Android Development: Lecture Notes

Joel Ross

2017-04-03

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

This course focuses on **Android Development**. But what is Android?

Android is an operating system. That is, it's software that connects hardware to software and provides general services. But more than that, it's a *mobile specific* operating system: an OS designed to work on *mobile* (read: handheld, wearable, carry-able) devices.

- Note that the term "Android" also is used to refer to the "platform" (e.g., devices that use the OS) as well as 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. So "Android Development" technically means developing applications that run on the specific OS, it also gets generalized to refer to developing any kind of software that interacts with the platform.

1.1 Android History

If you're going to develop systems for Android, it's good to have some familiarity with the platform and its history, if only to give you perspective on how and why the framework is designed the way it is.

- **2003**: The platform was originally founded by a start-up "Android Inc." which aimed to build a mobile OS operating system (similar to what Nokia's Symbian was doing at the time)
- **2005**: Android was acquired by Google, who was looking to get into mobile
- **2007**: Google announces the Open Handset Alliance, a group of tech companies working together to develop "open standards" for mobile platforms. Members included phone manufacturers like HTC, Samsung, and

Sony; mobile carriers like T-Mobile, Sprint, and NTT DoCoMo; hardware manufacturers like Broadcom and Nvidia; and others. The Open Handset Alliance now (2017) includes 86 companies.

  - Note this is the same year the first iPhone came out!

- **2008**: First Android device is released: the HTC Dream (a.k.a. T-Mobile G1)

  Specs: 528Mhz ARM chip; 256MB memory; 320x480 resolution capacitive touch; slide-out keyboard! Author's opinion: a fun little device.

- **2010**: First Nexus device is released: the Nexus One. These are Google-developed "flagship" devices, intended to show off the capabilities of the platform.

  Specs: 1Ghz Scorpion; 512MB memory; .37" at 480x800 AMOLED capacitive touch.

  - For comparison, the iPhone 7 Plus (2016) has: 2.34Ghz dual core A10 64bit Fusion; 3GB RAM; 5.5" at 1920x1080 display.

  As of 2016, this program has been superceded by the Pixel range of devices.

- **2014**: Android Wear, a version of Android for wearable devices (watches) is announced.

- **2016**: Daydream, a virtual reality (VR) platform for Android is announced

In short, Google keeps pushing the platform wider so it includes more and more capabilities.

Today, Android is incredibly popular (to put it mildly). Android is incredibly popular! (see e.g., here, here, and here)

- In any of these analyses there are some questions about what exactly is counted... but what we care about is that there are *a lot* of Android devices out there! And more than that: there are a lot of **different** devices!

1.1.1 Android Versions

Android has gone through a large number of "versions" since it's release:

Date	Version	Nickname	API Level
Sep 2008	1.0	Android	1
Apr 2009	1.5	Cupcake	3
Sep 2009	1.6	Donut	4
Oct 2009	2.0	Eclair	5
May 2010	2.2	Froyo	8
Dec 2010	2.3	Gingerbread	9

Date	Version	Nickname	API Level
Feb 2011	3.0	Honeycomb	11
Oct 2011	4.0	Ice Cream Sandwich	14
July 2012	4.1	Jelly Bean	16
Oct 2013	4.4	KitKat	19
Nov 2014	5.0	Lollipop	21
Oct 2015	6.0	Marshmallow	23
Aug 2016	7.0	Nougat	24
Mar 2017	O preview	*Android O Developer Preview*	

Each different "version" is nicknamed after a dessert, in alphabetica order. But as developers, what we care about is the **API Level**, which indicates what different programming *interfaces* (classes and methods) are available to use.

- You can check out an interactive version of the history through Marshmallow at https://www.android.com/history/
- For current usage breakdown, see https://developer.android.com/about/dashboards/

Additionally, Android is an "open source" project released through the "Android Open Source Project", or ASOP. You can find the latest version of the operating system code at https://source.android.com/; it is very worthwhile to actually dig around in the source code sometimes!

While new versions are released fairly often, this doesn't mean that all or even many devices update to the latest version. Instead, users get updated phones historically by purchasing new devices (every 18m on average in US). Beyond that, updates—including security updates—have to come through the mobile carriers, meaning that most devices are never updated beyond the version that they are purchases with.

- This is a problem from a consumer perspective, particularly in terms of security! There are some efforts on Google's part to to work around this limitation by moving more and more platform services out of the base operating system into a separate "App" called Google Play Services.
- But what this means for developers is that you can't expect devices to be running the latest version of the operating system—the range of versions you need to support is much greater than even web development!

1.1.2 Legal Battles

When discussing Android history, we would be remiss if we didn't mention some of the legal battles surrounding Android. The biggest of these is **Oracle v Google**. In a nutshell, Oracle claims that the *Java API* is copyrighted (that the method signatures themeselves and how they work are protected), so because Google uses that API in Android, Google is violating the copyright. In 2012 a California federal judge decided in Google favor (that one can't copyright an API). This was then reversed by the Federal Circuit court in 2014. The verdict was appealed to the Supreme courset in 2015, who refused to hear the case. It then went back to the the district court, which ruled that Google's use of the API was fair use. See https://www.eff.org/cases/oracle-v-google for a summary, as well as https://arstechnica.com/series/series-oracle-v-google/

- One interesting side effect of this battle: the latest version of Android (Nougat) uses the OpenJDK implementation of Java, instead of Google's own in-violation-but-fair-use implementation see here. This change *shouldn't* have any impact on us as developers, but it's worth keeping an eye out for potentially differences between Android and Java SE.

There have been other legal challenges as well. While not directly about Android, the other major relevant court battle is **Apple v Samsung**. In this case, Apple claims that Samsung infringed on their intellectual property (their design patents). This has gone back and forth in terms of damages and what is considered infringing; the latest development is that the Supreme Court heard the case and sided with Samsung that infringing design patents shouldn't lead to damages in terms of the entire device... it's complicated (the author is not a lawyer).

So overall: Android is a growing, evolving platform that is embedded in and affecting the social infrastructures around information technology in numerous ways.

1.2 Android Architecture and Code

Developing Android applications involves interfacing with the Android platform and framework. Thus you need a high level understanding of the architecture of the Android platform. See https://source.android.com/devices/ for more details.

Like so many other systems, the Android platform is built as a layered architecture:

- At it's base, Android runs on a Linux kernel for interacting with the device's processor, memory, etc. Thus an Android device can be seen as a Linux computer.
- On top of that kernel is the Hardware Abstraction Layer: an interface to drivers that can programmatically access hardware elements, such as the camera, disk storage, Wifi antenna, etc.
  - These drivers are generally written in C; we won't interact with them directly in this course.
- On top of the HAL is the Runtime and Android Framework, which provides a set of abstraction in the Java language which we all know an love.

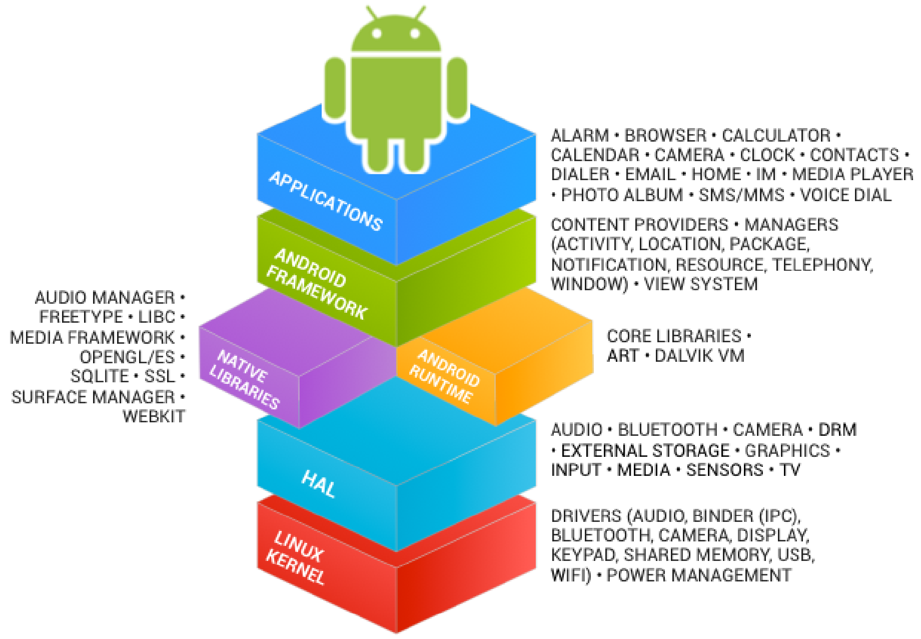
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 1.1: Android Architecture (image from: hub4tech)

For this course, Android Development will involve writing Java applications that interact with the Android Framework layer, which handles the task of interacting with the device hardware for us.

1.2.1 Programming Languages

There are two programming languages we will be working with in this course:

1. **Java:** Android code (program control and logic, as well as data storage and manipulation) is written in Java.

   Writing Android code will feel a lot writing any other Java program: you create classes, define methods, instantiate objects, and call methods on those objects. But because you're working within a **framework**, there is a set of code that *already exists* to call specific methods. As a developer, your task will be to fill in what these methods do in order to run your specific application.

   - In web terms, this is closer to working with Angular (a framework) than jQuery (a library).
   - Importantly: this course expects you to have "journeyman"-level skills in Java (apprenticeship done, not yet master). We'll be us-

ing a number of intermediate concepts (like generics and inheritance) without much fanfare or explanation (though see the appendix).

