Framework

Android applications are user-driven graphical applications. So to look at some of the *patterns* involved in this kind of software (without the overhead of the Android framework), let's consider how to build simple graphical applications in Java using the Swing library - We're going to be doing a little bit of Java programming; easiest to just do this in Sublime Text or whatever light-weight IDE you have floating around.

The **Swing** library is a set of Java classes used to specify graphical user interfaces (GUIs). These classes can be found in the `javax.swing` package. They also rely on the `java.awt` package (the "Advanced Windowing Toolkit"), which is an older GUI library that Swing builds on top of. - Fun fact: Swing library is named after the dance style: the developers wanted to name it after something hip and cool and popular. In the mid-90s.

Let's look at an incredibly basic GUI class: `MyGUI` found in the `src/main/java/` folder. Things to note about this class:

- The class subclasses `JFrame`. `JFrame` represents a "window" in your operating system, and does all the work of making that window show up and interact with the operating system in a normal way. By subclassing `JFrame`, we get that functionality for free! this is how we build all GUI applications.
  - We call the parent constructor (passing in the title for the window), and then specify what happens when we hit the "close" button.
- We then instantiate a `JButton`, which is a Java button
  - Note that `JButton` is the Swing version of a button, building off of the older `java.awt.Button` class.
- We then `.add()` this button to the `JFrame`. This puts the button inside the Window. A little but like using jQuery to add some HTML to the

page.

- Finally, we call `.pack()` to tell the Frame to resize itself to fit the contents, and then `.setVisible()` to make it actually appear.
- We run this program from `main` by just instantiating our specialized `JFrame`, which will contain the button.

We can compile and run it with `./gradlew -q run`. And voila, we have a basic button!

B.1 Events

If we click the button… nothing happens. Let's make it print out a message when clicked. We can do this through **event-based programming** (if you remember `onClick` events from JavaScript, this is the same idea).

The computer sees interactions with its GUI as a series of **events**. The event of clicking a button. The event of moving the mouse. The event of closing a window. Etc. Each thing you interact with *generates* and *emits* these events. So when you click on a button, it creates and emits an "I was clicked!" event. - You can think of this like the button shouting "Hey hey! I was pressed!" We can then respond to this shouting to have our program do something when the button is clicked.

Events, like everything else in Java, are Objects (of the `EventObject` type) that are created by the emitter. A `JButton` in particular emits `ActionEvents` when pressed (the "action" being that it was pressed). - When buttons are pressed, they shout out `ActionEvents`

In order to respond to this shouting, we need to "listen" for these events. Then whenever we hear that there is an event happening, we can react to it. - This is like a person manning a submarine radar. Or hooking up a baby monitor. Or following someone on Twitter, in a way.

But this is Java, so we need an object to listen for these events: a "listener" if you will. Luckily, Java provides a type that can listen for `ActionEvents`: `ActionListener`. This type has an `actionPerformed()` method that can be called in response to an event. - We use the Observer Pattern to connect this listener object to the button (`button.addActionListener(listener)`). This *registers* the listener, so that the Button knows who to shout at when something happens. Again, like following someone on Twitter. When the button is pressed, it will go to any listeners registered with it and call their `actionPerformed()` methods, passing in the `ActionEvent` it generated.

Look again: is `ActionListener` a class? No, it's an interface! Which means if we want to make an `ActionListener` object, we need to create a class that implements this interface (and provides the `actionPerformed()` method that can be called when the event occurs). There are a few ways we can do this:

1. Well we already have a class we're developing: `MyGUI`! So let's just make *that* class `implement ActionListener`. We'll fill in the provided method, and then specify that `this` object is the listener, and voila.
    - This is my favorite way to create listeners in Java (since it keeps everything self-contained: the `JFrame` handles the events its buttons produce).
    - We'll utilize a variant of this pattern in Android: we'll make classes implement listeners, and then "register" that listener somewhere else in the code (often in a nested class).
2. But what if we want to reuse our listener across different classes, but don't want to have to create a new `MyGUI` object to listen for a button to be clicked? We can instead use an **inner** or **nested** class. For example, a nested class `MyActionListener` that implements the interface, and then just instantiate one of these to register with the button.
    - This could be a `static` nested class, but then we can't have it access instance variables (because it belongs to the *class*, not the *object*). So might make it an inner class. Of course then we can't re-use it elsewhere without making the `MyGUI`... but at least we've organized the functionality a bit.
3. It seems sort of silly to create a whole new `MyActionListener` class that has one method and is just going to be instantiated once. So what if instead of giving it a name, we just made it an **anonymous class**?
    - Like how we've made *anonymous variables* by instantiating objects without assigning them to named variables... we can do the same thing with a class that just implements an interface. The syntax looks like:

    ```
    button.addActionListener(new ActionListener() {
        //class declaration goes here!

        public void actionPerformed(ActionEvent) { /*...*/}
    });
    ```

    - This is how buttons are often used in Android: we'll create an anonymous listener object to respond to the event that occurs when they are pressed.

