Android Development: Lecture Notes

Joel Ross

October 15, 2017

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 Autumn 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 cover the tools, programming languages, and architectures needed to develop applications for the Android platform.

This course expects you to have "journeyman"-level skills in Java (apprenticeship done, not yet master). It uses a number of intermediate concepts (like generics and inheritance) without much fanfare or explanation (though see the appendix). It also assumes some familiarity with developing interactive applications (e.g., client-side web 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. Visit us on GitHub to contribute improvements.

Part I

Lectures

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 8 Plus (2017) has: ~2.54Ghz hex-core A11 Bionic 64bit; 3GB RAM; 5.5" at 1920x1080 display.

  (As of 2016, this program has been superseded 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.

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!

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
Feb 2011	3.0	Honeycomb	11

Date	Version	Nickname	API Level
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
Aug 2017	8.0	Oreo	26

Each different "version" is nicknamed after a dessert, in alphabetical 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, as well as to divorce the OS from hardware requirements through the new Project Treble.
- 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! Android applications must be written for **heterogeneous devices**.

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 themselves 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 US Supreme Court in 2015, who refused to hear the case. It then went back to the the district court, which ruled in 2016 that Google's use of the API was fair use. This ruling is again under appeal. 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 Android Nougat and later 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 you as a developer, 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; as of this writing, 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 Building Apps

While Android applications can be developed using any programming environment, the official and best IDE for Android programming is **Android Studio**. This is a fork of JetBrain's IntelliJ IDEA application—a Java IDE customized for Android development. You will need to download and install this IDE.

- Be sure to download the Android Studio bundle that includes the **Android SDK** (Standard Development Kit): the tools and libraries needed for Android development. In particular, 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 runs the Android emulator: a virtual machine of an Android device.

  I recommend making sure that you have the SDK tools (the `tools` and `platform-tools` folder) available on your computer's `PATH` so you can use them from the command-line. By default, the SDK is found at `/Users/$USER/Library/Android/sdk` on a Mac, and at `C:\Users\$USERNAME\AppData\Local\Android\sdk` on Windows. While these tools are all built into the IDE, they can be useful fallbacks for debugging or automation.

Creating a Project

To begin your first application, launch Android Studio (it may take a few minutes to open). From the Welcome screen, choose to "Start a new Android Studio Project". This will open up a wizard to walk you through setting up the project.

- The "Company domain" should be a unique domain for you. For this course, you should include your UW NetID, e.g., `joelross.uw.edu`.
- Make a mental note of the project location so you can find your work later (e.g., if it's in `Desktop` or `Documents`).
- On the next screen, you will need to pick the *Minimum SDK* level that you wish to support—that is, what is the oldest version of Android your application will be able to run on? For this course, unless otherwise specified, you should target API 15 Ice Cream Sandwich (4.0.3) as a minimum, allowing your application to run on pretty much any Android device.

  Note that the Minimum SDK is different than the **Target SDK**, which is the version of Android your application has been tested and designed against. The Target SDK indicates what set of API features you have considered/coded against, even if your app can fall back to older devices that don't include those features. In many ways, the Target SDK is the "highest SDK I've worked with". For most of this course we will target either API 21 (Lollipop) or API 23 (Marshmallow).
- On the next screen, select to start with an *Empty Activity*. **Activities** are the basic component of Android, each of which acts as a "screen" or "page" in your app. Activities are discussed in more detail in the next lecture.
- Stick with the default name (`MainActivity`) on the next screen, and hit "Finish". Android Studio will take a few minutes to create your project and get everything set up. (Keep an eye on the bottom status bar to wait for everything to be finished). Once it is done, you have a complete (if simple) app!

Running the App

You can run your app by clicking the "Play" or "Run" button at the top of the IDE. But you'll need an Android Device to run the app on… luckily, Android Studio comes with one: a virtual Android Emulator. This virtual machine models emulates a generic device with hardware you can specify, though 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—it runs very, very slowly. The best way to speed the emulator up on any operating system is to 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.

  You can usually install this through Android Studio: go to `Tools > Android > SDK Manager` to open up the SDK manager for downloading different versions of the Android SDK and other support software. Under "SDK Tools", find "Intel x86 Emulator Accelerator (HAXM installer)", check it, and hit "OK" to download. Note that you may need to do additional installation/configuration manually, see the guides (Mac, Windows).

- It is of course also possible to run your app on a physical device. These are the best for development (they are the fastest, easiest way to test code), though you'll need a USB cable to be able to wire your device to your computer. Any device will work for this course; you don't even need cellular service (just WiFi should work).

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

In order to create an emulator for your machine, go to `Tools > Android > AVD Manager` to open up the *Android Virtual Device* Manager. You can then choose "Create Virtual Device…" in order to launch the wizard to specify a new emulator.

- The **Nexus 5** is a good choice of hardware profile for making sure you support "older" devices. The Nexus 5X or Pixel are also reasonable device profiles to test against.

- For now, you'll want to use a system image for Lollipop API 21 or 22, and almost certainly on x86 (Intel) hardware. Make sure to select one that includes the Google APIs (so you have access to special Google classes).

- The advanced settings can be used to specify things like the camera and whether it accepts keyboard input (should be on by default). These settings can always be changed later.

After the emulator boots, you can slide to unlock it… and your app should be loaded and started shortly thereafter!

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.

1.3 App Source Code

Android Studio will create a bunch of project files by default—almost all of which are use for something. By default, it will show your project using the **Android** view, which organizes the files thematically. If you instead change to the **Project** view you can see what the actual file system looks like (though we'll usually stick with the Android view).

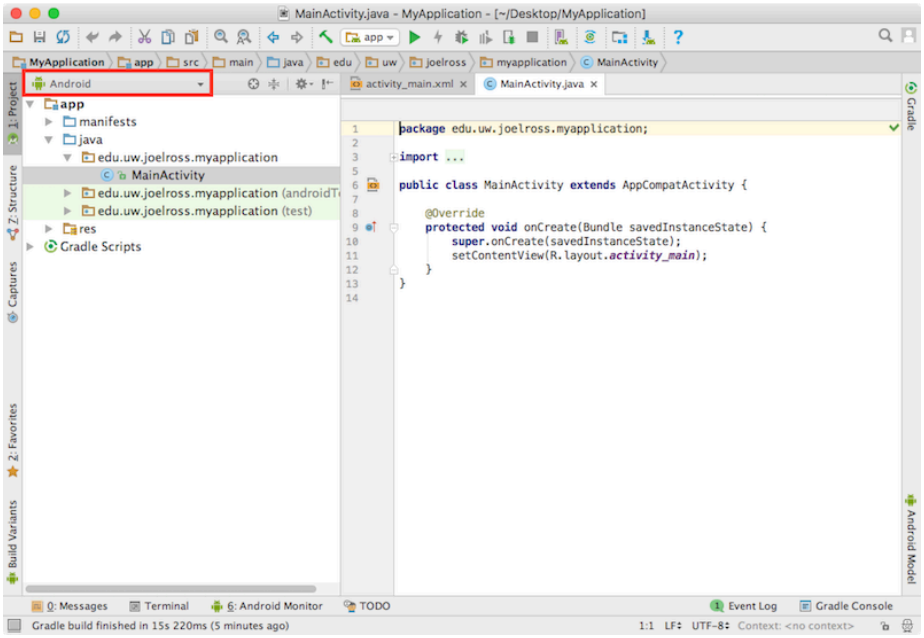

Figure 1.1: Android Studio. The "view" chooser is marked in red.

In the Android view, files are organized as follows:

- `app/` folder contains our application source code
  - `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. This is where the "logic" of the application goes
  - `res/` contains **XML resource** files used in the app. This is where we will put layout/appearance information

- `Gradle Scripts` contains scripts for the Gradle build tool, which is used to help compile the source code for installation on an device.

Each of these components will be discussed in more detail below.

XML Resources

The `res/` folder contains **resource** files. Resource files are used to define the *user interface* and other media assets (images, etc). for the application. Using separate files to define the application's interface than those used for the application's logic (the Java code) helps keep appearance and behavior separated. To compare to web programming: the resources contain the HTML/CSS content, while the Java code will contain what would normally by written in JavaScript.

The vast majority of resource files are specified in **XML** (E**X**tensible **M**arkup **L**anguage). XML has the exact same syntax as HTML, but you get to make up your own tags what whatever semantic values you want. Except we'll be using the tags that Android made up and provided: so defining an Android application interface will be a lot like defining a web page, but with a new set of elements. Note that this course expects you to have some familiarity with HTML or XML, but if not you should be able to infer the syntactical structure from the examples.

There are a large number of different kinds of resources, which are organized into different folders:

- `res/drawable/`: contains graphics (PNG, JPEG, etc) that will be "drawn" on the screen
- `res/layout/`: contains user interface XML layout files for the app's content
- `res/mipmap/`: contains launcher icon files in different resolutions to support different devices
- `res/values/`: contains XML definitions for general constants

There are other kinds of resources as well: see Available Resources or Lecture 2 for details.

The most common resource you'll work out are the **layout** resources, which are XML files that specify the visual layout of the component (like the HTML for a web page).

If you open a layout file (e.g., `activity_main.xml`) in Android Studio, by default it will be shown in a "Design" view. This view lets you use a graphical system to lay out your application, similar to what you might do with a PowerPoint slide. *Click the "Text" tab at the bottom to switch to the XML code view.*

- Using the design view is frowned upon by many developers for historical resources, even as it becomes more powerful with successive versions of

Android Studio. It's often cleaner and more effective to write out the layouts and content in direct XML 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 seen as less "professional". This course will focus on the XML code for creating layouts, rather than utilizing the design tool. See here for more on its features.

In the code view, you can see the XML: tags, attributes, values. Elements are nested inside one another. The provided XML code defines a layout (a `<android.support.constraint.ConstraintLayout>`) to organize things, and inside that is a `<TextView>` (a View representing some text).

- Note that most of the element attributes are **namespaced**, e.g. with an `android:` prefix, to avoid any potential conflicts (so we know we're talking about Android's `text` instead of something else).

The `android:text` attribute of the `<TextView>` contains some text. You can change that and *re-run the app* to see it update!

You will be able to specify what your app looks like by creating these XML layout files. For example, try replacing the `<TextView>` with a `<Button>`:

```
<Button
     android:id="@+id/my_button"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Click Me!"
     />
```

This XML defines a Button. The `android:text` attribute in this case specifies what text is shown on the button. Lecture 2 will describe in more detail the meaning of the other attributes, but you should be able to make a pretty good educated guess based on the names.

- (You can keep the `app:` scoped attributes if you want the button to stay in the center of the screen. Positioning will be discussed in Lecture 2).

The Manifest

Besides *resource files*, the other XML you may need to edit is the **Manifest File** `AndroidManifest.xml`, found in the `manifest/` folder in the Android project view. The Manifest acts like a "configuration" file for the application, specifying application-level details such as the app's name, icon, and permissions.

For example, you can change the displayed name of the app by modifying the `android:label` attribute of the `<application>` element. By default, the label is a **reference** to *another resource* found in the `res/values/strings.xml` file,

which contains definitions for string "constants". Ideally all user-facing strings—including things like button text—should be defined as these constants.

You will usually need to make at least one change to the Manifest for each app (e.g., tweaking the display name), so you should be familiar with it.

Java Activities

Besides using XML for specifying layouts, Android applications are written in **Java**, with the source code found in the `java/` folder in the Android project view (in a nested folder structure based on your app's package name). The Java code handles program control and logic, as well as data storage and manipulation.

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).

So while you can and will implement "normal" Java classes and models in your code, you will most frequently be utilizing classes a specific set of classes required by the framework, giving Android applications a common structure.

The most basic component in an Android program is an **Activity**, which represents a single screen in the app (see Lecture 3 for more details). The default provided `MainActivity` class is an example of this: the class *extends* `Activity` (actually it extends a subclass that supports Material Design components), allowing you to make your own customizations to the app's behavior within the Android framework.

In this class, we *override* the inherited `onCreate()` method that is called by the framework when the Activity starts—this method thus acts a little bit like the constructor for a class (though see Lecture 3 for a more nuanced discussion).

We call the super method to make sure the framework does it stuff, and then `setContentView()` to specify what the content (appearance) of the Activity should be. 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. Thus `R.layout.activity_main` refers to the `activity_main` layout found in the `res/layouts/` folder. That is how Android knows what layout file to show on the screen.

Dalvik

On a desktop, Java code needs to be compiled into bytecode and runs on a virtual machine (the Java Virtual Machine (JVM)). *Pre-Lollipop (5.0)*, Android code ran on a a virtual machine called **Dalvik**.

- 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 is stored in `.dex` or `.odex` ("[Optimized] Dalvik Executable") files, which is what was loaded onto the device. The process of converting from Java code to `dex` files is called **"dexing"** (so code that has been compiled and converted is called "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 native code is faster because no translation step is needed to talk to the actual hardware (via the OS).