2. **XML:** Android user interfaces and resources are specified in XML (E**X**tensible **M**arkup **L**anguage). To compare to web programming: the XML contains what would normally go in the HTML/CSS, while the Java code will contain what would normally go in the JavaScript.

   XML is just like HTML, but you get to make up your own tags. Except we'll be using the ones that Android made up; so it's like defining web pages, except with a new set of elements. This course expects you to have some familiarity with HTML or XML, but if not you should be able to infer what you need from the examples.

1.2.2 Building Apps

As stated above, we will write code in Java and XML. But how does that code get run on the phone's hardware?

Pre-Lollipop (5.0), Android code ran on Dalvik: a virtual machine similar to the JVM used by Java SE.

- Fun fact for people with a Computer Science background: Dalvik uses a register-based architecture rather than a stack-based one!

A developer would write *Java code*, which would then be compiled into *JVM bytecode*, which would then be translated into *DVM* (Dalvik virtual machine) bytecode, that could be run on Android devices. This DVM bytecode was stored in `.dex` or `.odex` ("[Optimized] Dalvik Executable") files, which is what was loaded onto the device. The process of converting from Jave code to `dex` files is called **"dexing"** (so code that has been built is "dexed").

Dalvik does include JIT ("Just In Time") compilation to native code that runs much faster than the code interpreted by the virtual machine, similar to the Java HotSpot. This navite code is faster because no translation step is needed to talk to the actual hardware (the OS).

From Lollipop (5.0) on, Android instead uses Android Runtime (ART) to run code. ART's biggest benefit is that it compiles the `.dex` bytecode into native code *on installation* using AOT ("Ahead of Time") compilation. ART continues to accept `.dex` bytecode for backwards compatibility (so the same dexing process occurs), but the code that is actually installed and run on a device is native. This allows for applications to have faster execution, but at the cost of longer install times—but since you only install an application once, this is a pretty good trade.

After being built, Android applications (the source, dexed bytecode, and any resources) are packaged into **`.apk`** files. These are basically zip files (they use the same gzip compression); if you rename the file to be `.zip` and you can unpackage them! The `.apk` files are then cryptographically signed to specify their authenticity, and either "side-loaded" onto the device or uploaded to an App Store for deployment.

- The signed `.apk` files are basically the "executable" versions of your program!
- Note that the Android application framework code is actually "pre-DEXed" (pre-compiled) on the device; when you write code, you're actually compiling against empty code stubs (rather than needing to include those classes in your `.apk`)! That said, any other 3rd-party libraries you include will be copied into your built App, which can increase its file size both for installation and on the device.

To summarize, in addition to writing Java and XML code, when building an App you need to:

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
6. Load it onto the device

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

There are a number of different hardware and software tools you will need to do Android development:

1.3.1 Hardware

Since Android code is written for a virtual machine anyway, Android apps can be developed and built on any computer's operating system (unlike some other mobile OS…).

But obviously Android apps will need to be run on Android devices. Physical devices are the best for development (they are the fastest, easiest way to test), though you'll need USB cable to be able to wire your device into your computer. Any device will work for this course; you don't even need cellular service (just WiFi should work). Note that if you are unfamiliar with Android devices, you should be sure to play around with the interface to get used to the interaction language, e.g., how to click/swipe/drag/long-click elements to use an app.

- You will need to turn on developer options in order to install development apps on your device!

If you don't have a physical device, it is also possible to use the Android Emulator, which is a "virtual" Android device. The emulator represents a generic device with hardware you can specify... but it does have some limitations (e.g., no cellular service, no bluetooth, etc).

- While it has improved recently, the Emulator historically does not work very well on Windows; I recommend you develop on either a Mac or a physical device. In either case, make sure you have enabled HAXM (Intel's Acceleration Manager, which allows the emulator to utilize your GPU for rendering): this speeds things up considerably.

1.3.2 Software

Software needed to develop Android applications includes:

- The Java 7 **SDK** (not just the JRE!) This is because you're writing Java code!
- Gradle or Apache ANT. These are *automated build tools*—in effect, they let you specify a single command that will do a bunch of steps at once (e.g., compile files, dex files, move files, etc). These are how we make the "build script" that does the 6 build steps listed above.
  - ANT is the "old" build system, Gradle is the "modern" build system (and so what we will be focusing on).
  - Note that you do not need to install Gradle separately for this course.
- Android Studio & Android SDK is the official IDE for developing Android applications. Note that the IDE comes bundled with the SDK. Android Studio provides the main build system: all of the other software (Java, Gradle) goes to support this.

  The SDK comes with a number of useful command-line tools. These include:

  - `adb`, the "Android Device Bridge", which is a connection between your computer and the device (physical *or* virtual). This tool is used for console output!
  - `emulator`, which is a tool used to run the Android emulator
  - *deprecated/removed* `android`: a tool that does SDK/AVD (Android Virtual Device) management. Basically, this command-line utility did everything that the IDE did, but from the command-line! It has recently been removed from the IDE.

  I recommend making sure that the SDK command-line tools are installed. Put the `tools` and `platform-tools` folders on your computer's `PATH`; you can run `adb` to check that everything works. All of these tools are built into the IDE, but they can be useful fallbacks for debugging.

1.4 Hello World

As a final introductory steps, this lecture will walk you through creating and running a basic App so that you can see what you will actually 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 your UW NetID in the domain.
   - Make a mental note of the project location so you can find your code later!
   - *Target*: this is the “minimum” SDK you support. We’re going to target Ice Cream Sandwich (4.0.3, API 15) for most this class, as the earliest version of Android most our apps will support.
     - Note that this is different than the “target SDK”, which is the version of Android you tested your application against (e.g., what system did you run it on?) For this course we will be testing on API 21 (Lollipop); we’ll specify that in a moment.
3. Select an *Empty Activity*
   - **Activities** are “Screens” in your application (things the user can do). Activities are discussed in more detail in the next lecture.
4. And boom, you 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 at the top of the IDE. But we’ll need a device to run the app on, so let’s make an emulator!

The **Nexus 5** is a good choice for supporting “older” devices. The new Pixel is also a reasonable device to test against.

- You’ll want to make sure you create a Lollipop device, using the Google APIs (so we have special classes available to us), and amost certainly running on x86 (Intel) hardware
- Make sure that you’ve specified that it accepts keyboard input. You can always edit this emulator later (`Tools > Android > AVD Manager`).

After the emulator boots, you 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 go to the **"Project"** view you can see what the actual file system looks like. In Android view, files are organized as follows:

- `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. You 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* in 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

  Note that ANT would instead 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 it is good to 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/`: contains graphics (PNG, JPEG, etc)
    - `res/layout/`: contains UI XML layout files
    - `res/mipmap/`: conatins launcher icon files in different resolutions
        * Fun fact: MIP stands for "*multum in parvo*", which is Latin for "much in little" (because multiple resolutions of the images are stored in a single file). "Map" is used because Mipmaps are normally used for texture mapping.
    - `res/values/`: contains XML definitions for general constants

  See also: http://developer.android.com/guide/topics/resources/available-resources.html, or Lecture 3.

We can also consider what the application code does. While we'll revisit this in more detail in the next lecture, it's useful to start seeing how the framework is structured:

We'll start with the **MyActivity** Java source file. This class extends `Activity` (actually it extends a subclass that supports Material Design components), allowing us making our own customizations to what the app does.

In this class, we override the `onCreate()` method that is called by the framework when the Activity starts (see next lecture).

- We call the super method, and then `setContentView()` to specify what the content (appearance) of our Activity is. This is passed in a value from something called `R`. `R` is a class that is **generated at compile time** and contains constants that are defined by the XML "resource" files! Those files are converted into Java variables, which we can access through the `R` class.

`R.layout` refers to the "layout" XML resource, so can go there (remember: inside `res/`). Opening these XML files they appear in a "design" view. This view lets you use a graphical system to lay out your application (similar to a PowerPoint slide).

- However, even as the design view becomes more powerful, using it is still frowned upon by many developers for historical reasons. It's often cleaner to write out the layouts and content in code. This is the same difference between writing your own HTML and using something like FrontPage or DreamWeaver or Wix to create a page. While those are legitimate applications, they are less "professional".

In the code view, we can see the XML: tags, attributes, values. Tags nested inside one another. The provided XML code defines a layout, and inside that is a `TextView` (a View representing some text), which has a value: text! We can change that and then *re-run the app* to see it update!

- It's also possible to define this value in `values/strings` (e.g., as a constant), then refer to as `@string/message`. More on this proces later.

Finally, as a fun demonstration, try to set an icon for the App (in Android Studio, go to: `File > New > Image Asset`)

Chapter 2

Activities and Logging

This lecture introduces **Activities**, which are the basic component used in Android applications. It aims to demonstrate how the interactive patterns used in other graphical applications are utilized in Android.

This lecture references code found at https://github.com/info448-s17/lecture02-activities, in the `android/` folder. As a first step, you'll need to create a new Android application with a single **Empty** Activity (e.g., `MainActivity`). Future chapters will have starter code to work from, but it is good practice to make a new application from scratch!

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 a 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 begin by thinking of them as full screens. We can have lots of Activities (screens) in an application, and they are loosely connected so we can easily move between them.

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!