Questions on this kind of event-driven programming?

B.2 Layouts and Composites

One more piece if we have time: what if we want to add a second button? If we try to just `.add()` another button... it replaces the one we just had! This is because Java doesn't know *where* to put the second button. Below? Above? Left? Right?

In order to have the `JFrame` contain multiple components, we need to specify a

layout, which knows how to organize items that are added to the Frame. We do this with the `.setLayout()` method. For example, we can give the frame a `BoxLayout()` with a `PAGE_AXIS` orientation to have it lay out the buttons in a vertical row. - Java has different `LayoutManagers` that each have their own way of organizing components. We'll see this same idea in Android (and talk about Android's layout system in *a lot* more detail next week)!

What if we want to do more complex layouts? Well we can use a different `LayoutManager`… but we can actually achieve a lot of flexibility simply by using *multiple containers*.

For example, we can make a `JPanel` object, which is basically an "empty" component. We can then add multiple buttons to this this panel, and add that panel to the `JFrame`. Because `JPanel` *is a* `Component` (just like `JButton` is), we can use the `JPanel` exactly as we used the `JButton`—this panel just happens to have multiple buttons.

And since we can put any `Component` in a `JPanel`, and `JPanel` is itself a component… we can create nest these components together into a tree in an example of the Composite Pattern. This allows us to create very complex user interfaces with just a simple `BoxLayout`! - We'll see this kind of "nested" composite pattern in Android next week!

Appendix C

Threads and HTTP Requests

This appendix introduces concepts in **concurrency and threading**, which are used extensively by Android though a framework-specific classes and options. For clarity, these concepts are introduced though a set of practice exercises in straight Java (though similar code can be utilized in Android). The code for these exercises can be found at https://github.com/info448-s17/lab-threads-http.

Additionally, this appendix introduces the Java code used to send **network requests**. Android will use *exactly* this code, but in order to experiment with it separate from the Android framework you'll be making network connections directly from Java.

C.1 Concurrency

Concurrency the process by which we have multiple *processes* (think: methods) running at the same time. This can be contrasted with processes that run **serially**, or one after another.

C.1.1 An Example: Algorithm Races!

As an example, note that one of the main concerns of computer science and software in general is speed: how fast will a particular program or algorithm run? For example, give two of the many sorting algorithms that have been invented, which one can sort a list of numbers more quickly?

- Sorting algorithms are usually covered in UW's *CSE 373* course, but don't worry if you haven't taken that course yet! All you need to know is that there are different techniques for sorting numbers, these techniques are given funny names, and one technique may be faster than another

Consider the provided `SortRacer.java` class (found in the `src/main/java` folder). The `main` method for this program runs two different sorting algorithms (currently Merge Sort and Quicksort), reporting when each one is finished.

Practice: Run this program using gradle: `./gradlew -q runSorts`. Note that it may take a few seconds for it to build and begin running, and the sorting itself may take a few seconds!

Of course, it's not really a "race" at the moment: rather, each sorting algorithm is run **serially** (that is, one after another). If we really wanted them to race, we'd like the algorithms to run **concurrently** (at the same time).

Computers as a general rule do exactly one thing a time: your central processing unit (CPU) just adds two number together over and over again, billions of times a second

- The standard measure for *rate* (how many times per second) is the `hertz` (Hz). So a 2 gigahertz (GHz) processor can do 2 billion operations per second.

However, we don't realize that computers do only one thing at a time! This is because computers are really good at *multitasking*: they will do a tiny bit of one task, and then jump over to another task and do a little of that, and then jump over to another task and do a little of that, and then back to the first task, and so on.

These "tasks" are divided up into two types: **processes** and **threads**. ***Read this brief summary of the difference between them***.

So by breaking up a program into threads (which are "interwoven"), we can in effect cause the computer to do two tasks at once. This is *especially* useful if one of the "tasks" might take a really long time–rather than **blocking** the application, we can let other tasks also make some progress while we're waiting for the long task to finish.

C.1.2 Threading the Race

Currently the two sorting algorithms run in the same thread, one after another. You should break them into two *different* threads that can run **concurrently**, letting them actually be able to race!