However, *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 *at 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—and since you only install an application once, this is a pretty good trade.

After being built, an Android application (the source, dexed bytecode, and any non-code resources such as images) are packaged into an **`.apk`** file. This are basically zip files (it uses the same gzip compression); if you rename the file to be `.zip` and you can uncompress it! The `.apk` file is then cryptographically signed to specify its authenticity, and either "side-loaded" onto the device or uploaded to an App Store for deployment.

- In short: the signed `.apk` file is basically the "executable" version of your program!
- Note that the Android application framework code (e.g., the base `Activity` class) is actually "pre-DEXed" (pre-compiled) on the device; when you write code, you're 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.
- Usefully, since Android code is written for a virtual machine anyway, An-

droid apps can be developed and built on any computer's operating system (unlike some other mobile OS…).

Gradle Scripts

To summarize, after writing your Java and XML source code, in order to "build" and run your app you need to:

1. Generate Java source files (e.g., `R`) from the XML resource files
2. Compile the Java code into JVM bytecode
3. "dex" the JVM bytecode into Dalvik bytecode
4. Pack code and other assets into an `.apk`
5. Cryptographically sign the `.apk` file to authorize it
6. Transfer the `.apk` to your device, install, and run it!

This is a lot of steps! Luckily the IDE handles it for us using an *automated build tool* called **Gradle**. Such tools let you, in effect, specify a single command that will do all of these steps at once.

It is possible to customize the build script by modifying the Gradle script files, found in the `Gradle Scripts` folder in the Android project view. There are a lot of these by default:

- `build.gradle`: Top-level Gradle build; project-level (for building!)
- `app/build.gradle`: Gradle build specific to the app. **Use this one to customize your project!**, such as for adding dependencies or external libraries.
  - For example, 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 older Android applications were developed using Apache ANT. The build script was stored in the `build.xml` file, with `build.properties` and `local.properties` containing global and local build settings. While Gradle is more common these days, you should be aware of ANT for legacy purposes.

It is also possible to use Gradle to build and install your app from the command-line if you want. You'll need to make sure that you have a device (either physical or virtual) connected and running. Then from inside the project folder, you can build and install your app with

```
# use the provided Gradle wrapper to run the `installDebug` script
./gradlew installDebug
```

You can also launch the app from the command-line with the command

```
# use adb to start
adb shell am start -n package.name/.ActivityName
```

You can run both these commands in sequence by connecting them with an `&&` (which short-circuits, so it will only launch if the build was successful).

1.4 Logging & ADB

In Android, we can't use `System.out.println()` because we don't actually have a console 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, because people make mistakes).

To perform this logging, we'll use the `android.util.Log`[1] 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`.

- You will need to **`import`** the `Log` class!

  You can have Android Studio automatically add the `import` for a class by selecting that class name and hitting `alt-return` (you will be prompted if the class name is ambiguous). For better results, turn on *"Add unambiguous imports on the fly"* in the IDE Preferences.

The device's log file is stored persistently… 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!

Log Methods

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

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

- **`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 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";
```

Logcat

You can view the logs via `adb` (the debugging bridge) and a service called `Logcat` (from "log" and "conCATenation", since it concatenates 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.

- Something else to test: Cause the app to throw a runtime `Exception`! 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?*

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.

Toast

Logs are great for debugging output, but remember that they are only visible for *developers* (you need to have your phone plugged into the IDE or SDK!) If you want to produce an error or warning message for the *user*, you need to use a different technique.

One 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):

```
Toast toast = Toast.makeText(this, "Hello toast!", Toast.LENGTH_SHORT);
toast.show();

//as one line. Don't forget to show()!
Toast.makeText(this, "Hello toast!", Toast.LENGTH_SHORT).show();
```

Toasts are created by using the `Toast.makeText()` factory method (instead of calling a constructor). This method takes three parameters: the *Context*, or what is producing the Toast (see Chapter 3), the text to display, and an `int` constant representing the length the Toast should appear.

Toasts are intended to be a way to provide information to the user (e.g., giving them quick feedback), but they 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).

1.5 Adding Interaction

Finally, we've created a button and discussed how to show visual information to the user… so let's hook those together!

As with JavaScript, in order to have our button do something, we need to register a *callback function* that can be executed when the button is clicked. In Java, these callback functions are supplied by "listener" objects who can respond to *events* (see Appendix B for a more detailed discussion).

First, we need to get access to a variable that represents the `Button` we defined in the XML—similar to what you do with `document.getElementById()` in JavaScript. The method to access an element in Android is called `findViewById()`, and can be called directly on the `Activity`:

```
Button button = (Button)findViewById(R.id.my_button);
```

As an argument, we pass in a value defined in the *auto-generated* `R` class that represents the button's `id` value—this is based on what we put in the `<Button>`'s `android:id` attribute. The exact format is discussed in Lecture 2.

- Note that the method returns a `View` (a superclass of `Button`), so we almost always *typecast* the result. See Lecture 2 for more on the `View` class.

We can then register a listener (callback) by calling the `setOnClickListener()` method and passing in an **anonymous class** to act as the listener:

```
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Perform action on click
    }
});
```

Tab completion is your friend!! Try just typing the `button.`, and then selecting the method name from the provided list. Similarly, you can begin to type `new OnClick` and then tab-complete the rest of the class definition. The Android Studio IDE makes this ubiquitous boilerplate code easy to produce.

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

Chapter 2

Resources and Layouts

This lecture discusses **Resources**, which are used to represent elements or data that are separate from the behavior (functional logic) of an app. In particular, this lecture focuses on how resources are used to define **Layouts** for user interfaces. This lecture focuses on the XML-based source code in an Android app; the Activities lecture begins to detail the source code written in Java.

This lecture references code found at https://github.com/info448/lecture02-layouts.

2.1 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), usually defined in XML files. Resources represent components that are *separate* from the app's behavior, so are 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 "design" tab in Android Studio). 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!

What should be a resource? In general:

- Layouts should **always** be defined as resources
- UI controls (buttons, etc) should *mostly* be defined as resources (they are part of layouts), though behavior will be defined programmatically in Java
- Any graphic images (drawables) should be defined as resources
- Any *user-facing* strings should be defined as resources
- Style and theming information should be defined as resources

As introduced in Lecture 1, there are a number of different resource types used in Android, and 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/`: contains launcher icon files in different resolutions
- `res/values/`: contains XML definitions for general constants, which can include:
  - `/strings.xml`: short string constants (e.g., button labels)
  - `/colors.xml`: color constants
  - `/styles.xml` : constants for style and theme details
  - `/dimen.xml` : 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.

R

Resources are usually defined as XML (which is similar in syntax to HTML). When an application is compiled, the build tools (e.g., Gradle) will **generate** an additional Java class called `R` (for "resource"). This class contains what is basically a giant list of static "constants"—at least one for each XML element.

For example, consider the `strings.xml` resource, which is used to define String constants. The provided `strings.xml` defines two constants of type `<string>`. The `name` attribute specifies the name that the variable will take, and the content of the element gives that variable's value. Thus

```
<string name="app_name">Layout Demo</string>
<string name="greeting">Hello Android!</string>
```

will in effect be compiled into constants similar to:

```
public static final String app_name = "My Application";
public static final String greeting = "Hello Android!";
```

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

All of the resource constants are compiled into *inner classes* inside `R`, one for each resource type. So an `R` file containing the above strings would be structured like:

```
public class R {
    public static class string {
        public static final String app_name = "My Application";
        public static final String greeting = "Hello Android!";
    }
}
```

This allows you to use **dot notation** to refer to each resource based on its type (e.g., `R.string.greeting`)—similar to the syntax used to refer to nested JSON objects!

- For most resources, the identifier is defined as an element attribute (`name` attribute for values like Strings; `id` for specific View elements in layouts). For more complex resources such as entire layouts or drawables, the identifier is the *filename* (without the file extension): for example `R.layout.activity_main` refers to the root element of the `layout/activity_main.xml` file.
- More generally, each resource can be referred to with `[(package_name).]R.resource_type.identifier`.
- Note that the file name `string.xml` is just a convention for readability; all children of a `<resource>` element are compiled into `R` dependent on their type, not their source code location. So it is possible to have lots of different resource files, depending on your needs. The `robot_list.xml` file is not a standard resource.

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

If you actually open the `R.java` file, you'll see that the static constants are actually just **`int`** values that are *pointers* to element references (similar to passing a `pointer*` around in the C language); the content of the value is stored elsewhere (so it can be adjusted at runtime; see below). This does mean that in our Java code we usually work with `int` as the data type for XML resources such as Strings, because we're actually working with pointers *to* those resources.

- For example, the `setContentView()` call in an Activity's `onCreate()` takes in a resource `int`.
- 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 fast).

Because the `R` class is included in the Java, we can access these `int` constants directly in our code (as `R.resource_type.identifier`), as in the `setContentView()` method. However, if you want to actually get the String value, you can look that up by using the application's `Resources()` object:

```
Resources res = this.getResources(); //get access to application's resources
String myString = res.getString(R.string.myString); //look up value of that resource
```

- The other comment method that utilizes resources will be `findViewById(int)`, which is used to reference a `View` element (e.g., a button) specified in a layout resource in order to call methods on it in Java, as in the example from the previous 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`.

It is also possible to reference one resource from another within the XML using the `@` symbol, following the schema `@[<package_name>:]<resource_type>/<resource_name>`. For example, in the Manifest you can see that the application's label is referred to via `@string/app_name`.

- You 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 XML attribute. This is most commonly used with the `android:id` attribute (`android:id="@+id/identifier"`) to create a variable referring to that View; see below for details.

Alternative Resources

One main advantage to separating resources from the Java code is that it allows them to be **localized** and changed depending on the device! Android allows the developer to specify folders for "alternative" resources, such as for different languages or device screen resolutions. **At runtime**, Android will check the configuration of the device, and try to find an alternative resource that matches that configuration. 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`)

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

- Screen orientation (`port` for portrait, `land` for landscape)
- Specific 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 *relative* to the device size!
- Platform version (`v1`, `v4`, `v7`… for each API number)

Configurations are indicated using the **directory name**, giving folders the form `<resource_name>(-<config_qualifier>)+`. For example, the `values-fr/` would contain constant values for devices with a French language configuration.

- Importantly, the resource file itself should to be the *same* for both the qualifier and unqualified resource name (e.g., `values/strings.xml` and `values-fr/strings.xml`). This is because Android will load the file inside the qualified resource if it matches the device's configuration *in place of* the "default" unqualified resource. The names need to be the same so one can replace the other!

- You can see this in action by using the *New Resource* wizard (`File > New > Android resource file`) to create a string resource (such as for the `app_name`) in another language. Change the device's language settings (via the device's `Settings > Language & Input > Language`) to see the content automatically adjust!

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

- You can view the directory structure that supports this by switching to the `Package` project view in Android Studio.

2.2 Views

The most common type of element you'll define as a resource are **Views**[3]. `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 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).

[3] http://developer.android.com/reference/android/view/View.html

Here's the big trick: one subclass of `View` is `ViewGroup`[4]. A `ViewGroup` is a View 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.

- Like the HTML DOM, Android Views are thus structured into a *tree*, what is known as the **View hierarchy**.

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, a `Layout` is simply a `ViewGroup` that provide "ordering" and "positioning" information for the Views inside of it. Layouts 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).

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`.

All View have **properties** which define the state of the View. Properties are usually specified 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.

  The `@+` syntax is used to define a *new* View `id` resource—almost like you are declaring a variable inside the element attribute! You will need to use the `@+` whenever you specify a new `id`, which will allow it to be

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

referenced either from the Java code (as `R.id.identifier`) or by other XML resources (as `@id/identifier`).

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 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 are 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 scaled 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 (best practice: reference a color resource!), etc.

- There are lots of other properties as well! You can see a listing of generic properties in the `View`[5] 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: you're effectively specifying an inline `style` attribute for that element, and one that won't affect child elements. In order to define shared style properties, you'll need to use styles resources, which are discussed in a later lecture.

Views and Java

Displaying a View on a screen is called **inflating** that View. The process is called "inflating" based on the idea that it is "unpacking" or "expanding" a compact resource description into a complex Java Object. When a View is inflated, it is instantiate as an object: the inflation process changes the `<Button>` XML into a `new Button()` object in Java, with the property attributes passed as a parameter to that constructor. Thus you can think of each XML element as representing a particular Java Object that will be instantiated and referenced at runtime.

- This is almost exactly like how JSX components in React are individual objects!
- Remember that you can get a reference to these objects from the Java code using the `findViewById()` method.

Once you have a reference to a View object in Java, it is possible to specify visual properties dynamically via Java methods (e.g., `setText()`, `setPadding()`). However, 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 dynamically replace one layout resource with another (see below).

- Views also have inspection methods such as `isVisible()` and `hasFocus()` if you need to check the View's state.

DO NOT instantiate or define Views or View appearances in an Activity's `onCreate()` method, unless the properties (e.g., content) truly cannot be determined before runtime! **DO** specify layouts in the XML instead.

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

Practice

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

You should specify the content of the image in the XML resource using the `android:src` attribute (use `@` to reference a `drawable`), but you can specify the content dynamically in Java code if you want to change it later.

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

2.3 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 structure the elements on the screen. 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.

LinearLayout

Probably the simplest Layout to understand is the `LinearLayout`. This Layout orders the children Views 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 such as padding or background color; the support of the attributes is inherited! (But remember that the properties themselves are not inherited by child elements: you can't set the `textSize` for a Layout and have it apply to all child Views).

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 default elements 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!

- This is a similar behavior to the `flex-grow` property in the CSS Flexbox framework.

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 declared for individual child Views to state where they are positioned; the `android:gravity` property specifies where the content of an element should be aligned.

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 LinearLayout containing "rows" of horizontal LinearLayouts (which contain Views). As with HTML, there are lots of different options for achieving any particular interface layout.

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):