Also to note from the documentation[1]:

> 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 Activity can represent what a user is doing at one time. If the

[1] https://developer.android.com/reference/android/app/Activity.html

user does something else, that should be a different Activity (and so probably a different screen).

2.1 Making Activities

We create our own activities by *subclassing* (extending) the framework's `Activity` class. We use **inheritance** to make a specialized type of `Activity` (similar to extending `JFrame` in Swing apps). By extending this class we inherit all of the methods that are needed to control how the Android OS interacts with the Activity.

If you look at the default Empty `MainActivity`, it actually subclasses `AppCompatActivity`, which is a already specialized kind of Activity that provides an `ActionBar` (the toolbar at the top of the screen with the name of you app). If you change the class to just extend `Activity`, that bar disappears.

To make this change, you will need to import the `Activity` class! The keyboard shortcut in Android Studio is `alt+return`, or you can do it by hand (look up the package)! You can also set Android Studio to automatically import classes you use.

There are a number 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 "sub-activities" that get nested in larger Activities. We'll talk about Fragments more in a letter lecture.

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

2.2 The Activity Lifecycle

Although we never call a constructor or `main`, Activities do have an *incredibly* well-defined lifecycle—that is, a series of **events** that occur during usage (e.g., when the Activity is created, when it is stopped, etc).

When each of these events occur, Android executes a **callback method**, similar to how you call `actionPerformed()` to react to a "button press" event in Swing. We can **override** these methods in order to do special actions (read: run our own code) when these events occur.

What is the lifecycle?

[3] http://developer.android.com/images/activity_lifecycle.png

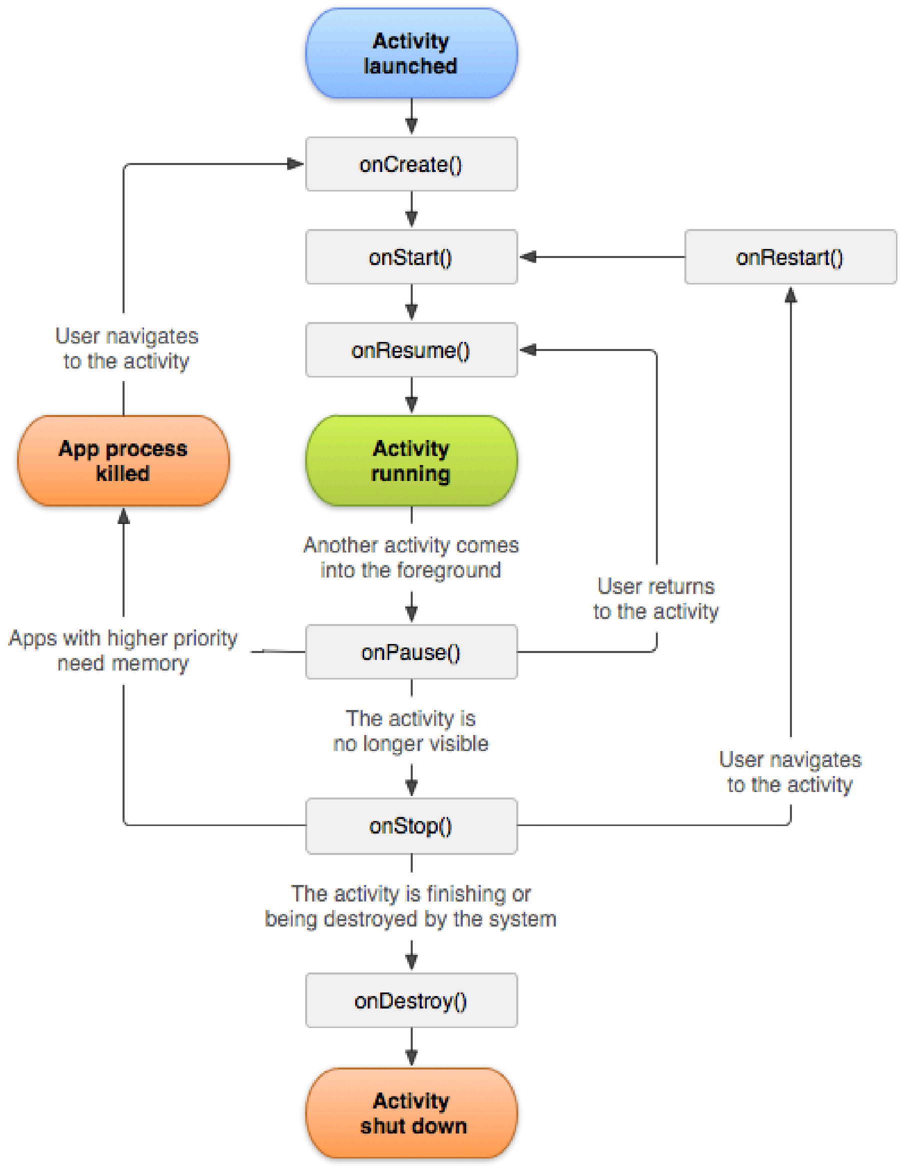
Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to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

Figure 2.1: Lifecycle state diagram, from Google. See also an alternative, simplified diagram here.[3]

There are 7 “events” that occur in the Activity Lifecycle, which are designated by the *callback function* that they execute:

- **`onCreate()`**: called when the Activity is **first** created/instantiated. This is where you initialize the UI (e.g., specify the layout to use), similar to what might go in a constructor.
- **`onStart()`**: called just before the Activity becomes **visible** to the user.

  The difference between `onStart()` and `onCreate()` is that `onStart()` can be called more than once (e.g., if you leave the Activity, thereby hiding it, and come back later to make it visible again).
- **`onResume()`**: called just before **user interaction** starts, indicating that the Activity is ready to be used! This is a little bit like when that Activity “has focus”.

  While `onStart()` is called when the Activity becomes visible, `onResume()` is called when then it is ready for interaction. It is possible for an Activity to be visible but not interactive, such as if there is a modal pop-up in front of it (partially hiding it).
- **`onPause()`**: called when the system is about to start another Activity (so about to lose focus). This is the “mirror” of `onResume()`. *When paused, the activity stays visible!*

  This callback is usually used to *quickly and temporarily* store unsaved changes (like saving an email draft in memory) or stop animations or video playback. The Activity may be being left (on its way out), but could just be losing focus.
- **`onStop()`**: called when the activity is no longer visible. (e.g., another Activity took over, but this also be because the Activity has been destroyed. This callback is a mirror of `onStart()`.

  This callback is where you should persist any state information (e.g., saving the user’s document or game state). It is intended to do more complex “saving” work than `onPause()`.
- **`onRestart()`**: called when the Activity is coming back from a “stopped” state. This event allows you to run distinct code when the App is being “restarted”, rather than created for the first time. It is the least commonly used callback.
- **`onDestroy()`**: called when the 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 the App.

  Android apps run on devices with significant hardware constraints in terms of both memory and battery life. Thus the Android OS is very aggressive about not leaving Apps running “in the background”. If it determines that an App is no longer necessary (such as because it has been hidden for a while), that App will be destroyed. Note that this destruction is unpredictable, as the "necessity" of an App being open is dependent on the OS's resource allocation rules.

The `onDestroy()` callback can do final app cleanup, but its better to have such functionality in `onPause()` or `onStop()`.

Note that apps may not need to use all of these callbacks! 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 Callback Methods

In the default `MainActivity` the `onCreate()` callback has already been overridden for us, since that's where the layout is specified.

Notice that this callback 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, or dictionary in Python). Bundles can only hold basic types (numbers, Strings) and so are used for temporarily "bunding" *small* amounts of information.

This `Bundle` parameter in particular stores 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! The `Bundle` stores current layout information in it by default (if the Views have ids)—technically, it calls a `onSaveInstanceState()` callback for each View in the layout, and the provided Views that we utilize tend to save important state information (like entered text) already. See Saving and restoring activity state for details.

Also note that we call `super.onCreate()`. ***Always call up the inheritance chain!***. This allows the system-level behavior to continue without any problem.

We can also add other callbacks: for example, `onStart()` (see the documentation for examples).

But how can we know if the lifecycle events are getting called?

2.3 Logging & ADB

In Android, we can't use `System.out.println()` because we don't actually have a terminal to print to! More specifically, the device (which is where the application is running) doesn't have access to standard out (`stdout`), which is what Java means by `System.out`.

- It is possible to get access to `stdout` with `adb` using `adb shell stop; adb shell setprop log.redirect-stdio true; adb shell start`, but this is definitely not ideal.

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). Logged messages can be filtered, categorized, sorted, etc. Logging can also be disabled in production builds for performance reasons (though it often isn't).

To perform this logging, we'll use the `android.util.Log`[4] 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`.

- Remember to import the `Log` class!

The device's log file is stored persistantly... sort of. It's a 16k file, but it is shared across the *entire* system. Since every single app and piece of the system writes to it, it fills up fast. Hence filtering/searching becomes important, and you tend to watch the log (and debug your app) in real time!

2.3.1 Log Methods

`Log` provides methods that correspond to different level of priority (importance) of the messages being recorded. From low to high priority:

- **`Log.v()`**: VERBOSE output. This is the most detailed, for everyday messages. This is often the go-to, default level for logging.

  Ideally, `Log.v()` calls should only be compiled into an application during development, and removed for production versions.

- **`Log.d()`**: DEBUG output. This is intended for lower-level, less detailed messages (but still code-level, that is referring to specific programming messages).

  These messages can be compiled into the code but are removed at runtime in production builds through Gradle.