In Java, we create a Thread by creating a class that `implements` the **`Runnable`** interface. This represents a class that can be "run" in a separate thread! The `run()` method required by the interface acts a bit like the "main" method for

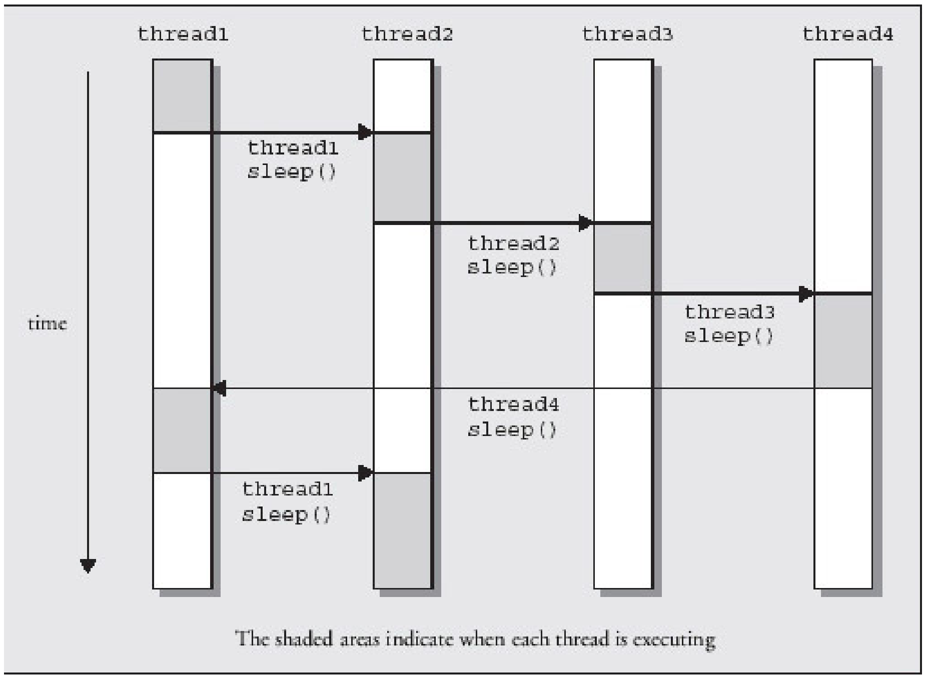


Figure C.1: Diagram of thread switching (source unknown)

that Thread: when we start the Thread running, that is the method that will get called.

Practice: Create two new `Runnable` classes, one for each Sorting method.

- These should be nested classes (think: should they be `static`?).
- When each `Runnable` is `run`, you should create a new *shuffled* array of numbers and then call the appropriate sorting method on that list. Remember to print out when you start and finish sorting (just like is currently done in the `main()` method).

If we just instantiate the `Runnable()` and call its `run()` method, that won't actually execute the method on a different thread (remember: an interface is just a "sign"; we could have called the interface and method whatever we wanted and it would still compile). Instead, we execute code on a separate thread by using an instance of the **`Thread`** class. This class actually does the work of running code on a separate thread.

`Thread` has a constructor that takes in a `Runnable` instance as a parameter—you pass an object representing the "code to run" to the `Thread` object (this is an example of the *Strategy Pattern*). You then can actually **start** the `Thread` by calling its `.start()` method (*not* the `run` method!).

Practice: Modify the `main()` method so you create new `Threads` to execute each `Runnable` Make sure you actually `start()` the threads!

- Anonymous variables will be useful here; you don't need to assign a variable name to the `Runnable` objects or even the `Thread` objects if you just use them directly.

Now run your program! Do you see the Threads running at the same time? Try running the program multiple times and see what kind of differences you get.

- There are some print statements you can uncomment in the `Sorting` class if you want to see more concrete evidence of the Threads running concurrently.
- You are also welcome to try racing different sorting algorithms (you'll want to use a smaller list of numbers, particularly for the painfully slow BubbleSort). You can even race more than two algorithms—just create additional Threads!

And that's the basics of creating Threads in Java!

C.2 HTTP Requests

Consider the provided `MovieDownloader.java` class (found in the `src/main/java/` folder). This Java code (which is *directly* portable to

Android) accesses the database at omdbapi.com, a wrapper around the IMDB API calls for getting information about movies.

You can run this program with the `./gradlew -q runMovies` task. It will prompt you for a movies to search for, and then print out the results (in JSON format).

Practice: add descriptive comments to the `downloadMovieData()` method, explaining what the code does and how it works. The goal is to understand the classes and methods are that are being used here (particularly the use of `HttpUrlConnection`, `InputStream`, and `BufferedReader`), and demonstrate that understanding through explanatory comments. You should also pay particular attention to the use of `try/catch` blocks (see here for one explanation).

Note that we'll utilize this exact code in Android, so you should be familiar with what it is doing!