```
<TextView
    android:id="@+id/first"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="FirstString" />
<TextView
    android:id="@+id/second"
    android:layout_height="wrap_content"
    android:layout_below="@id/first"
    android:layout_alignParentLeft="true"
    android:text="SecondString" />
```

You 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!)

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.

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 documentation 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 GridView, 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.
- CoordinatorLayout is a class provided as part of an extra support library, and provides support for Material Design widgets and animations. See Lecture 5 for more details.

Combining and Inflating Layouts

It is possible to combine multiple layout resources files. 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. 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 the `R` class)
- The second parameter is a `ViewGroup` to act as the "parent" for this View—e.g., what layout should the View be inflated 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 parameters to use). If not assigning to parent on inflation, you can later attach the View using methods in `ViewGroup` (e.g., `addView(View)`).

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 explicit inflation 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 isn't as common in modern development. A much cleaner solution is to use a `ViewStub`[6]. 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"
```

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

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

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

There are many other options for displaying or changing View content. Just remember to define as much of the View as possible in the XML, so that the Java code is kept simple and separate.

2.4 Inputs

So far we you have used some basic Views such as `TextView`, `ImageView`, and `Button`.

A `Button` is an example of an Input Control. These are simple (read 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.

You can change the lecture code's `MainActivity` to show a 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 `strings.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 1. This involves *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 will 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:

```
<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". Specifying 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 in 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 the layout changes).

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.

Chapter 3

Activities

This lecture introduces **Activities**, which are the basic component used in Android applications. Activities provide a framework for the Java code that allows the user to interact with the layouts defined in the resources.

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

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

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

3.1 Making Activities

We specify an Activity for an app by *subclassing* (extending) the framework's `Activity` class. We use **inheritance** to make a specialized type of `Activity`. By extending this class, we inherit all of the methods that are needed to control how the Android OS interacts with the Activity—behaviors like showing the screen, allowing Activities to change, and closing the Activity when it is no longer being used.

If you look at the default Empty `MainActivity`, it actually subclasses `AppCompatActivity`, which is itself a already a specialized subclass 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 to import a class in Android Studio is `alt+return`, or you can do it by hand (look up the package in the documentation)! I recommend that you change the IDE's preferences 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 of the available classes have been deprecated in favor of **Fragments**, which are sort of like "sub-activities" that get nested in larger Activities. Fragments will be discussed in a later lecture.

3.2 The Activity Lifecycle

An important point to note: does this Activity have a **constructor** that we call? *No!* We never write code that **instantiates** our Activity (that is: 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.

Although we never call a constructor or `main()` method, Activities do have an very 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**, just like how you specified the `onClick()` method to react to a button press. We can *override* these lifecycle callbacks 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

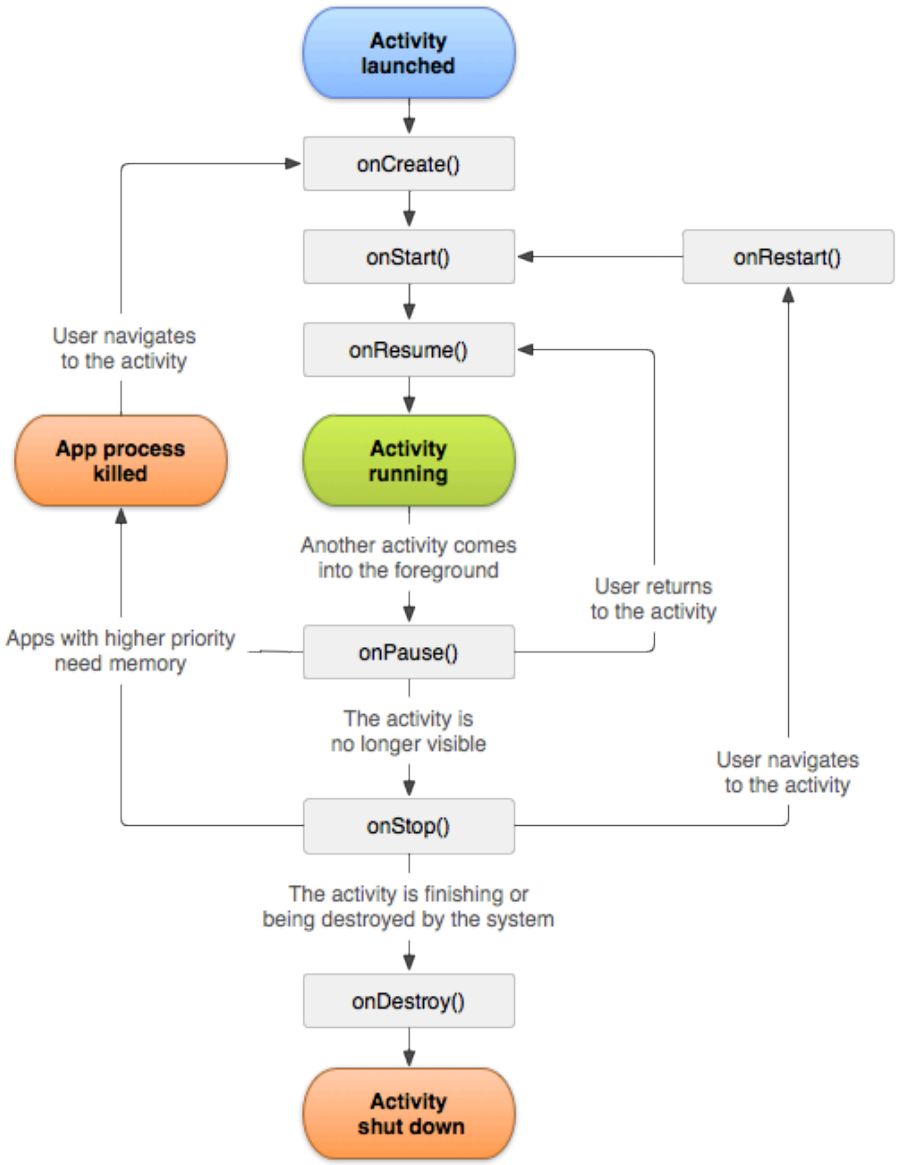

Figure 3.1: Lifecycle state diagram, from Google[3]. See also an alternative, simplified diagram.

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 which layout to use), and otherwise do the kinds of work that 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 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). "onFocus" would have been a better name for this callback.
- **`onPause()`**: called when the system is about to start another Activity (so this one is 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 closed (and so is 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 also possibly because the current 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 lifecycle 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 Activity on its own.

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

they are more reliably executed.

Activities are *also* destroyed (and recreated) when the device's configuration changes—such as if you rotate the phone!

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 (shut down). Note that this destruction is unpredictable, as the "necessity" of an app being open is dependent on the OS's resource allocation rules.

Thus in practice, you should implement Activities as if they could be destroyed at any moment—you cannot rely on them to continue running if they are not visible.

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.

Overriding the Callback Methods

When you create an new Empty `MainActivity`, the `onCreate()` callback has already been overridden for you, 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 "bundling" *small* amounts of information. See before 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). Again, the IDE's auto-complete feature lets you just type the name of the callback and get the whole method signature for free!

We can quickly add in the event callbacks and `Log.v()` calls to confirm that they are executed. Then you can use the phone to see them occur:

- `onCreate()`, `onStart()` and `onResume()` are called when the app is instantiated.
- You can `onPause()` the Activity by dragging down the notification drawer from the top of the screen.

- You can `onStop()` the Activity by going back to the home screen (click the circle at the bottom).
- You can `onDestroy()` the Activity by changing the configuration and rotating the phone: click the "rotate" button on the emulator's toolbar.

Saving and Restoring Activity State

As mentioned above, an Activity's `onCreate()` method takes in a `Bundle` as a parameter. This `Bundle` is used to store information about the Activity's current state, so that if the Activity is destroyed and recreated (e.g., when the phone is rotated), it can restored in the same state and the user won't lose any data.

For example, the `Bundle` can store state information for View elements, such as what text a user has typed into an `EditText`. That way when the user rotates their phone, they won't lose the form input they've entered! If a View has been given an `android:id` attribute, then that id is used to *automatically* save the state of that View, with no further effort needed on your own. So you should always give input Views ids!

You can also add your own custom information to the `Bundle` by overriding the Activity's `onSaveInstanceState()` callback (use the one for `AppCompactActivity` that only takes one parameter). It takes as a parameter the `Bundle` that is being constructed with the saved data: you can add more information to this Bundle using an appropriate `put()` method (similar to the method used for Maps, but type-sensitive):

```
//declare map key as a constant
private static final String MSG_KEY = "message_key";

@Override
protected void onSaveInstanceState(Bundle outState) {
    //put value "Hello World" in bundle with specified key
    outState.putString(MSG_KEY, "Hello World");
    super.onSaveInstanceState(outState);
}
```

- Note that you should always declare Bundle keys as *CONSTANTS* to help with readability/modifiability and to catch typos.
- Be sure to always call `super.onSaveInstanceState()` so that the super class can do its work to save the View hierarchy's state! In fact, the reason that Views "automatically" save their state is because this method is calling their own `onSaveInstanceState()` callback.

You can access this saved Bundle from the Activity's `onCreate()` method when the Activity is recreated. Note that if the Activity is being created for the *first*

time, then the Bundle will be `null`—checking for a null value is thus a good way to check if the Activity is being recreated or not:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if(savedInstanceState != null){ //Activity has been recreated
        String msg = savedInstanceState.getString(MSG_KEY);
    }
    else { //Activity created for first time

    }
}
```

Remember that a `Bundle` can only hold a *small* amount of primitive data: just a couple of numbers or Strings. For more complex data, you'll need to use the alternative data storage solutions discussed in later lectures.

3.3 Context

If you look at the documentation for the `Activity` class, you'll notice that it is itself a subclass of something called a **`Context`**[4]. `Context` is an **abstract class** 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. 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. For example, the `getResources()` method discussed in the last chapter is a method of the `Context` class, because we need to have some way of saying *which* set of resources to load!

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).

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

- 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>` elements 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. 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—like `getResources()`—called as undecorated methods (without an explicit `this`).

You'll need to refer to the Context whenever you want to do something beyond the Activity you're working with: whether that's accessing resources, showing a Toast (the first parameter to `Toast.makeText()` is a `Context`), or opening another Activity.

3.4 Multiple Activities

The whole point of interfacing with the Activity Lifecycle is to 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 lifecycle 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 should edit this Activity's layout resource so that the `<TextView>` displays an appropriate message.

Importantly, for every Activity you make, an entry gets added to the **Manifest** file `AndroidManifest.xml`. This file acts like the *"table of contents"* for our application, giving information about what your app looks (that is, what Activities it has) like so that the OS can open appropriate Activities as needed. (If you create an Activity's `.java` file manually, you will need to add this entry manually as well).

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).

Intents

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 that other Activity).

- 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 letters you'd send through the mail: 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 constructors, but the one we'll start with looks like:

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

The first parameter is the `Context` by which this Intent will be delivered (e.g., `this`). Note that we use the fully qualified `MainActivity.this` to indicate that we're not talking about the anonymous event handler 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.

After having instantiated the `Intent`, we can use that message to start an Activity by calling the `startActivity()` method (inherited from `Activity`), and 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 in a later lecture; 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!

Back & Tasks

We've shown that we can have lots of Activities (and of course many more can exist across 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 abstract 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.