- **`Log.i()`**: INFO output. This is intended for "high-level" information, such at the user level (rather than specifics about code)
- **`Log.w()`**: WARN output. For warnings
- **`Log.e()`**: ERROR output. For errors
- Also if you look at the API... `Log.wtf()`!

These different levels are used to help "filter out the noise". So you can look just at errors, at errors and warnings, at error, warn, and info... all the way down

[4]http://developer.android.com/reference/android/util/Log.html

to seeing *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 tag 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:

```
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 Logcat Command-line Tool for more details.

Demo: And now we can finally log out some of the Lifecycle callbacks to see them being executed!

- Start by implementing `onResume()`. Note the wonders of tab completion! Have it log out at `INFO` level. On the device, hit the main `menu` (circle) button to send the Activity to the background, and watch the callback be executed.
- Implement `onStop()` and switch out of the app to watch it be stopped.
- `onDestroy()` can easily be called if you set the phone to "Don't Keep Activities" (at bottom of developer settings). Or you can simply *rotate* the phone (which causes the Activity to be destroyed and then recreated in the new orientation).
- 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

Once you can “output” some content (via Log), the next step is to add some “input” via an interface element: for example, a Button we can click.

In **`res/layouts/activity_main.xml`** (the Activity’s layout), add the following code inside the `<android.support.constraint.ConstraintLayout>` element, **replacing** the current `<TextView>` element.

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

This XML defines a Button. The `android:text` attribute specifies the text that is on the button. The next lecture will describe in more detail how exactly this XML works (and what’s is meant by the `id`, and `layout_width/height`), but you should be able to make a pretty good educated guess based on the names.

- 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 be able to click on it. So we need to register a “listener” for it (in Java), just like with Swing apps:

```
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 we defined in the XML. The `findViewById()` method “finds” the appropriate XML element with the given `id`. We discuss why we wrote the parameter as `R.id.my_button` in the next lecture tomorrow. Note that this method returns a `View`, so we want to **cast** the value into the more specific `Button` (which has methods we want to use).

We can register a listener with that button through the `.setOnClickListener()` method, passing in an **anonymous class** to act as the listener. (Again, tab-completion is our friend!). This is *just like* what you would do with a Swing app.

Finally, we can fill in the method to have it log out something when clicked.

Overall, this button is an example of an Input Control. These will be discussed in more detail in Lecture 4.

2.5 Multiple Activities

The whole point of interfacing with the Activity Lifecycle is handle the fact that Android applications can have multiple activities and interact with multiple other applications. In this section we'll briefly discuss how to include multiple Activities within an app (in order to sense how the life cycle may affect them). Note that working with multiple Activities will be discussed in more detail in a later lecture.

We can easily create a New Activity through Android Studio by using `File > New > Activity`. We could also just add a new `.java` file with the Activity class in it, but using Android Studio will also provide the `onCreate()` method stub as well as a layout resource.

- For practice, make a new **Empty** Activity called `SecondActivity`. You shuld edit this Activity's layout resource so that the `<TextView>` displays an appropriate message.

Importantly, for every Activity we make, an entry gets added to the **Manifest** file `AndroidManifest.xml`. This file acts like the *"table of contents"* for our application, telling the device Operating System information about what our app looks (that is, what Activities it has) like so that the OS can open appropriate Activities as needed.

Activities are listed as `<activity>` elements nested in the `<application>` element. If you inspect the file you will be able to see an element representing the first `MainActivity`; that entry's child elements will be discussed later.

- We can add `android:label` attributes to these `<activity>` elements in order to give the Activities nicer display names (e.g., in the ActionBar).

2.5.1 Intents and Context

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 perform a particular action (i.e., start up and display on the screen). These messages are called **Intents**, and are used to communicate between app components like Activities. The Intent system allows Activities to communicate, even though they don't have references to each other (we can't just call a method on them).

- I don't have a good justification for the name, other than Intents announce an "intention" for the OS to do something (like start an Activity)
- 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.

An `Intent` is an object we *can* instantiate: for example, we can create a `new Intent` in the event handler for when we click the button on `MainActivity`. The `Intent` class has a number of different cnstructors, but the one we'll start with looks like:

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

The second parameter to this constructor is the *class* we want to send the Intent to (the `.class` property fetches a reference to the class type; this is metaprogramming!). Effectively, it is the "address" on the envelop for the message we're sending.

The first parameter refers to the current **Context**[5] in which the message should be delivered. `Context` is an **abstract class** (and a superclass of `Activity`) that acts as a reference for information about the current running environment: it represents environmental data (information like "What OS is running? Is there a keyboard plugged in?"). You can *almost* think of the Context as representing the "Application", though it's broader than that (`Application` is actually a subclass of `Context`!)

The Context is *used* to do "application-level" actions: mostly working with resources (accessing and loading them), 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 wish to refer to:

- The Application context (e.g., an `Application` object) references the state of the entire application. It's basically the Java object that is built out of the `<application>` element in the Manifest (and so contains that level of information).
- The Activity context (e.g., an `Activity` object) that references the state of that Activity. Again, this roughly corresponds to the Java objects created out of the `<activity>` tags from the Manifest.

Each of these `Context` objects exist for the life of its respective component: that is, an `Activity` Context is available as long as the Activity exists (disappearing after `onDestroy()`), whereas `Application` Contexts survive as long as the application does. Note htat we'll almost always use the `Activity` context, as it's safer and less likely to cause memory leaks.

- Inside an `Activity` object (e.g., in a lifecycle callback function), you can refer to the current `Activity` using `this`. And since `Activity` is a `Context`, you can also use `this` to refer to the current Activity context. You'll often see `Context` methods called as undecorated methods (without an explicit `this`).

[5]https://developer.android.com/reference/android/content/Context.html

After having instantiated the `new Intent`, we can use that message to start an Activity by calling the `startActivity()` method (inherited from `Activity`), passing it the `Intent`:

```
startActivity(intent);
```

This method will "send" the message to the operating system, which will deliver the Intent to the appropriate Activity, telling that Activity to start as soon as it receives the message.

With this interaction in place, we can now click a button to start a second activity, and see how that impacts our Lifecycle callbacks.

- 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.

- For example, if you look back at the Manifest, you can see that the `MainActivity` has an `<intent-filter>` child element that allows it to receive particular kinds of Intents—including ones for when an App is launched for the first time!

2.6 Back & Tasks

We've shown that we can have lots of Activities (and of course many more can exist cross multiple apps), and we are able to move between them by sending Intents and clicking the "Back" button. But how exactly is that "Back" button able to keep track of where to go to?

The abstarct data type normally associated with "back" or "undo" functionality is a **stack**, and that is exactly what Android uses. Every time you *start* a new Activity, Android instantiates that object 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 Activity that is now at the top.

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 feed 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 of your device.

Tasks usually start from the Android "Home Screen"—when you launch an application, that then starts a new Task. Starting new Activities from that application will add them to the Stack of the task. If you go *back* to the Home

[6] http://developer.android.com/images/fundamentals/diagram_backstack.png

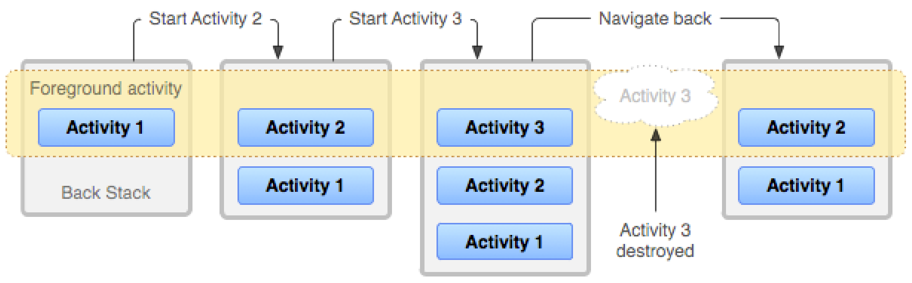

Figure 2.2: An example of the Activity stack, from Google[6].

Screen, the Task you're currently on is moved to the background, so the "back" button won't let you navigate that Stack.

- It's useful to think of Tasks as being like different tabs or browsers, with the "back stack" being the history of web pages visited wthin that tab.
- As a demonstration, try switching to another (built-in) app and then back to the example app; how does the back button work in each situation?

An 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). It is possible to modify this behavior though, see - Though it is possible to modify this, see Managing Tasks

2.6.1 Up Navigation

We can make this "back" navigation a little more intuitive for users by providing explicit up navigation, rather than just forcing users 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). To do this, we just need to add a little bit of configuration to our Activities:

- In the Java code, we want to add more functionality to the `ActionBar`. *Think*: which lifecycle callback should this specification be put in?

  ```java
  //specify that the ActionBar should have an "home" button
  getSupportActionBar().setHomeButtonEnabled(true);
  ```

- Then in the **Manifest**, add an `android:parentActivityName` attribute to the `SecondActivity`, with a value set to the full class name (including package **and** appname!) of your `MainActivity`. 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.

```
<activity android:name=".SecondActivity"
        android:label="Second Activity"
        android:parentActivityName="edu.uw.activitydemo.MainActivity">
        <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="edu.uw.activitydemo.MainActivity" />
</activity>
```

The `<meta-data>` element is to provide backwards compatibilit for API level 15 (since the `android:parentActivityName` is only defined for API level 16+).