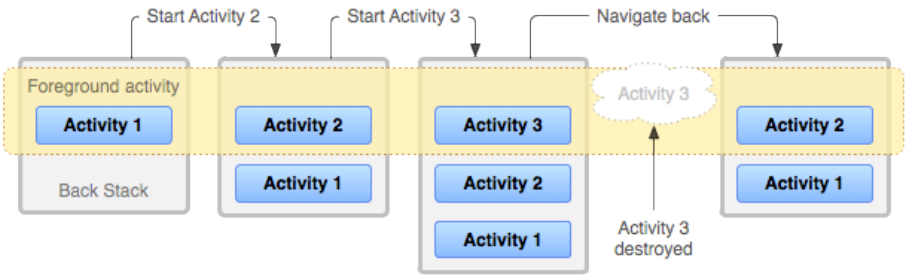

Figure 3.2: An example of the Activity stack, from Google[5].

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

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

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"—then launching an application 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 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 in a web browser, with the "back stack" being the history of web pages visited within 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 Managing Tasks

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?

```
//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 compatibility for API level 15 (since the `android:parentActivityName` attribute is only defined for API level 16+).

Chapter 4

Data-Driven Views

Lecture 3 discussed how to use Views to display content and support user interaction. This lecture extends those concepts and presents techniques for creating **data-driven views**—views that can *dynamically* present a data model in the form of a *scrollable list*. It also explains how to access data on the web using the Volley library. Overall, this process demonstrates a common way to connect the user interface for the app (defined as XML) with logic and data controls (defined in Java), following the **Model-View-Controller** architecture found throughout the Android framework.

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

4.1 ListViews and Adapters

In particular, this lecture discussed how to utilize a ListView[1], 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 re-inflating 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.

[1] https://developer.android.com/guide/topics/ui/layout/listview.html

- Note that a more advanced and flexible version of this behavior is offered by the `RecyclerView` class, which works in mostly the same way but requires a few extra steps to set up. See also this guide for more details.

The `ListView` control uses a **Model-View-Controller (MVC)** architecture. This is a design 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 (just as a controller should).

- 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 normally should define such hard-coded 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).

   We don't really need to specify a full Layout (though we could if we wanted): just a basic `TextView` will suffice. Have the width `match_parent` and the height `wrap_content`. *Don't forget an* `id`*!*

```
<!-- need to include the XML namespace (xmlns) so the `android` namespace validates -->
<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. Add 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`[2] 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`!

```
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` to access resources, the layout resource to use for each item, the `TextView` within that layout (the target of the mapping), and the data array (the source of the mapping). Also note that this instance utilizes *generics*: 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.

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

And that's all that is needed to create a scrollable list of data! To track the process: the Adapter will go through each item in the *model*, and "translate" that item into the contents of a View. These Views will then be displayed in a scrollable list.

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

[2] https://developer.android.com/reference/android/widget/Adapter.html

- The `position` 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 you 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*!)

4.2 Networking with Volley

A list with hard-coded data isn't very useful. It would be better if that data could be accessed dynamically, such as downloaded from the Internet!

There are a couple of different ways to programmatically send network requests from an Android application. The "lowest level" is to utilize the `HttpURLConnection` API. With this API, you call methods to open a connection to a URL and then to send an HTTP Request to that location. The response is returned as an `InputStream`, which you need to "read" byte by byte in order to reconstruct the received data (e.g., to make it back into a String). See this example for details.

While this technique is effective, it can be tedious to implement. Moreover, downloading network data can take a while—and these network method calls are synchronous and **blocking**, so will prevent other code from running while it downloads—including code that enables the user interface! Such block will lead to the infamous *"Application not responding" (ANR)* error. While it is possible to send such requests asynchronously on a *background thread* to avoid blocking, that requires additional overhead work. See the Background Services Lecture for more details.

To solve these problems with less work, it can be be more effective to utilize an **external library** that lets us abstract away this process and just talk about making network requests and getting data back from them. (This is similar to how in web programming the `fetch` API abstracts the opaque `XMLHttpRequest` object). In particular, this lecture will introduce the **Volley** library, which is an external library developed and maintained by Google. It provides a number of benefits over a more "manual" approach, including handling multiple concurrent requests and enabling the caching of downloaded data. It also causes network requests to be handled asynchronously on a background thread without any additional effort!

Volley's main "competitor" is the Retrofit library produced by Square. While Retrofit is usually faster at processing downloaded data, Volley has built-in

support for handling images (which will be useful in the future), and has a slightly more straightforward interface.

Using Volley

Because Volley is an external library (it isn't built into the Android framework), you need to explicitly download and include it in your project. Luckily, we can use the *Gradle* build system to do this for us by listing Volley as a **dependency** for the project. Inside the *app-level* `build.gradle` file, add the following line inside the `dependencies` list:

```
compile 'com.android.volley:volley:1.0.0'
```

This will tell Android that it should download and include version `1.0.0` of the Volley library when it builds the app. Hit the "Sync" button to update and rebuild the project.

- External libraries will be built into your app, increasing the file size of the compiled `.apk` (there is more code!). Though this won't cause any problems for us, it's worth keeping in mind as you design new apps.

Once you have included Volley as a dependency, you will have access to the classes and API to use in your code.

In order to request data with Volley, you will need to instantiate a `Request` object based on the type of data you will be downloading: a `StringRequest` for downloading text data, a `JsonRequest` for downloading JSON formatted data, or an `ImageRequest` for downloading images.

The constructor for `StringRequest`, for example, takes 4 arguments:

1. A constant representing the HTTP method (verb) to use. E.g., `Request.Method.GET`
2. The URL to send the request to (as a String)
3. A `Response.Listener` object, which defines a callback function to be executed when the response is received.
4. A `Response.ErrorListener` object, which defines a callback function to be executed in case of an error.

Because the last two *listener* objects are usually defined with anonymous classes, this can make the Request constructor look more complicated than it is:

```
//silly example: get 20 random dinosaur names
String url = "http://dinoipsum.herokuapp.com/api/?format=text&words=20&paragraphs=1";

Request myRequest = new StringRequest(Request.Method.GET, url,
        new Response.Listener<String>() {
            public void onResponse(String response) {
                Log.v(TAG, response);
```

```
            }
        }, new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                Log.e(TAG, error.toString());
            }
        });
```

(Also note that the `Response.Listener` is a *generic* class, in which we specify the format we're expecting the response to come back in. This is `String` for a `StringRequest`, but would be e.g., `JSONObject` for a `JsonObjectRequest`).

In order to actually *send* this Request, you need a `RequestQueue`, which acts like a "dispatcher" and handles sending out the Requests on background threads and otherwise managing the network operations. We create a dispatcher with default parameters (for networking and caching) using the `Volley.newRequestQueue()` factory method:

```
RequestQueue requestQueue = Volley.newRequestQueue(getApplicationContext());
```

The factory method takes in a Context for managing the cache; the best practice is to use the application's Context so it isn't dependent on a single Activity.

Once you have a RequestQueue, you can add your request to that in order to "send" it:

```
requestQueue.add(myRequest);
```

If you test this code, you'll notice that it 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 `AndroidManifest.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) each time you use it. This process is discussed in the Files and Permissions Lecture. Using the Internet is

not a dangerous permission, so only requires the permission declaration in the Manifest.

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 prove we got it!

Of course, we'd like to display that data on the screen (rather than just log it out). That is, we want to put it into the ListView, meaning that we need to feed it back into the Adapter (which 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! Note that you may need to do data parsing on the response body, such as splitting a String or constructing an `array` or `ArrayList` out of JSON data.
- 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.

You can use the `JsonObjectRequest` class to download data as a JSON Object rather than a raw String. JSON Objects and Arrays can be converted into Java Objects/Arrays using two classes: `JSONObject` and `JSONArray`. The constructors for each of these classes take a JSON String, and you can call the `getJSONArray(key)` and `getJSONObject(key)` in order to get nested objects and arrays from inside a `JSONObject` or `JSONArray`.

RequestQueue Singletons

If you are going to make multiple network requests for your application (which you usually will for anything of a reasonable size), it is wasteful to repeatedly instantiate new `RequestQueue` objects—these can take up significant memory and step on each others toes.

Instead, the best practice is to use the Singleton Design Pattern to ensure that your entire application only uses a *single* RequestQueue.

To do this, you will want to create an entire class (e.g., `RequestSingleton`) that will only ever be instantiated once (it will be a "singleton"). Since the Volley `RequestQueue` is controlled by this singleton, it means there will only ever be one `RequestQueue`.

```
public class RequestSingleton { //make static if an inner class!

    //the single instance of this singleton
    private static RequestSingleton instance;
```

```
    private RequestQueue requestQueue = null; //the singleton's RequestQueue

    //private constructor; cannot instantiate directly
    private RequestSingleton(Context ctx){
        //create the requestQueue
        this.requestQueue = Volley.newRequestQueue(ctx.getApplicationContext());
    }

    //call this "factory" method to access the Singleton
    public static RequestSingleton getInstance(Context ctx) {
        //only create the singleton if it doesn't exist yet
        if(instance == null){
            instance = new RequestSingleton(ctx);
        }

        return instance; //return the singleton object
    }

    //get queue from singleton for direct action
    public RequestQueue getRequestQueue() {
        return this.requestQueue;
    }

    //convenience wrapper method
    public <T> void add(Request<T> req) {
        requestQueue.add(req);
    }
}
```

This structure will let you make multiple network requests from multiple components of your app, but without trying to have multiple "dispatchers" taking up memory.

Downloading Images

In addition to downloading text or JSON data via HTTP requests, Volley is also able to support downloading *images* to be shown in your app.

In general, handling images in Android is a difficult task. Images are large files (often multiple megabytes in size) that may require extensive and lingering data transfer to download and require processor-intensive decoding in order to be displayed. Since mobile devices are *resource constrained* (particularly in memory), trying to download and display lots of images—say in a scrollable list—can quickly cause problems. See Handling Bitmaps and Loading Large

Bitmaps Efficiently for some examples of the complexity needed to work with images.

Volley provides some support to make downloading and processing images easier. In particular, it provides built-in support for *network management* (so that data transfers are most efficiently optimized), *caching* (so you don't try to download the same image twice), and for easily *displaying* network-loaded images.

The other popular image-management libraries are Glide, Square's Picasso, and Facebook's Fresco. Google recommends using Glide for doing complex image work. However, if Volley's image loading is sufficient for your task, that allows you to only need to work with a single library and networking queue.

In order to effectively download images with Volley, you need to set up an `ImageLoader`. This object will handling downloading remote images as well as *caching* them for the future.

To instantiate an `ImageLoader`, you need to provide a `RequestQueue` as well as an `ImageCache` object that represents how image data should be cached (e.g., in memory, on disk, etc.). The Volley documentation suggests using a `LruCache` object for caching to memory (though you can use a `DiskBasedCache` as well):

```
//instantiate the image loader
//params are the requestQueue and the Cache
ImageLoader imageLoader = new ImageLoader(requestQueue,
        new ImageLoader.ImageCache() {  //define an anonymous Cache object
            //the cache instance variable
            private final LruCache<String, Bitmap> cache = new LruCache<String, Bitmap>(20);

            //method for accessing the cache
            @Override
            public Bitmap getBitmap(String url) {
                return cache.get(url);
            }

            //method for storing to the cache
            @Override
            public void putBitmap(String url, Bitmap bitmap) {
                cache.put(url, bitmap);
            }
        });
```

- It's a good idea to make this `ImageLoader` an instance variable of the `RequestSingleton`.

Once you have the `ImageLoader`, you can use it to download an image by calling it's `get()` method, specifying a callback listener that will be executed when the image is finished downloading.

However, you almost always want to download an image in order to display it. Volley makes this easy by providing a customized View called `NetworkImageView`. A `NetworkImageView` is able to handle the downloading of its source image on its own, integrating that process into the Activity's lifecycle (e.g., so it won't download when the View isn't displayed).

You declare a `NetworkImageView` in the layout XML in the same way you would specify an `ImageView`:

```
<com.android.volley.toolbox.NetworkImageView
    android:id="@+id/img_remote"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="fitXY"
    />
```

- The `android:scaleType` attribute indicates how the image should be scaled to fit the View.

In order to download an image into this View, you call the `setImageUrl()` method on the View from within your Java, specify the image url to load into the View and the `ImageLoader` to use for this network access:

```
NetworkImageView netView = (NetworkImageView)findViewById(R.id.img_remote);
netView.setImageUrl("https://ischool.uw.edu/fb-300x300.png", imageLoader);
```

And that will let you download and show images from the Internet (without needing to make them into a `drawable` resource)!

Chapter 5

Material Design

This lecture discusses Material design: the *design language* created by Google to support mobile user interfaces. The design language focuses on mobile designs (e.g., for Android), but can also be used across platforms (e.g., on the web through css frameworks).

This lecture references code found at https://github.com/info448/lecture05-material.

Material Design support was introduced in API 21 Lollipop, and so a device running that version of Android or later is required to access all the Material features supported by Android. However, most functionality is also available via compatibility libraries (i.e., `AppCompat`), and so can be utilized on older devices as well.

5.1 The Material Design Language

Material design is a *design language*: a "vocabulary" of visual and interactive patterns that are shared across systems. Material forms both a "look and feel" for Google applications and modern (API 21+) Android apps in general, as well as providing a

A video explanation of the Material language (via https://developer.android.com/design/material/index.html)

In summary, the Material Design language is based around three main principles:

- ***Material is the metaphor***. The Material design language is built around presenting user interfaces as being made of virtual **materials**, which ara paper-like surfaces floating in space. Each surface has a uniform thickness (`1dp`), but different *elevation* (conveyed primarily through

shadows and *perspective*). This physical metaphor helps to indicate different affordances and user interactions: for example, a button is "raised" and can be "pushed down".

- ***Motion provides meaning***. Material also places great emphasis on the use of **motion** of these materials in order to help describe the relationships between components as well as make design overall more "delightful". Material applications include lots of animations and flourishes, making the app's usage continuous and connected. Surfaces are able to change shape, size, and position (as long as they stay in their plane—they cannot fold, but they can split and rejoin) in response to user input.
- ***Bold, graphic, intentional***. Material applications follow a particular aesthetic in terms of things such as color (not muted), imagery (usually lots of it). Material applications *look* like material applications, though they can still be customized to meet your particular needs.

For more information on the Material design language, see the **official guidelines** (click the hamburger button on the left to navigate). This documentation contains extensive examples and instructions, ranging from suggested font sizes to widget usage advice.

This lecture focuses on ways to implement specific aspects of the Material Design language in Android applications, rather than on specifics of how to obey the language guidelines.

5.2 Material Styles & Icons

The first and easiest step towards making a Material-based application is to utilize the provided Material Themes in order to "skin" your application. These themes are available on API 21 Lollipop and later; for earlier operating system, you can instead use equivalent themes in `AppCompat` (which are in fact the default themes for new Android apps!)

Applying themes (including Material themes) are discussed in more detail in the Styles & Themes lab.

- You can see what specific properties are applied by these styles and themes by browsing the source code for the Android framework—check out the `styles_material.xml` and `themes_material.xml` (these will also reference values defined in the variable `color` and `dimens` resources). The `AppCompat` styles and themes can be source code for the `v7` support library.

In practice, the biggest part of utilizing a Material Theme is defining the color palette for your app. The Material design specification describes a broad color palette (with available swatches); however, it is often more useful to pick your

colors using the provided **color picker tool**[1], which allows you to easily experiment with different color combinations.

- Pick your *primary* and *secondary* color, and then assign the resource values `colorPrimary`, `colorPrimaryDark`, and `colorAccent` in `res/values/colors.xml`. This will let you easily "brand" your application.

Material-specific attributes such as `android:elevation` are also available in API 21+, though making shadows visible on arbitrary elements requires some additional work.

In addition to styles, Material also includes a large set of **icons** for use in applications. These icons supplement the built in `ic_*` drawables that are built into the platform and are available for use (and show up as auto-complete options in the IDE).

Instead these icons are available as vector drawables—rather than being a `.png` file, images are defined as using an XML schema similar to that used in Scalable Vector Graphics (SVG).

SVG is used to represent vector graphics—that is, pictures defined in terms of the connection between points (lines and other shapes), rather than in terms of the pixels being displayed (called a raster image). In order to be shown on a screen, these lines are converted (rastered) into grids of pixels based on the size and resolution of the display. This allows SVG images to "scale" or "zoom" independent of the size of the display or the image—you never need to worry about things getting blurry as you make them larger!

In order to include a Material icon, you will need to generate the XML code for that particular image. Luckily, Android Studio includes XML definitions for all the Material icons (though you can also define your own vector drawables).

- To create a vector drawable, select `File > New > Vector Asset` from the Android Studio menu. You can then click on the icon to browse for which Material icon you want to create.

- By default the icon will be colored *black*. If you wish to change the color of the icon when included in your layout, specify the color through the `android:tint` attribute of the View in your XML (e.g., on the `ImageButton`).

It is also possible to define arbitrary vector shapes in API 21+. For example, the starter code includes a resource for a `<shape>` element that is shaped like an oval.

[1] https://material.io/color/#!/

5.3 Design Support Libraries

In addition to the styling and resource properties available as part of the Android framework, there are also additional components available through extra **support libraries**.

Similar to Volley, these libraries are not built into Android and so need to be explicitly listed as dependencies in your app's `build.gradle` file. For example:

```
compile 'com.android.support:design:26.1.0'
```

will include the latest version (as of this writing) of the **design support library**, which includes elements such as the Floating Action Button and Coordinator Layouts (described below).

Note that if you have issues installing the dependency, the setup for including support libraries was changed in July 2017 to support downloading dependency libraries through maven rather than directly from Android Studio. To support this, you will need to change the *project's* `build.gradle` repository declaration to look like:

```
allprojects {
    repositories {
        jcenter()
        maven {
            url "https://maven.google.com"
        }
    }
}
```

Widgets

The support libraries support a number of useful widgets (specialized Views) for creating Material-styled apps.

RecyclerView

The `RecyclerView` is a more advanced version of a `ListView`, providing a more robust system for support interactive Views within the list as well as providing animations for things like adding and removing list items.

This class is part of the `v7` support library (and not the Material Design library specifically), and so you will need to include it specifically in your *app's* `build.gradle` file (the same way you included Volley):

```
compile 'com.android.support:cardview-v7:26.1.0'
```

Implementing a `RecyclerView` is very similar to implementing a `ListView` (with the addition of the ViewHolder pattern), though you also need to declare a `LayoutManager` to specific if your RecyclerView should use a List or a Grid.

The best way to understand the RecyclerView is to look at the example and modify that to fit your particular item view. For example, you can change a ListView into a RecyclerView by adapting the sample code provided by Google's official documentation[2]:

- First, you will need to replace the XML View declaration with a `<android.support.v7.widget.RecyclerView>` element. In effect, we're just modifying the **controller** for the list.
- There are a few extra steps when setting up the `RecyclerView` in the Java code. In particular, you will also need to associate a `LayoutManager` object to with the View—for example, a `LinearLayoutManager` to show items in a line (a la a ListView), or a `GridLayoutManager` to show items in a grid (a la a GridView).
- All `RecyclerViews` require *custom adapters*: we cannot just use a built-in adapter such as `ArrayAdapter`. To do this, create a class that `extends RecyclerView.Adapter<VH>`.
  - The generic in this class is a class representing a View Holder. This is a pattern by which each individual Views that will be inflated and references are stored as individual *objects* with the particular Views to be modified (e.g., `TextView`) saved as instance variables. This avoids the need for the adapter to repeatedly use `findViewById()`, which is an expensive operation (since it involves crawling the View hierarchy tree!)

    The `ViewHolder` will be *another* class—usually an inner class of the adapter. It acts as a simple "container" object (similar to a `struct` in C). You can assign these instance variables the results of the `findViewById()` calls in the ViewHolder's constructor.
  - The `RecyclerView` (not the `ViewHolder`) requires you to override the `onCreateViewHolder()` method, which will `inflate()` the View and instantiate a `ViewHolder` for each item when that item needs to be displayed for the first time.
  - In the overridden `onBindViewHolder()` you can do the actual work of assigning model content to the particular View (e.g., called `setText()`). You don't need to use `findViewById()` here, because you've already saved a reference to that View in the `ViewHolder`!
  - The `getItemCount()` is used internally for the RecyclerView to determine when you've gotten to the "end" of the list.

[2] https://developer.android.com/training/material/lists-cards.html#RecyclerView

- Finally, you can assign the custom adapter to the `RecyclerView` in order to associate the model and the View.

And that's about it! While it is more code and more complex than a basic ListView, as soon as you've added in the ViewHolder pattern or done any other kinds of customizations, you're basically at the same level. Additionally

Note that *unlike* with a ListView, we usually modify the items shown in a `RecyclerView` by modifying the data model (e.g., the array or ArrayList) directly. But we will need to notify the `RecyclerView`'s adapter of these changes by calling one of the various `.notify()` methods on the adapter (e.g., `adapter.notifyItemInserted(position)`). This will cause the adapter to "refresh" the display, and it will actually animate the changes to the list by default!

Using a RecyclerView instead of a ListView also enables common user actions such as "drag-to-order" or "swipe-to-dismiss". See this guide for a walkthrough on adding these capabilities.

Cards

The `v7` support library also provides a View for easily styling content as **Cards**. A `CardView` is basically a `FrameLayout` (a ViewGroup that contains one child View), but include borders and shadows that make the group look like a card.

- You will need to load the `CardView` class as a gradle dependency using `compile 'com.android.support:cardview-v7:26.1.0'`.

To utilize a `CardView`, simply include it in your layout as you would any other ViewGroup:

```
<android.support.v7.widget.CardView
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="@dimen/card_width"
    android:layout_height="@dimen/card_height"
    android:layout_gravity="center"
    card_view:cardCornerRadius="4dp">

    <!-- A single TextView in the card -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/card_text" />

</android.support.v7.widget.CardView>
```

- Notice the `card_view:cardCornerRadius` attribute; this is an example of how specific Views may have their own custom properties (sometimes

available via a different schema).

Since Cards are `FrameLayouts`, they should only contain a single child. If you want to include multiple elements inside a Card (e.g., an image, some text, and a button), you will need to nest another ViewGroup (such as a `LinearLayout`) inside the Card.

For design suggestions on using Cards, including spacing information, see the Material Design guidelines.

If you want to include a circular image in your card (or anywhere else in your app), the easiest solution is to include an external library that provides such a View, the most popular of which is `de.hdodenhof.circleimageview`.

Floating Action Buttons (FAB)

While RecyclerViews and Cards are found in the `v7` support library, the most noticeable and interesting components come from the Design Support Library, which specifically includes components for supporting Material Design.

- This library should be includes in gradle as `com.android.support:design:26.1.0`, as in the above example.

The most common element from this library is the **Floating Action Button (FAB)**. This is a circular button that "floats" above the content of the screen (at a higher elevation), and represents the *primary action* of the UI.

- A screen should only ever have one FAB, and only if there is a single main action that should be performed. See the design guidelines for more examples on how to use (and not use) FABS.

Like a Card, you can include a FAB in your application by specifying it as an element in your XML:

```
<android.support.design.widget.FloatingActionButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end|bottom"
        android:layout_margin="@dimen/fab_margin"
        android:src="@drawable/ic_my_icon" />
```

- Because FABs are a subclass of `ImageButton`, we can just replace an existing button with a FAB without breaking anything.

Fabs support a number of additional effects. For example, if you make the FAB clickable (via `android:clickable`), you can specify an `app:rippleColor` to give it a rippling effect when pressed. Further details will be presented below.

Snackbars

The Design Support Library also includes an alternative to Toast messages called **Snackbars**. This is a user-facing pop-up message that appears at the bottom of the screen (similar to a Toast).

Snackbars are shown using a similar structure to Toasts:

```
Snackbar snack = Snackbar.make(view, "Let's go out to the lobby!", Snackbar.LENGTH_L
```

- Instead of calling the `Toast.makeText()` factory method, we call the `Snackbar.make()` factory method. The first parameter in this case needs to be a View that the Snackbar will be "attached" to (so shown with)—however, it doesn't really matter which View is given, since the method will search up the view heirarchy until it gets to the root content view or a `CoordinatorLayout`.
- You'll notice that the Snackbar overlays the content (including the FAB). This will be addressed below by introducing a `CoordinatorLayout`.

Additionally, it is possible to give Snackbar's their own *action* that the user can activate by clicking on the Snackbar. This allows the bar to, for example, show a delete confirmation but providing an "undo" action. The action is specified by calling the `.setAction()` method on the Snackbar, and passing in a title for the action as well as an `OnClickListener`:

```
mySnackbar.setAction("Click", new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        //...
    }
});
```

- For practice, make the Snackbar `.hide()` the FAB, but provide an "undo" action that will `.show()` it!

Again, see the design guidelines for more examples on how to use (and not use) Snackbars.

Coordinator Layout

In order to fix the Snackbar and Fab overlap, we'll need to utilize one of the most powerful but complex classes in the Material support library: the **CoordinatorLayout**. This layout is described as "a super-powered Framelayout", and provides support for a number of interactive and animated behaviors involves other classes. A lot of the "whizbang" effects in Material are built on top of the CoordinatorLayout (or other classes that rely on it).

To start our exploration of CoordinatorLayout, let's begin by fixing the Snackbar overlap. To do this, we'll take the existing layout for the activity and "wrap" it in a `<android.support.design.widget.CoordinatorLayout>` element:

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Previous layout elements -->
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <!-- etc -->

    </RelativeLayout>
</android.support.design.widget.CoordinatorLayout>
```

We will also need to move the FAB definition so that it is a *direct child* of the CoordinatorLayout. Once we have done so, we should be able to click the button and watch it move up to make room for the Snackbar!

- How this works is that the CoordinatorLayout allows you to give **Behaviors** to its child views; these Behaviors will then be executed when the state of the CoordinatorLayout (or its children) changes. For example, the built-in `FloatingActionButton.Behavior` defines how the button should move in response to it's parent changing size, but Behaviors can also be defined in response to use interactions such as swipes or other gestures.

Scrolling Layouts

Indeed, the built-in behaviors can be quite complex (and wordy to implement), and the best way to understand them is to see them in action. To see an example of this, create a **new** Activity for your application (e.g., `File > New > Activity`). But instead of creating an *Empty* Activity as you've done before, you should instead create a new **ScrollingActivity**.

- Modify the FAB action so that when you click on it, you send an `Intent` for to open up this new Activity:

  ```
  startActivity(new Intent(MainActivity.this, ScrollingActivity.class));
  ```

- And once you've opened up the Activity… try scrolling! You should see the ActionBar collapse while the FAB moves up and disappears.

This is an example of a collection of behaviors built into `CoordinatorLayout` and other classes in the Design Support library. To get a sense for how they work, open up the newly created `activity_scrolling.xml` layout resource and see how this layout was constructed!

- At the root of the layout we find the `CoordinatorLayout`, ready to "coordinate" all of its children and allow them to interact.
- The first child is an `AppBarLayout`. This layout specifically supports responding to scroll events produced from within the CoordinatorLayout (e.g., when the user scrolls through the text content). You can control the visibility of this element based on scrolling using the `app:layout_scrollFlags` attribute.

  The AppBarLayout works together with its child `CollapsingToolbarLayout`, which does just what it says on the tin. It shows a larger title, but then shrinks down in response to scrolling. Here the `scrollFlags` are declared: `scroll|exitUntilCollapse` indicates two flags (combined with a bitwise OR `|`): that the content should scroll, and that it should shrink to its minimum height until it collapses.

  The Toolbar itself is finally defined as a child of `CollapsingToolbarLayout`, though other children could be added here as well. For example, an `ImageView` could be included as a child of the CollapsingToolbarLayout in order to create a collapsing image!

  (Check the documentation for details about all of the specific attributes).
- After the collapsing AppBar, the `CoordinatorLayout` includes a `NestedScrollView` (declared in a separate file for organization). This is a scrollable view (similar to what is used in a ListView), but both include and be included within scrollable layouts.
  - Notice that this element includes an `app:layout_behavior` attribute, which refers to a particular class: `AppBarLayout$ScrollingViewBehavior` (the `$` is used to refer to a compiled nested class). This Behavior will "automatically scroll any AppBarLayout siblings", allowing the scrolling of the page content to *also* cuase the AppBarLayout to scroll!
- Finally, we have the FAB for this screen. The biggest piece to note here is how the FAB includes the `app:layout_anchor` attribute assigned a reference to the AppBarLayout. This indicates that the FAB should follow (scroll with) the AppBarLayout; the `app:anchorGravity` property indicates where it should be relative to its anchor. Moreover, the FAB's default behavior will cause it to disappear when then is no room… and since the AppBarLayout exits on collapse, the FAB disappears as well!

In sum: the `NestedScrollingView` has a Behavior that will cause the AppBarLayout to scroll with it. The AppBarLayout has Behaviors that allow it

to collapse, and the FAB is connected to that layout to moves up with it and eventually disappear.

Custom Behaviors

We can also create our own custom behaviors if we want to change the way elements interact with the CoordinatorLayout. For example, we can create a Behavior so that the FAB on the `MainActivity` shrinks and disappears when the Snackbar is shown, rather than moving up out of the way!

First, we will create a new Java class to represent our `ShrinkBehavior`. This class will need to `extend CoordinatorLayout.Behavior<FloatingActionButton>` (because it is a `CoordinatorLayout.Behavior` and it will be applied to a `FloatingActionButton`).

- We will need also need to override the constructor so that we can declare/instantiate this class from the XML:

```
public ShrinkBehavior(Context context, AttributeSet attrs) {
    super(context, attrs);
}
```

The next step is to make sure the Behavior is able to react to changes in the Snackbar. To do this we have to make the Behavior report that is has the Snackbar as a *dependency*. This way when the CoordinatorLayout is propagating events and changes to all of its children, it will know that it should also inform the FAB about changes to the Snackbar. We do this by overriding the `layoutDependsOn()` method:

```
public boolean layoutDependsOn(CoordinatorLayout parent,
                               FloatingActionButton child, View dependency) {
    //add SnackbarLayout to the dependency list (if any)
    return dependency instanceof Snackbar.SnackbarLayout ||
        super.layoutDependsOn(parent, child, dependency);
}
```

- (Technically the `super` class doesn't have any other dependencies, but it's still good practice to call up the tree).

Finally, we can specify what should happen when one of the dependency's Views change by overriding the `onDependentViewChange()` callback:

```
public boolean onDependentViewChanged(CoordinatorLayout parent, FloatingActionButton child, Vi
        if(dependency instanceof Snackbar.SnackbarLayout){
            //calculate how much Snackbar we see
            float snackbarOffset = 0;
            if(parent.doViewsOverlap(child, dependency)){
                snackbarOffset = Math.min(snackbarOffset, dependency.getTranslationY() - depen
```

```
            }
            float scaleFactor = 1 - (-snackbarOffset/dependency.getHeight());

            child.setScaleX(scaleFactor);
            child.setScaleY(scaleFactor);
            return true;
        }else {
            return super.onDependentViewChanged(parent, child, dependency);
        }
    }
```

- This method will be passed a reference to the `CoordinatorLayout` that is managing the changes, the `FloatingActionButton` who is receiving the change, and *which dependency* had it's View changed. We check that the dependency is actually a `Snackbar` (since we might have multiple dependencies and want to respond differently to each one), and then call some getters on that dependency to figure out how tall it is (and thus how much we should shrink by). Finally, we use setters to change the scaling of the `child` (the FAB), thereby having it scale!
- And because this scale is dependent on the Snackbar's height, the FAB will also "grow back" when the Snackbar goes away!

This is about the simplest form of Behavior we can have: more complex behaviors can be based on scroll or fling events, utilizing different state attributes for different dependencies!

Custom behaviors are very tricky to write and design; the overridden functions are not very well documented, and by definition these behaviors involve coordinating lots of different classes! Most custom behaviors are designed by reading the Android source code (e.g., for `FloatingActionButton.Behavior`) and modifying that as an example. My suggestion is to search online for behaviors similar to the one you're trying to achieve, and work from there.

5.4 Animations

One of the key principles of Material Design was the use of *motion*: many of the previous examples have involved adding animated changes to elements (e.g., moving or scrolling). The Material theme available in API 21+ provides a number of different techniques for including **animations** in your application—in particular, using animation to give user feedback and provide connections between elements when the display changes. As a final example, this section will cover how to add simple Activity Transitions so that Views "morph" from one Activity to the next.

In order to utilize Activity Transitions, you will need to enable them in your

application's *theme* by adding an addition `<item>` in the theme declaration (in `res/values/styles.xml`):

```
<!-- in style: enable window content transitions -->
<item name="android:windowActivityTransitions">true</item>
```

There are three different kinds of Activity Transitions we can specify:

1. *enter* transitions, or how Views in an Activity enter the screen
2. *exit* transitions, or how Views in an Activity leave the screen
3. *shared element* transitions, or how Views that are shared between Activities change

We'll talk about the later, though the previous two follow a similar process.

In order to animate a **shared element** between two Activities, we need to give them matching identifiers so that the transition framework knows to animate them. We do this by giving each of the shared elements an `android:transitionName` attribute, making sure that they have the *same* value.

- For example, we can give the FAB in each activity an `android:transitionName="fab"`.
- Because the FAB is *anchored*, we actually need to do some extra work (because FAB will morph to the "unanchored" point and then get moved once the anchorage is calculated). The easiest workaround for this is to wrap the anchored FAB inside an anchored `FrameLayout`—the FAB just then becomes a normal element with the FrameLayout handling the scrolling behavior.

```
<FrameLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_anchor="@id/app_bar"
    app:layout_anchorGravity="bottom|end"
    android:elevation="12dp"
    >

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:transitionName="same_fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@android:drawable/ic_dialog_email" />
</FrameLayout>
```

Finally, we need to make sure that the Intent used to start the Activity also starts up the transition animation. We do this be including an additional argument to the `startActivity()` method: an options `Bundle` containing details

about the animation:

```
//transition this single item
ActivityOptions options = ActivityOptions.makeSceneTransitionAnimation(MainActivity.

// start the new activity
startActivity(new Intent(MainActivity.this, ScrollingActivity.class), options.toBund
```

This should cause the FAB to "morph" between Activties (and even morph back when you hit the "back" button)!

For more about what makes effective animation, see the Material deisgn guidelines.

I find Activity Transitions to be slick, but finicky: getting them to work properly takes a lot of effort and practice. Most professional apps that utilize Material Design use extensive custom animations for generating these transitions. For examples of more complex Material Design animations and patterns, check out sample applications such as cheesesquare or plaid.

Property Animation

Bonus content! More about animation!

Android actually supports a number of different animation systems that can be used within and *across* Views. For example:

- Android includes a robust framework for Scene Transitions even outside of Material; see Adding Animations for more details.
- Android also supports OpenGL Animations for doing 3D animated systems. This requires knowing the OpenGL API.

In this section, we will discuss how to use **Property Animation**. This is a general animation framework in which you specify a start state for an Object *property* (attribute), an end state for that property, and a duration for the animation; the Android systems then changes the property from the start state to the end over that length of time—thereby producing animation!

- The change in property state over time (that is, at any given "frame") is calculated using **interpolation**. This is basically a "weighted average" between the the start and end states, where the weight is determined by how close you are to the "start" or "end". While we often use *linear interpolation* (so that being 70% across means the end gets 70% of the weight), it is also possible to use *non-linear interpolation*) (e.g., you need to get 70% across in for the end to have 50% of the weight).

The main engine for doing this kind of interpolated animation in Android is

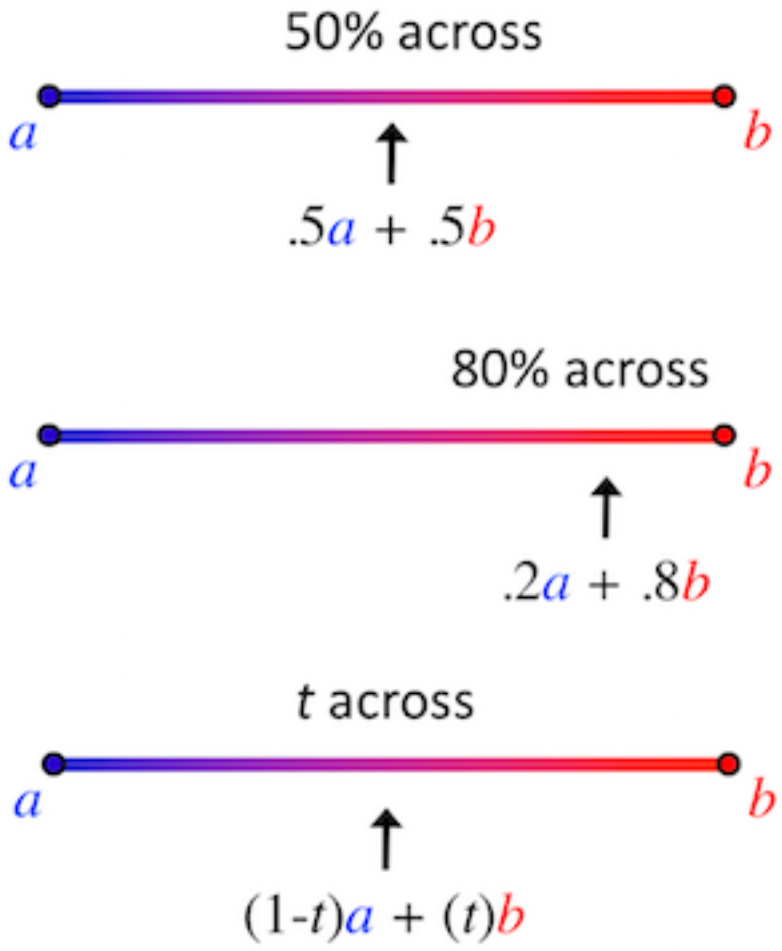

Figure 5.1: Linearly interpolating colors

the `ValueAnimator`[3] class. This class lets you specify the start state, end state, and animation duration. It will then be able to run through and calculate all of the "intermediate" values throughout the interpolated animation. The `ValueAnimator` class provides a number of static methods (e.g., `.ofInt()`, `.ofFloat()`, `.ofArgb()`) which creates "animators" for interpolating different *value types*. For example:

```
ValueAnimator animation = ValueAnimator.ofFloat(0f, 1f); //interpolate between 0 and
animation.setDuration(1000); //over 1000ms (1 second)
animation.start(); //run the animation
```

Of course, just performing this interpolation over time doesn't produce any visible result—it's changing the numbers, but those numbers don't correspond to anything.