2.7 Toasts

We have previously demonstrated how to use `Log` to output messages to logcat. 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 printline debugging, which is still a legitimate debugging process.

Note that Android Studio does have a built-in debugger if you're comfortable with such systems.

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 (and users can't normally read logs). Android provides a number of different classes for doing visual notifications, including alert-style and customizable Dialogs, which we'll talk about in a later lecture.

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 to quickly display a message.

- It's called a "Toast" because it pops up!

Toasts are pretty simple to implement, as with the following example (from the official documentation):

```
Context context = this; //getApplicationContext(); //use application context to avoid disappea
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 anonymously, 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 code 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 a way to provide information to 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, especially when trying to solve crashes or see more complex output.

Chapter 3

Resources and Layouts

This lecture discusses **Resources**, which are used to represent elements or data that are separate from the behavior (functional logic) of the app. In particular, this lecture focuses on how resources are used to define **Layouts** for user interfaces. While the Activities lecture focused on the Java portion of Android apps; this lecture focuses on the XML.

This lecture references code found at https://github.com/info448-s17/lecture03-layouts.

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. This is because resources represent elements (e.g., content) that is *separate* from the code (the behavior of the app), so is kept separate from the Java code to support the **Principle of Separation of Concerns**

- By defining resources in XML, they can be developed (worked on) *without* coding tools (e.g., with systems like the graphical "layout design" tab). Theoretically 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, keeping resources separate allows you to 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!

  - This is similar to how in web development we may want to have the same JavaScript from different HTML.

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), though behavior will be defined programmtically (in Java)
- Any graphic images (drawables) should be resources
- Any *user-facing* strings should be resources
- Style and theming information should be resources

As introduced in Lecture 1, there are a number of different resource types used in Android, many of which can be found in the `res/` folder of a default Android project, including:

- `res/drawable/`: contains graphics (PNG, JPEG, etc)
- `res/layout/`: contains UI XML layout files
- `res/mipmap/`: conatins launcher icon files in different resolutions
- `res/values/`: contains XML definitions for general constants
  - `/strings`: short string constants (e.g., labels)
  - `/colors`: color constants
  - `/styles` : constants for style and theme details
  - `/dimen` : dimensional constants (like default margins); not created by default in Android Studio 2.3+.

The details about these different kinds of resources is a bit scattered throughout the documentation, but Resource Types[1] is a good place to start, as is Providing Resources.

3.1.1 Alternate Resources

These aren't the only names for resource folders: as mentioned above, part of the goal of resources is that they can be **localized**: changed depending on the device! You are thus able to specify folders for "alternative" resources (e.g., special handling for another language, or for low-resolution devices). At runtime, Android will check the configuration of the device, and try to find an alternative resource that matches that config. If it it *can't* find a relevant alternative resource, it will fall back to the "default" resource.

There are many different configurations that can be used to influence resources; see Providing Resources[2]. To highlight a few options, you can specify different resources based on:

- Language and region (e.g., via two-letter ISO codes)
- Screen size(`small`, `normal`, `medium`, `large`, `xlarge`)
- Screen orientation (`port` for portrait, `land` for landscape)

[1]https://developer.android.com/guide/topics/resources/available-resources.html
[2]http://developer.android.com/guide/topics/resources/providing-resources.html

- Specifc screen pixel density (dpi) (`ldpi`, `mdpi`, `hdpi`, `xhdpi`, `xxhdpi`, etc.). `xxhdpi` is pretty common for high-end devices. Note that dpi is "dots per inch", so these values represent the number of pixels across *relative* to the device size!
- Platform version (`v1`, `v4`, `v7`… for each API number)

Configurations are indicated using the **directory name**, giving them the form `<resource_name>(-<config_qualifier>)+`

- You can see this in action by using the *New Resource* wizard (`File > New > Android resource file`) to create a welcome message (a string resource, such as for the `app_name`) in another language[3], and then changing the device's language settings to see the content automatically adjust!

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

- Switch to the `Package` view in Android Studio to see how the folder structure for this works.

3.1.2 XML Details

Resources are usually defined as XML (which is similar in syntax to HTML). The `strings.xml` example used above involves fairly simple elements but more complexresource is pretty simple, but more complex details can be seen in the `activity_main.xml` resource inside `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` instead of something else).
- We can use the **`@`** symbol to reference one resource from another, following the schema `@[<package_name>:]<resource_type>/<resource_name>`
- We can also use the **`+`** symbol to create a *new* resource that we can refer to; this is a bit like declaring a variable inside an attribute. This is most commonly used with the `android:id` attribute (`android:id="@+id/identifier"`), see below for details.

3.1.3 R

Although XML resources are defined separately from the Java code, resources can be accessed from within Java. When an application is compiled, the build tools (e.g., gradle) **generate** an additional Java class called **`R`** (for "resource"). This class contains what is basically a giant list of static "constants"—one for

[3]https://www.webucator.com/blog/2010/03/saying-hello-world-in-your-language-using-javascript/

each resource! These constants are organized into subclasses, one for each resource type. This allows you to refer to a specific resource in the Java code as `[(package_name).]R.resource_type.identifier` similar to the kind of syntax used to refer to a nested JSON object! For example: `R.string.hello` (the `hello` string resource), `R.drawable.icon` or `R.layout.activity_main`

- For most resources, the identifier is defined as an element attribute (`id` for specific View elements in layouts, `name` attribute for values). For more complex resources such as entire layouts or drawables, the identifier is the *filename* (without the XML); hence `R.layout.activity_main` refers to the root element of the `layout/activity_main.xml` file.
- Note that that `@` symbol used in the XML goes to the `R` Java file to look things up, so follows the same reference syntax.

You can find the generated `R.java` file inside `app/build/generated/source/r/debug/...` (Use the Project Files view in Android Studio).

The static constants inside the `R.java` file are often just `ints` that are *pointers* to element references (similar to passing a `pointer*` around in the C language). So in the Java, we usually work with `int` as the data type for XML resources, because we're actually working with pointers *to* those resources.

- You can think of each `int` constant as a "key" or "index" for that resource (in the list of all resources). 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 that XML. (By hard work, I mean in terms of implementation. Android is looking up these references directly in memory, so the look-up is a fast `O(1)`).

Because the `R` class is included in the Java, we can access these constants directly in our code (as `R.resource_type.identifier`). For example, the `setContentView()` call in an Activity's `onCreate()` takes in a resource `int`.

- The other comment method that utilizes resources will be `findViewById(int)`, which is used to reference a `View` element (e.g., a button) from the resource in order to call methods on it in Java. This is the same method used with the Button example in the Activities lecture

The `R` class is regenerated all time (any time you change a resource, which is often); when Eclipse was the recommend Android IDE, you often needed to manually regenerate the class so that the IDE's index would stay up to date! You can perform a similar task in Android Studio by using `Build > Clean Project` and `Build > Rebuild Project`.

3.2 Views

The most common type of element we'll define in resources are **Views**[4]. `View` is the superclass for visual interface elements—a visual component on the screen is a View. Specific types of Views include: TextViews, ImageViews, Buttons, etc.

- `View` is a superclass for these components because it allows us to use **polymorphism** to treat all these visual elements the same way as instances of the same type. We can lay them out, draw them, click on them, move them, etc. And all the behavior will be the same—though subclasses can also have "extra" features

Here's the big trick: one subclass of `View` is `ViewGroup`[5]. 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, following the Composite Pattern. This ends up working a lot like HTML (which can have DOM elements like `<div>` inside other DOM elements), allowing for complex user interfaces.

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.

Technically, `Layouts` are simply `ViewGroups` that provide "ordering" and "positioning" information for the Views inside of them. they let the system "lay out" the Views intelligently and effectively. *Individual views shouldn't know their own position*; this follows from good good object-oriented design and keeps the Views encapsulated.

Android studio does come with a graphical Layout Editor (the "Design" tab) that can be used to create layouts. However, most developers stick with writing layouts in XML. This is mostly because early design tools were pathetic and unusable, so XML was all we had. Although Android Studio's graphical editor can be effective, for this course you should create layouts "by hand" in XML. This is helpful for making sure you understand the pieces underlying development, and is a skill you should be comfortable with anyway (similar to how we encourage people to use `git` from the command-line).

3.2.1 View Properties

Before we get into how to group Views, let's focus on the individual, basic `View` classes. As an example, consider the `activity_main` layout in the lecture code. This layout contains two individual `View` elements (inside a `Layout`): a `TextView` and a `Button`.

[4]http://developer.android.com/reference/android/view/View.html
[5]http://developer.android.com/reference/android/view/ViewGroup.html

All View have **properties** which define the state of the View. Properties are usually defined within the resource XML as element *attributes*. Some examples of these property attributes are described below.

- **`android:id`** specifies a unique identifier for the View. This identifier needs to be unique within the layout, though ideally is unique within the entire app (for clarity).

  Identifiers must be legal Java variable names (because they are turned into a variable name in the `R` class), and by convention are named in `lower_case` format.

  - *Style tip*: it is useful to prefix each View's id with its type (e.g., `btn`, `txt`, `edt`). This helps with making the code self-documenting.

  You should give each interactive `View` a unique id, which will allow its state to automatically be saved as a `Bundle` when the Activity is destroyed. See here for details.

- **`android:layout_width`** and **`android:layout_height`** are used to specify the View's size on the screen (see ViewGroup.LayoutParams for documentation). These values can be a specific value (e.g., `12dp`), but more commonly is one of two special values:

  - `wrap_content`, meaning the dimension should be as large as the content requires, plus padding.
  - `match_parent`, meaning the dimension should be as large as the *parent* (container) element, minus padding. This value was renamed from `fill_parent` (which has now been deprecated).

  Android utilizes the following dimensions or units:

  - **`dp`** is a "density-independent pixel". On a 160-dpi (dots-per-inch) screen, `1dp` equals `1px` (pixel). But as dpi increases, the number of pixels per `dp` increases. These values should be used instead of `px`, as it allows dimensions to work independent of the hardware's dpi (which is *highly* variable).
  - **`px`** is an actual screen pixel. *DO NOT USE THIS* (use `dp` instead!)
  - **`sp`** is a "scale-independent pixel". This value is like `dp`, but is scale by the system's font preference (e.g., if the user has selected that the device should display in a larger font, `1sp` will cover more `dp`). *You should **always** use `sp` for text dimensions, in order to support user preferences and accessibility.*
  - **`pt`** is 1/72 of an inch of the physical screen. Similar units `mm` and `in` are available. *Not recommended for use.*

- **`android:padding`**, **`android:paddingLeft`**, **`android:margin`**, **`android:marginLeft`**, etc. are used to specify the margin and padding for Views. These work basically the same way they do in CSS: padding is the space between the content and the "edge" of the View, and margin is the space between Views. Note that unlike CSS, margins between elements do not collapse.

- **`android:textSize`** specifies the "font size" of textual Views (use `sp` units!), **`android:textColor`** specifies the color of text (reference a color resource!), etc.

- There are lots of other properties as well! You can see a listing of generic properties in the `View`[6] documentation, look at the options in the "Design" tab of Android Studio, or browse the auto-complete options in the IDE. Each different `View` class (e.g., `TextView`, `ImageView`, etc.) will also have their own set of properties.

Note that unlike CSS, styling properties specified in the layout XML resources are not inherited; we're effectively specifying an inline `style` attribute for that element, and one that won't affect child elements. In order to define shared style properies, you'll need to use styles resources, which are discussed in a later lecture.

While it is possible to specify these visual properties dynamically via Java methods (e.g., `setText()`, `setPadding()`). You should **only** use Java methods to specify View properties when they *need* to be dynamic (e.g., the text changes in response to a button click)—it is much cleaner and effective to specify as much visual detail in the XML resource files as possible. It's also possible to simply replace one layout resource with another (see below).

- Views also have inspection methods such as `isVisible()` and `hasFocus()`; we will point to those as we need them.

Do not define Views or View appearances in an Activity's `onCreate()` callback, unless the properties (e.g., content) truly cannot be determined before runtime! Specify layouts in the XML instead.

3.2.2 Practice

Add a new `ImageView` element that contains a picture. Be sure and specify its `id` and size (experiment with different options).

You can specify the content of the image in the XML resource using the `android:src` attribute (use `@` to reference a `drawable`), or you can specify the content dynamically in Java code:

```
ImageView imageView = (ImageView)findViewById(R.id.img_view);
imageView.setImageResource(R.drawable.my_image);
```

[6] http://developer.android.com/reference/android/view/View.html#lattrs

3.3 Layouts

As mentioned above, a Layout is a grouping of Views (specifically, a `ViewGroup`). A Layout acts as a container for other Views, to help organize things. Layouts are all subclasses of `ViewGroup`, so you can use its inheritance documentation to see a (mostly) complete list of options, though many of the listed classes are deprecated in favor of later, more generic/powerful options.

3.3.1 LinearLayout

Probably the simplest Layout to understand is the `LinearLayout`. This Layout simply orders the children View in a line ("linearly"). All children are laid out in a single direction, but you can specify whether this is horizontal or vertical with the `android:orientation` property. See LinearLayout.LayoutParams for a list of all attribute options!

- Remember: since a `Layout` is a `ViewGroup` is a `View`, you can also utilize all the properties discussed above; the attributes are inherited!

Another common property you might want to control in a LinearLayout is how much of any remaining space the elements should occupy (e.g., should they expand). This is done with the `android:layout_weght` property. After all element sizes are calculated (via their individual properties), the remaining space within the Layout is divided up proportionally to the `layout_weight` of each element (which defaults to `0` so they get no extra space). See the example in the guide for more details.

- *Useful tip*: Give elements `0dp` width or height and `1` for weight to make everything in the Layout the same size!

You can also use the `android:layout_gravity` property to specify the "alignment" of elements within the Layout (e.g., where they "fall" to). Note that this property is specified on individual child Views.

An important point Since Layouts *are* Views, you can of course nest `LinearLayouts` inside each other! So you can make "grids" by creating a vertical Layout containing "rows" of horizontal Layouts (which contain Views). As with HTML, there are lots of different options for achieving any particular interface layout.

3.3.2 RelativeLayout

A `RelativeLayout` is more flexible (and hence powerful), but can be more complex to use. In a `RelativeLayout`, children are positioned "relative" to the parent **OR** *to each other*. All children default to the top-left of the Layout, but you can give them properties from `RelativeLayout.LayoutParams` to specify where they should go instead.

For example: `android:layout_verticalCenter` centers the View vertically within the parent. `android:layout_toRightOf` places the View to the right of the View with the given resource id (use an `@` reference to refer to the View by its 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" />
```

(Recall that the `@+` syntax defines a *new* View id, like declaring a variable!)

You do not need to specify both `toRightOf` and `toLeftOf`; think about placing one element on the screen, then putting another element relative to what came before. This can be tricky. For this reason the author prefers to use `LinearLayouts`, since you can always produce a Relative positioning using enough LinearLayouts (and most layouts end up being linear in some fashion anyway!)

3.3.3 ConstraintLayout

`ConstraintLayout` is a Layout provided as part of an extra support library, and is what is used by Android Studio's "Design" tool (and thus is the default Layout for new layout resources). `ConstraintLayout` works in a manner conceptually similar to `RelativeLayout`, in that you specify the location of Views in relationship to one another. However, `ConstraintLayout` offers a more powerful set of relationships in the form of *constraints*, which can be used to create highly responsive layouts. See the class documentation for more details and examples of constraints you can add.

The main advantage of `ConstraintLayout` is that it supports development through Android Studio's Design tool. However, since this course is focusing on implementing the resource XML files rather than using the specific tool (that may change in a year's time), we will primarily be using other layouts.

3.3.4 Other Layouts

There are many other layouts as well, though we won't go over them all in depth. They all work in similar ways; check the individual class's documentatoion for details.

- FrameLayout is a sort of "placeholder" layout that holds a **single** child View (a second child will not be shown). You can think of this layout as a way of adding a simple container to use for padding, etc. It is also highly useful for situations where the framework requires you to specify a Layout resource instead of just an individual View.
- GridLayout arranges Views into a Grid. It is similar to LinearLayout, but places elements into a grid rather than into a line.

  Note that this is different than a Grid_View_, which is a scrollable, adaptable list (similar to a `ListView`, which is discussed in the next lecture).
- TableLayout acts like an HTML table: you define `TableRow` layouts which can be filled with content. This View is not commonly used.

3.3.5 Combining and Inflating Layouts

It is possible to combine multiple layout resources. This is useful if you want to dynamically change what Views are included, or to refactor parts of a layout into different XML files to improve code organization.

As one option, you can *statically* include XML layouts inside other layouts by using an `<include>` element:

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

But it is also possible to dynamically load views "manually" (e.g., in Java code) using the `LayoutInflator`. This is a class that has the job of "inflating" (rendering) Views. The process is called "inflating" based on the idea that it is "unpacking" or "expanding" a compact resource description into a complex Java Object. LayoutInflator is implicitly used in the `setContentView()` method, but can also be used independently with the following syntax:

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

Note that we never instantiate the `LayoutInflator`, we just access an object that is defined as part of the Activity.

The `inflate()` method takes a couple of arguments:

- The first parameter is a reference to the resource to inflate (an `int` saved in `R`)
- The second parameter is a `ViewGroup` to act as the "parent" for this View—e.g., what layout should the View be inflate inside? This can be `null` if there is not yet a layout context; e.g., you wish to inflate the View but not show it on the screen yet.
- The third (optional) parameter is whether to actually attach the inflated View to that parent (if not, the parent just provides context and layout params to use). If not assigning to parent on inflation, you can later attach

the View using methods in `ViewGroup` (e.g., `addView(View)` similar to what we've done with Swing).

Manually inflating a View works for dynamically loading resources, and we will often see UI implementation patterns that utilize Inflators.

However, for dynamic View creation it tends to be messy and hard to maintain (UI work should be specified entirely in the XML, without needing multiple references to parent and child Views) so it isn't as common in modern development. A much cleaner solution is to use a `ViewStub`[7]. A `ViewStub` is like an "on deck" Layout: it is written into the XML, but isn't actually shown until you choose to reveal it via Java code. With a `ViewStub`, 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:

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

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

[7]http://developer.android.com/training/improving-layouts/loading-ondemand.html

Chapter 4

Interactive Views

This lecture discusses how to use Views to support user interaction and dynamic content, building on the previous lecture as while drawing on concepts introduced in the Threads and HTTP Requests Appendix.

This lecture references code found at https://github.com/info448-s17/lecture04-inputs-lists.

4.1 Inputs

The previous lecture discussed **Views** and ViewGroups (**Layouts**), and introduced some basic Views such as `TextView`, `ImageView`, and `Button`.