We can access the interpolated values by registering a listener and overriding the callback we're interested in observing (e.g., `onAnimationUpdate()` from `ValueAnimator.AnimatorUpdateListener`). But more commonly, we want to have our interpolated animation change the *property* of some object—for example, the color of a View, the position of an Button, or the instance variables of an object such as a `Ball`.

We can do this easily using the `ObjectAnimator`[4] subclass. This subclass runs an animation just like the `ValueAnimator`, but has the built-in functionality to change a property of an object on each interpolated step. It does this by calling a **setter** for that property—thus the object needs to have a setter method for the property we want to animate:

```
//change the "alpha" property of foo: will call `foo.setAlpha(float)`
ObjectAnimator anim = ObjectAnimator.ofFloat(foo, "alpha", 0f, 1f);
anim.setDuration(1000);
anim.start();
```

- This example will mutate the object by calling the `setAlpha()` method (the name of the method is generated from the property name following normal CamelCasing style).
- If the object lacks such a setter, such as because we are using a class provided by someone else, we can either make a "wrapper" which will call the appropriate mutating function, or just utilize a `ValueAnimator` with an appropriate listener.

For example, we can use this approach to change the squares's size or position using an interpolated animation (make it "pulse").

[3] http://developer.android.com/guide/topics/graphics/prop-animation.html#value-animator

[4] http://developer.android.com/guide/topics/graphics/prop-animation.html#object-animator

- Note that we're just changing the object property; the only reason we would see the changed display is because Android is responding to changes in the View.

The `ObjectAnimator` interpolator methods support a number of variations as well:

- As long as the object has an appropriate **getter**, it is possible to only pass the Animator an ending value (indicating that the animation should interpolate "from current state to specified end state")
- We can use `.setRepeatCount(ObjectAnimator.INFINITE)` and `.setRepeatMode(ObjectAnimator.REVERSE)` to cause it to repeat back and forth.

If we want to include multiple animations in sequence, we can use an `AnimatorSet`, which gives us methods used to specify the ordering:

```
//example from docs
ObjectAnimator animX = ObjectAnimator.ofFloat(obj, "x", 50f);
ObjectAnimator animY = ObjectAnimator.ofFloat(obj, "y", 100f);
AnimatorSet animSetXY = new AnimatorSet();
animSetXY.playTogether(animX, animY);
animSetXY.start();
```

AnimatorSet animations can get complicated, and we may want to reuse them. Thus best practice is to instead define them in XML as resources. Animation resources are put inside the `/res/animator` directory (**not** the `/res/anim/` folder, which is for View Animations).

```
<set android:ordering="together"> <!-- together is default -->
  <objectAnimator
          android:propertyName="x"
          android:duration="500"
          android:valueTo="400"
          android:valueType="intType"/>
  <!-- ... -->
</set>
```

- See Property Animation Resources[5] for the full XML schema.
- Note that by defining animations as resources, it also mens that we can easily have different device configurations use different animations (e.g., perhaps objects move faster on larger displays).

In order to utilize the XML, we will need to **inflate** the Animator resource, just as we have done with Layouts:

[5] http://developer.android.com/guide/topics/resources/animation-resource.html#Property

```
ObjectAnimator anim = (ObjectAnimator)AnimatorInflater.loadAnimator(context, R.anim.
anim.setTarget(myObject);
anim.start();
```

Note that we can also use this same framework to animate changes to `Views`: Buttons, Images, etc. Views are objects and have properties (along with appropriate *getter* and *setter* methods), so we can use just an `ObjectAnimator`! See Animating Views for a list of properties we can change (a list that includes `x, y, rotation, alpha, scaleX, scaleY`, and others)

To make this process even simpler, Android also provides a `ViewPropertyAnimator` class. This Animator is able to easily animate multiple properties together (at the same time), and does so in a much more efficient way. We can access this Animator via the `View#animate()` method. We then call relevant shortcut methods on this Animator to "add in" addition property animations to the animation set:

```
//animate x (to 100) and y (to 300) together!
myView.animate().x(100).y(300);
```

This allows you to easily specify moderately complex property animations for `View` objects.

- But really, if you want to animate layout changes on a modern device, you should use transitions, such as the ones we used with Material design.

There are many more ways to customize exactly what you want your animation to be. You can also look at official demos for more examples of different styles of animation.

Resources

- Material Design Guidelines check the whole document (via the hamburger menu on the left).
- Material Design for Developers (Google) official documentation for implementing material design
- Material Design Primer (CodePath) excellent compiled documentation and examples for implementing Material patterns (CodePath in general is an excellent resource).
- Android Design Support Library (Google Blog) an introduction to the support library features
- Mastering the Coordinator Layout (Blog) a great set of examples of how to use CoordinatorLayout.

- Using CoordinatorLayout in Android Apps (Blog) another good explanation of CoordinatorLayout
- https://lab.getbase.com/introduction-to-coordinator-layout-on-android/

Chapter 6

Fragments

This lecture discusses Android **Fragments**. A Fragment is "a behavior or a *portion* of user interface in Activity." You can think of them as "mini-activities" or "sub-activities". Fragments are designed to be **reusable** and **composable**, so you can mix and match them within a single screen of a user interface. While XML resource provide reusable and composable *views*, Fragments provide reusable and composable *controllers*. Fragments allow us to make re-usable pieces of Activities that can have their own layouts, data models, event callbacks, etc.

This lecture references code found at https://github.com/info448/lecture06-fragments.

Fragments were introduced in API 11 (Honeycomb), which provided the first "tablet" version of Android. Fragments were designed to provide a UI component that would allow for side-by-side activity displays appropriate to larger screens:

Instead of needing to navigate between two related views (particularly for this "master and detail" setup), the user can see both views within the same Activity… but those "views" could also be easily split between two Activities for smaller screens, because their required *controller logic* has been isolated into a Fragment.

Fragments are intended to be **modular**, **reusable** components. They should ***not*** depend on the Activity they are inside, so that you can be flexible about when and where they are displayed!

Although Fragments are like "mini-Activities", they are *always* embedded inside an `Activity`; they cannot exist independently. While it's possible to have Fragments that are not visible or that don't have a UI, they still are part of an Activity. Because of this, a Fragment's lifecycle is directly tied to its containing Activity's lifecycle. (e.g., if the Activity is paused, the Fragment is too. If the

[1] https://developer.android.com/images/fundamentals/fragments.png

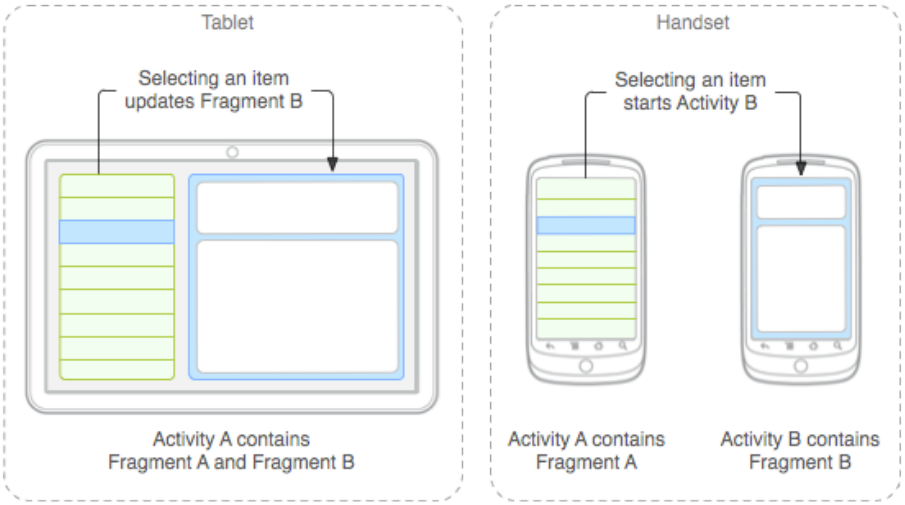

Figure 6.1: Fragment example, from Google[1].

Activity is destroyed, the Fragment is too). However, Fragments also have their own lifecycle with corresponding lifecycle callbacks functions.

The Fragment lifecycle is very similar to the Activity lifecycle, with a couple of additional steps:

- **`onAttach()`**: called when the Fragment is first associated with ("added to") an Activity, and thus gains a **Context**. This callback is generally used for initializing communication between the Fragment and its Activity.

  This callback is mirrored by `onDetach()`, for when the Fragment is removed from an Activity.

- **`onCreateView()`**: called when the View (the user interface) is about to be drawn. This callback is used to establish any details dependent on the View (including adding event listeners, etc).

  Note that code initializing data models, or anything that needs to be *persisted* across configuration changes, should instead be done in the `onCreate()` callback. `onCreate()` is not called if the fragment is *retained* (see below).

  This callback is mirrored by `onDestroyView()`, for when the Fragment's UI View hierarchy is being removed from the screen.

- **`onActivityCreated()`**: called when the *containing Activity's*

[2]https://developer.android.com/images/fragment_lifecycle.png

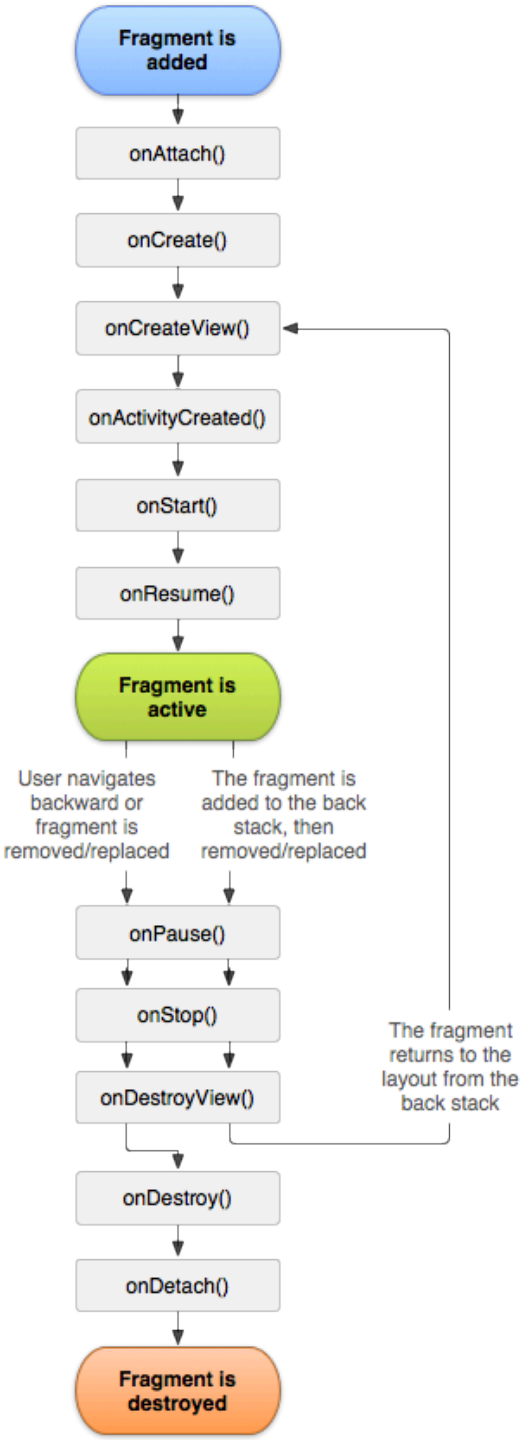

Figure 6.2: Fragment lifecycle state diagram,from Google[2].

`onCreate()` method has returned, and thus indicates that the Activity is fully created. This is useful for *retained* Fragments.

This callback has no mirror!

6.1 Creating a Fragment

In order to illustrate how to make a Fragment, we will **refactor** the `MainActivity` to use Fragments for displaying the list of movies. This will help to illustrate the relationship between Activities and Fragments.

To create a Fragment, you subclass the `Fragment` class. Let's make one called `MovieListFragment` (in a `MovieListFragment.java` file). You can use Android Studio to do this work: via the `File > New > Fragment > Fragment (blank)` menu option. (**DO NOT** select any of the other options for in the wizard for now; they provide template code that can distract from the core principles).

There are two versions of the `Fragment` class: one in the framework's `android.app` package and one in the `android.support.v4` package. The later package refers to the Support Library. As discussed in the previous chapter, these are libraries of classes designed to make Android applications *backwards compatible*: for example, `Fragment` and its related classes came out in API 11 so aren't in the `android.app` package for earlier devices. By including the support library, we can include those classes on older devices as well!

- Support libraries *also* include additional convenience and helper classes that are not part of the core Android package. These include interface elements (e.g., `ConstraintLayout`, `RecyclerView`, or `ViewPager`) and accessibility classes. See the features list for details. Thus it is often useful to include and utilize support library versions of classes even when targeting later devies so that you don't need to "roll your own" versions of these convenience classes.