A `Button` is an example of an Input Control. These are simple (single-purpose; not necessarily lacking complexity) widgets that allow for user input. There are many such widgets in addition to `Button`, mostly found in the `android.widget` package. Many correspond to HTML `<input>` elements, but Android provided additional widgets at well.

Launch the lecture code's `MainActivity` with a content View of `R.id.input_control_layout` to see an example of many widgets (as well as a demonstration of a more complex layout!). These widgets include:

- Button, a widget that affords clicking. Buttons can display text, images or both.
- EditText, a widget for user text entry. Note that you can use the `android:inputType` property to specify the type of the input similar to an HTML `<input>`.
- Checkbox, a widget for selecting an on-off state
- RadioButton, a widget for selecting from a set of choices. Put `RadioButton` elements inside a `RadioGroup` element to make the buttons mutually

exclusive.
- ToggleButton, another widget for selecting an on-off state.
- Switch, yet another widget for selecting an on-off state. This is just a `ToggleButton` with a slider UI. It was introduced in API 14 and is the "modern" way of supporting on-off input.
- Spinner, a widget for picking from an array of choices, similar to a drop-down menu. Note that you should define the choices as a resource (e.g., in `strrings.xml`).
- Pickers: a compound control around some specific input (dates, times, etc). These are typically used in pop-up dialogs, which will be discussed in a future lecture.
- …and more! See the `android.widget` package for further options.

All these input controls basically work the same way: you define (instantiate) them in the layout resource, then access them in Java in order to define interaction behavior.

There are two ways of interacting with controls (and Views in general) from the Java code:

1. Calling **methods** on the View to manipulate it. This represents "outside to inside" communication (with respect to the View).
2. Listening for **events** produced by the View and responding to then. This represents "inside to outside" communication (with respect to the View).

An example of the second, event-driven approach was introduced in Lecture 2. This involved *registering a listener* for the event (after acquiring a reference to the View with `findViewById()`) and then specifying a **callback method** (by instantiating the Listener interface) that wiould be "called back to" when the event occurs.

- It is also possible to specify the callback method in the XML resource itself by using e.g., the `android:onClick` attribute. This value of this attribute should be the *name* of the callback method: It is also possible to

  ```
  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="handleButtonClick" />
  ```

  The callback method is declared in the Java code as taking in a `View` parameter (which will be a reference to whatever View caused the event to occur) and returning `void`:

  ```
  public void handleButtonClick(View view) { }
  ```

- We will utilize a mix of both of these strategies (defining callbacks in both the Java and the XML) in this class.

  Author's Opinion: It is arguable about which approach is "better". Spec-

> ifying the callback method in the Java code helps keep the appearance and behavior separate, and avoids introducing hidden dependencies for resources (the Activity must provide the required callback). However, as buttons are made to be pressed, it isn't unreasonable to give a "name" in the XML resource as to what the button will do, especially as the corresponding Java method may just be a "launcher" method that calls something else. Specifying the callback in the XML resource may often seem faster and easier, and we will use whichever option best supports clarity of our code.

Event callbacks are used to respond to all kind of input control widgets. CheckBoxes use an `onClick` callback, ToggleButtons use `onCheckedChanged`, etc. Other common events can be found in the View documentation, and are handled via listeners such as `OnDragListener` (for drags), `OnHoverListener` (for "hover" events), `OnKeyListener` (for when user types), or `OnLayoutChangeListener` (for when layout changes display).

In addition to listening for events, it is possible to call methods directly on referenced Views to access their state. In addition to generic View methods such as `isVisible()` or `hasFocus()`, it is possible to inquire directly about the state of the input provided. For example, the `isChecked()` method returns whether or not a checkbox is ticked.

This is also a good way of getting access to inputted content from the Java Code. For example, call `getText()` on an `EditText` control in order to fetch the contents of that View.

- For practice, try to log out the contents of the included `EditText` control when the `Button` is pressed!

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

4.2 ListViews and Adapters

The remainder of the lecture utilizes the `list_layout` Layout in the lecture code. Modify `MainActivity` so that it uses this resource as its `viewContent`.

Having covered basic controls, this section will now look at some more advanced interactive Views. In particular, it will discuss how to utilize a ListView, which is a `ViewGroup` that displays a scrollable list of items! A `ListView` is basically a `LinearLayout` inside of a `ScrollView` (which is a `ViewGroup` that can be scrolled). Each element within the LinearLayout is another `View` (usually a Layout) representing a particular item in a list.

But the `ListView` does extra work beyond just nesting Views: it keeps track of what items are already displayed on the screen, inflating only the visible items (plus a few extra on the top and bottom as buffers). Then as the user scrolls, the ListView takes the disappearing views and *recycles* them (altering their content, but not reinflating from scratch) in order to reuse them for the new items that appear. This lets it save memory, provide better performance, and overall work more smoothly. See this tutorial for diagrams and further explanation of this recycling behavior.

- Note that a more advanced and flexible version of this behavior is offered by the `RecyclerView`. See also this guide for more details.

The `ListView` control uses a **Model-View-Controller (MVC)** architecture. This is a deisgn pattern common to UI systems which organizes programs into three parts:

1. The **Model**, which is the data or information in the system
2. The **View**, which is the display or representation of that data
3. The **Controller**, which acts as an intermediary between the Model and View and hooks them together.

The MVC pattern can be found all over Android. At a high level, the resources provide *models* and *views* (separately), while the Java Activities act as *controllers*.

- *Fun fact*: The Model-View-Controller pattern was originally developed as part of the Smalltalk language, which was the first Object-Oriented language!

Thus in order to utilize a `ListView`, we'll have some data to be displayed (the *model*), the *views* (Layouts) to be shown, and the `ListView` itself will connect these together act as the *controller*. Specifically, the `ListView` is a subclass of `AdapterView`, which is a View backed by a data source—the `AdapterView` exists to hook the View and the data together (a controller!)

- There are other `AdapterViews` as well. For example, `GridView` works exactly the same way as a `ListView`, but lays out items in a scrollable grid rather than a scrollable list.

In order to use a `ListView`, we need to get the pieces in place:

1. First we specify the **model**: some raw data. We will start with a simple `String[]`, filling it with placeholder data:

   ```
   String[] data = new String[99];
   for(int i=99; i>0; i--){
       data[99-i] = i+ " bottles of beer on the wall";
   }
   ```

   While we could define this data as an XML resource, we'll create it dynamically for testing (and to make it changeable later!)

2. Next we specify the **view**: a `View` to show for each datum in the list. Define an XML layout resource for that (`list_item` is a good name and

a common idiom).

For simplicity's sake we don't need to specify a full Layout, just a basic `TextView`. Have the width `match_parent` and the height `wrap_content`. *Don't forget an* `id`*!*

```xml
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/txtItem"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

To make it look better, you can specify `android:minHeight=?android:attr/listPreferredItemHeight` (using the framework's preferred height for lists), and some `center_vertical` gravity. The `android:lines` property is also useful if you need more space.

3. Finally, we specify the **controller**: the `ListView` itself. At that item to the Activity's Layout resource (*practice*: what should its dimensions be?)

To finish the controller `ListView`, we ned to provide it with an `Adapter` which will connect the *model* to the *view*. The Adapter does the "translation" work between model and view, performing a mapping from data types (e.g., a `String`) and View types (e.g., a `TextView`).

Specifically, we will use an `ArrayAdapter`, which is one of the simplest Adapters to use (and because we have an array of data!) An `ArrayAdapter` creates Views by calling `.toString()` on each item in the array, and setting that `String` as the content of a `TextView`!

```java
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    R.layout.list_item_layout, R.layout.list_item_txtView, myStringArray);
```

- Note the parameters of the constructor: a `Context`, the item layout resource, the TextView reource, and the data array. Also note that this instance utilizes *generics*, since we're using an array of `Strings` (as opposed to an array of `Dogs` or some other type).

We acquire a reference to the `ListView` with `findViewById()`, and call `ListView#setAdapter()` to attach the adapter to that controller.

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

And that's all that is needed to create a scrollable list of data!

Each item in this list is selectable (can have an `onClick` callback). This allows us to click on any item in order to (for example) view more details about the item. Utilize the `AdapterView#setOnClickListener(OnItemClickListener)` function to register the callback.

- The `postion` parameter in the `onItemClick()` callback is the index of the item which was clicked. Use `(Type)parent.getItemAtPosition(position)` to access the data value associated with that View.

Additionally, 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 tutorial for an example on making a custom adapter to fill in multiple `Views` with data from a list!

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

In the previous section we created a `ListView` utilizing an adapter to display a list of Strings. But Appendix C provides an implementation for fetching data from the Internet which gave us a list of Strings. Can we combine these? You betchya!

The lecture code provides a `MovieDownloader` class containing the exact same networking code utilized in the Appendix. We can then simply specify that the *model* `String[]` should be the result of the `downloadMovieData()` method, rather than manually created with a loop.

If you test this code, you'll notice that it doesn't work! The program will crash with a `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 and displaying text! See Android Threads for more details.

- 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 they are doing lots of tasks simultaneously by switching between them (i.e., 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. You can't get to step 4 without finishing step 3. With an event-driven system like Android, each method call is fast enough that this isn't a problem (you're done handling one click by the time the next occurs). But long, drawn-out processes like network access (or processing bitmaps, or accessing a database), could cause other tasks to have to wait. It's like a traffic jam!

- Tasks such as 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!

Thus we need to move the network code *off* the Main Thread, onto a **background thread**, thereby allowing it to run without blocking the user interaction that occurs on the Main Thread. To do this, we will use a class called `ASyncTask` to perform a task (such as network access) asynchronously—without waiting for other Threads.

Learning Android Development involves knowing about what classes exist, and can be used to solve problems, but how were we able to learn about the existing of this highly useful (and specialized) `ASyncTask` class? We started from the official API Guide on Processes and Threads Guide, which introduces this class! Thus to learn about new Android options, *read the docs*.

Note that an `ASyncTask` background thread will be *tied to the lifecycle of the Activity*: if we close the Activity, the network connection will die as well. A better but *much* more complex solution would be to use a `Service`—which is covered in a future lecture. But since this example just involves getting a small amount of data, we don't really care if the network connection gets dropped.

`ASyncTask` can be fairly complicated, but is a good candidate to practice learning from the API documentation. Looking at that documentation, the first thing you should notice (or would if the API was a little more readable) is that `ASyncTask` is **`abstract`**, meaning you'll need to *subclass* it in order to use it. Thus you can subclass it as an *inner* class inside the Activity that will use it (`MovieDownloadTask` is a good name).

You should also notice that `ASyncTask` is a *generic* class with three (3) generic parameters: the type of the Parameter to the task, the type of the Progress measurement reported by the task, and the type of the task's Result. We can fill in what types of Parameter and Result we want from our asynchronous method (e.g., take in a `String` and return a `String[]`), and use the `Void` type for the Progress measurement (since we won't be tracking that).

When we "run" an AsyncTask, it will do four (4) things, represented by four methods:

1. `onPreExecute()` is called *on the UI thread* before we run the task. This method can be used to perform any setup for the task.
2. `doInBackground(Params...)` is called *on the background thread* to do the work we want to be performed asynchronously. We **must** override this method (it's `abstract`!) The params and return type for the method need to match the `ASyncTask` generic types.
3. `onProgressUpdate()` can be indirectly called *on the UI thread* if we want to update our progress (e.g., update a progress bar). Note that UI changes can **only** be made on the UI thread!
4. `onPostExecute(Result)` is called *on the UI thread* to process any task results, which are passed as parameters to this method when `doInBackground` is finished.

The `doInBackground()` is what occurs on the background thread (and is the heart of the task), so we put our network accessing method call in there.

We can then *instantiate* a new `ASyncTask` object in the Activity's `onCreate()` callback, and call `ASyncTask#execute(params)` to start the task running on its own thread.

If you test this code, you'll notice that it still doesn't work! The program will crash with a `SecurityException`.

As a security feature, Android apps by default have very limited access to the overall operating system (e.g., to do anything other than show a layout). An app can't use the Internet (which might consume people's data plans!) without explicit permission from the user. This permission is given by the user at *install time*.

In order to get permission, the app needs to ask for it ("Mother may I…"). We do that by declaring that the app uses the Internet in the `Manifest.xml` file (which has all the details of our app!)

```
<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 need to add code to request "dangerous" permissions (like Location, Phone, or SMS access; Internet is *not* dangerous) each time you use it.

- For "normal" permissions (e.g., Internet), you declare the permission need in the Manifest.
- For "dangerous" permissions (e.g., Location), you declare the permission need in the Manifest **and** request permission programmatically in code each time you want to use it.

Once we've requested permission (and have been granted that permission by virtue of the user installing our application), we can finally connect to the Internet to download data. We can log out the request results to provide it.

In order to get the downloaded data into a ListView, we utilize the `doPostExecute()` method. This method is run 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!

We take that passed in `String[]` and put that into the `ListView`. Specifically, we feed it into the `Adapter`, which then works to populate the views.

- First clear out any previous data items in the adapter using `adapter.clear()`.
- Then use `adapter.add()` or (`adapter.addAll()`) to add each of the new data items to the Adapter's model.
- You can call `notifyDataSetChanged()` on the Adapter to make sure that the View knows the data has changed, but this method is already called by the `.add()` method so isn't necessary in this situation.

To finalize the app: we can enable the user to search for different movies by copying the `EditText` and `Button` Views from the previous `input_layout` resource, accessing the text from the former when the later is pressed. We can then pass the `EditText` content String into the `ASyncTask#execute()` function (since we've declared that the generic `ASyncTask` takes that type as the first Parameter).

- We can actually pass in multiple `String` arguments using the `String... params` spread operator syntax (representing an arbitrary number of items of that type). See here for details. The value that the `ASyncTask` methods *actually* get is an array of the arguments.

In the end, we are able to downlod data from the Internet and show an interactive list of that data in the app! 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 provide a particular piece of functionality.

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/info448/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.info448.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:

```
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:

```
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(Huggable 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. In order to become familiar with some of the *coding 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

This appendix references code found at https://github.com/info448-s17/lecture02-activities, in the `java/` folder. Note that this tutorial involves Java Programming: while it is possible to do this in Android Studio, it's often easier to just utilize a light-weight text editor such as Visual Studio Code or Sublime Text.

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. The class *subclasses* (extends) `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 using this framework.

Most of the work defining a Swing GUI happens in the `JFrame` constructor (called when the GUI is "created").

1. We first call the parent constructor (passing in the title for the window), and then call a method to specify what happens when we hit the "close" button.

2. We then instantiate a `JButton`, which is a class representing a Java Button. Note that `JButton` is the Swing version of a button, building off of the older `java.awt.Button` class.
3. We then `.add()` this button to the `JFrame`. This puts the button inside the window. This process is similar to using jQuery to add an HTML element to web page.
4. Finally, we call `.pack()` to tell the Frame to resize itself to fit the contents, and then `.setVisible()` to make it actually appear.
5. We run this program from `main` by just instantiating our specialized `JFrame`, which will contain the button.

You can compile and run this program with `./gradlew -q run`. And voila, we have a basic button app!

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 handling `click` events from JavaScript, this is the same idea).

Most computer systems see 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 write code to respond to this shouting to have our application 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). In other words, 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.

But this is Java, and everything in Java is based on Objects, 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.

But look carefully: `ActionListener` is not a concrete class, but an abstract **interface**. This 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. We already have a class we're developing: `MyGUI`! So we can 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, create a nested class `MyActionListener` that implements the interface, and then just instantiate one of those to register with the button.

    - This could be a `static` nested class, but then it wouldn't be able to access instance variables (because it belongs to the *class*, not the *object*). So you might want to make it an inner class instead. Of course then you can't re-use it elsewhere without making the `MyGUI` (whose instance variables it referenes anyway)... but at least we've organized the functionality.

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**? This is similar to how you've made *anonymous variables* by instantiating objects without assigning them to named variables, you're just doing the same thing with a class that just implements an interface. The syntax looks like:

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

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

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.

B.2 Layouts and Composites

What if we want to add a second button? If we try to just `.add()` another button… it replaces the one we previously 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.

```
container.setLayout(new BoxLayout(container, BoxLayout.PAGE_AXIS));
container.add(theButton);
container.add(otherButton);
```

- Java has different `LayoutManagers` that each have their own way of organizing components. We'll see this same idea in Android.

What if we want to do more complex layouts? We could look for a more complex `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`!

- This is similar to how we can create complex web layouts just by nesting lots of `<div>` elements.

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

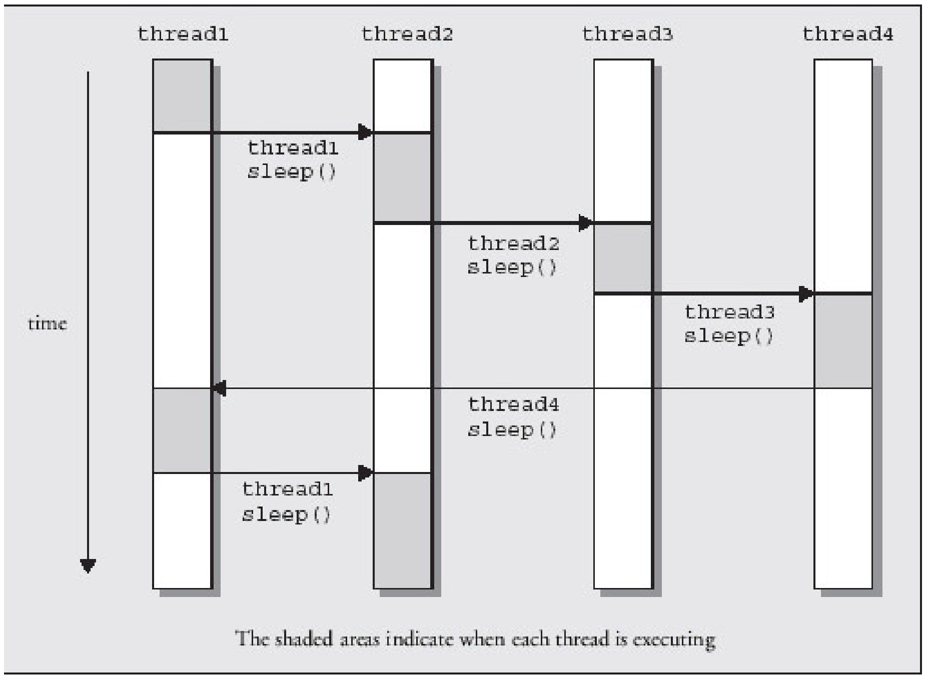

Figure C.1: Diagram of thread switching (source unknown)

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!