- The main disadvantage to using support libraries is that they need to be included in your application, so will make the final `.apk` file larger (and may potentially require workarounds for method count limitations). You will also run into problems if you try and mix and match versions of the classes (e.g., from different versions of the support library). But as always, you should *avoid premature optimization*. Thus in this course you should **default** to using the support library version of a class when given a choice!

After we've created the `MovieListFragment.java` file, we'll want to specify a layout for that Fragment (so it can be shown on the screen). As part of using the New Fragment Wizard we were provided with a `fragment_movie_list` layout that we can use.

- Since we want the Movie list to live in that Fragment, we can move (copy) the View definitions from `activity_main` into `fragment_movie_list`.
- We will then adjust `activity_main` so that it instead contains an empty `FrameLayout`. This will act as a simple **"container"** for our Fragment (similar to an empty `<div>` in HTML). *Be sure to give it an* `id` *so we can refer to it later!*.

It is possible to include the Fragment directly through the XML, using the XML to instantiate the Fragment (the same way that we have the XML instantiate Buttons). We do this by specifying a `<fragment>` element, with an `android:name` attribute assigned a reference to the `Fragment` class:

```
<fragment
   android:id="@+id/fragment"
     android:name="edu.uw.fragmentdemo.MovieListFragment"
     android:layout_width="match_parent"
     android:layout_height="match_parent"/>
```

Defining the Fragment in the XML works (and will be fine to start with), but in practice it is *much* more common and worthwhile to instantiate the Fragments **dynamically** at runtime in the Java code—thereby allowing the Fragments to be dynamically determined and changed. We will start with the XML version to built the Fragment, and then shift to the Java version.

We can next begin filling in the Java logic for the Fragment. Android Studio provides a little starter code: a constructor and the `onCreateView()` callback—the later is more relevant since we will use that to set up the layout (similar to in the `onCreate()` function of `MainActivity`). But the `MainActivity#onCreate()` method specifies a layout by calling `setContentView()` and passing a resource id. With Fragments, we can't just "set" the View because the Fragment *belongs to* an Activity, and so will exist *inside* its View hierarchy! Instead, we need to figure out which ViewGroup the Fragment is inside of, and then **inflate** the Fragment's View inside that ViewGroup.

This "inflated" View is referred to as the ***root view***: it is the "root" of the Fragment's View tree (the View that all the Views inside the Fragment's layout will be attached to). We access the root view by *inflating* the fragment's layout, and saving a reference to the inflated View:

```
View rootView = inflater.inflate(R.layout.fragment_layout, container, false);
```

- Note that the `inflater` object we are calling `inflate()` on was passed as a parameter to the `onCreateView()` callback. The parameters to the `inflate()` method are: the layout to inflate, the `ViewGroup` (`container`) into which the layout should be inflated (also passed as a parameter to the callback), and whether or not to "attach" the inflated layout to the container (`false` in this case because the Fragment system already handles the attachment, so the inflate method doesn't need to). The `onCreateView()` callback must `return` the inflated *root view*, so that the

system can perform this attachment.

With the Fragment's layout defined, we can start moving functionality from the Activity into the Fragment.

- The `ListView` and `adapter` setup will need to be moved over. The UI setup (including initializing the Adapter) will be moved from the Activity's `onCreate()` to the Fragment's `onCreateView()`. However, you will need to make a few changes during this refactoring:
  - The `findViewById()` method is a method of the `Activity` class, and thus can't be called on an implicit `this` inside the Fragment. Instead, the method can be called on the **root view**, searching just that View and its children.
  - The Adapter's constructor requires a `Context` as its first parameter; while an `Activity` is a `Context`, a `Fragment` is *not*—Fragments operate in the Context of their containing Activity! Fragments can refer to the Activity that they are inside (and the `Context` it represents) by using the `getActivity()` method. Note that this method is used *primarily* for getting a reference to a `Context`, not for arbitrary communication with the Activity (see below for details)
- The `downloadMovieData()` helper method can be moved over, so that it belongs to the Fragment instead of the Activity. It will be the *Fragment's* responsibility to handle data downloading.

Activity-to-Fragment Communication

This example has intentionally left the *input controls* (the search field and button) in the Activity, rather than making them part of the Fragment. Apart from being a useful demonstration, this allows the Fragment to have a single purpose (showing the list of movies) and would let us change the search UI independent of the displayed results. But since the the button is in the Activity while the downloading functionality is in the Fragment, we need a way for the Activity to "talk" to the Fragment. We thus need a *reference* to the contained Fragment—access to the XML similar to that provided by `findViewById`.

We can get a reference to a contained Fragment from an Activity by using a `FragmentManager`. This is an object responsible for (ahem) managing Fragment. It allows us to "look up" Fragments, as well as to manipulate which Fragments are shown. We access this FragmentManager by calling the `getSupportFragmentManager()` method on the Activity, and then can use `findFragmentById()` to look up an XML-defined Fragment by its `id`:

```
//MovieListFragment example
MovieListFragment fragment = (MovieListFragment)getSupportFragmentManager().findFrag
```

- Note that we're using a method to explicit access the **support** `FragmentManager`. The Activity class (API level 15+) is able to work with both the platform and support `FragmentManager` classes. But because these classes don't have a shared `interface`, the Activity needs to provide different Java methods which can return the correct type.

Once you have a reference to the Fragment, this acts just like any other object—you can call any `public` methods it has! For example, if you give the Fragment a public method (e.g., `searchMovies()`), then this method can be called from the Activity:

```
//called from Activity on the referenced fragment
fragment.searchMovies(searchTerm)
```

(The parameter to this public method allows the Activity to provide information to the Fragment!)

At this point, the program should be able to be executed… and continue to function in exactly the same way! The program has just been refactored, so that all the movie downloading and listing work is **encapsulated** inside a Fragment that can be used in different Activities.

- In effect, we've created our own "widget" that can be included in any other screen, such as if we repeatedly wanted the list of movies to be available alongside some other user interface components.

6.2 Dynamic Fragments

The real benefit from encapsulating behavior in a Fragment is to be able to support multiple Fragments within a single Activity. For example, in the the archetypal "master/detail" navigation flow, one screen (Fragment) holds the "master list" and another screen (Fragment) holds details about a particular item. This is a very common navigation pattern for Android apps, and can be seen in most email or news apps.

- On large screens, Fragments allow these two Views to be placed side by side!

In this section, we will continue to refine the Movie app so that when the user clicks on a Movie in the list, the app shows a screen (Fragment) with details about the selected movie.

Instantiating Fragments

To do this, we will need to instantiate the desired Fragment **dynamically** (in Java code), rather than statically in the XML using the `<fragment>` element.

This is because we need to be able to dynamically change *which* Fragment is currently being shown, which is not possibly for Fragments that are "hard-coded" in the XML.

Unlike Activities, Fragments (such as `MovieListFragment`) **do** have constructor methods that can be called. In fact, Android *requires* that every Fragment include a default (no-argument) constructor that is called when Fragments are created by the system! While we are have access to the constructor, it is considered best practice to **not** call this constructor directly when you want to instantiate a Fragment, and to in fact leave the method empty. This is because we do not have full control over when the constructor is executed: the Android system may call the no-argument constructor whenever it needs to recreate the Activity (or just the Fragment), which can happen at arbitrary times. Since only this default constructor is called, we can't add an additional constructor with any arguments we may want the Fragment to have (e.g., the `searchTerm`)… and thus it's best to not use it at all.

Instead, we specify a **simple factory** method (by convention called `newInstance()`) which is able to "create" an instance of the Fragment for us. This factory method can take as many arguments as we want, and then does the work of passing these arguments into the Fragment instantiated with the default constructor:

```
public static MyFragment newInstance(String argument) {
    MyFragment fragment = new MyFragment(); //instantiate the Fragment
    Bundle args = new Bundle(); //an (empty) Bundle for the arguments
    args.putString(ARG_PARAM_KEY, argument); //add the argument to the Bundle
    fragment.setArguments(args); //add the Bundle to the Fragment
    return fragment; //return the Fragment
}
```

In order to pass the arguments into the new Fragment, we wrap them up in a `Bundle` (an object containing basic *key-value pairs*). Values can be added to a `Bundle` using an appropriate `putType()` method; note that these do need to be primitive types (`int`, `String`, etc.). The `Bundle` of arguments can then be assignment to the Fragment by calling the `setArguments()` method.

- We will be able to access this `Bundle` from inside the Fragment (e.g., in the `onCreateView()` callback) by using the `getArguments()` method (and `getType()` to retrieve the values from it). This allows us to dynamically adjust the content of the Fragment's Views! For example, we can run the `downloadMovieData()` function using this argument, fetching movie results as soon as the Fragment is created (e.g., on a button press)… allowing the `downloadMovieData()` function to again be made private, for example.

- Since the `Bundle` is a set of *key-value* pairs, each value needs to have a particular key. These keys are usually defined as `private` constants

(e.g., `ARG_PARAM_KEY` in the above example) to make storage and retrieval easier.

We will then be able to instantiate the Fragment (e.g., in the Activity class), passing it any arguments we wish:

```
MyFragment fragment = MyFragment.newInstance("My Argument");
```

Transactions

Once we've instantiated a Fragment in the Java, we need to attach it to the view hierarchy: since we're no longer using the XML `<fragment>` element, we need some other way to load the Fragment into the `<FrameLayout>` container.

We do this loading using a **`FragmentTransaction`**[3]. A transaction represents a *change* in the Fragment that is being displayed. You can think of this like a bank (or database) transaction: they allow you to add or remove Fragments like we would add or remove money from a bank account. We instantiate new transactions representing the change we wish to make, and then "run" that transaction in order to apply the change.

To create a transaction, we utilize the `FragmentManager` again; the `FragmentManager#beginTransaction()` method is used to instantiate a **new** `FragmentTransaction`.

Transactions represent a set of Fragment changes that are all "applied" at the same time (similar to depositing and withdrawing money from multiple accounts all at once). We specify these transactions using by calling the `add()`, `remove()`, or `.replace()` methods on the `FragmentTransaction`.

- The `add()` method lets you specify which View **container** you want to add a particular Fragment to. The `remove()` method lets you remove a Fragment you have a reference to. The `replace()` method removes any Fragments in the container and then adds the specified Fragment instead.
- Each of these methods returns the modified `FragmentTransaction`, so they can be "chained" together.

Finally, we call the `commit()` method on the transaction in order to "submit" it and have all of the changes go into effect.

We can do this work in the Activity's search click handler to add a Fragment, rather than specifying the Fragment in the XML:

```
FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
//params: container to add to, Fragment to add, (optional) tag
```

[3] https://developer.android.com/reference/android/support/v4/app/FragmentTransaction.html

```
transaction.add(R.id.container, myFragment, MOVIE_LIST_FRAGMENT_TAG);
transaction.commit();
```

- The third argument for the `add()` method is a "tag" we apply to the Fragment being added. This gives it a "name" that we can use to find a reference to this Fragment later if we want (via `FragmentManager#findFragmentByTag(tag)`). Alternatively, we can save a reference to the Fragment as an instance variable; this is faster but more memory intensive (and can cause possible leaks, since the reference keeps the Fragment from being reclaimed by the system).

Inter-Fragment Communication

We can this transaction-based structure to instantiate and load a **second Fragment** (e.g., a "detail" view for a selected Movie). We can add functionality (e.g., in the `onClick()` handler) so that when the user clicks on a movie in the list, we **`replace()`** the currently displayed Fragment with this new details Fragment.

However, remember that Fragments are supposed to be **modular**—each Fragment should be *self-contained*, and not know about any other Fragments that may exist (after all, what if we wanted the master/detail views to be side-by-side on a large screen?)

Using `getActivity()` to reference the Activity and `getSupportFragmentManager()` to access the manager is a violation of the Law of Demeter—don't do it!

Instead, we have Fragments communicate by passing messages through their contained Activity: the `MovieFragment` should tell its Activity that a particular movie has been selected, and then that Activity can determine what to do about it (e.g., creating a `DetailFragment` to display that information).

The recommended way to provide Fragment-to-Activity communication is to define an **interface**. The Fragment class should specify an `interface` (for one or more public methods) that its containing Activity *must* support—and since the Fragment can only exist within an Activity that implements that interface, it knows the Activity has the specified public methods that it can call to pass information to that Activity.

As an example of this process:

- Create a new `interface` inside the Fragment (e.g., `OnMovieSelectedListener`). This interface needs a public method (e.g., `onMovieSelected(Movie movie)`) that the Fragment can call to give instructions or messages to the Activity.
- In the Fragment's `onAttach()` callback (called when the Fragment is first associated with an Activity), we can check that the Activity actually im-

plements the interface by trying to *cast* it to that interface. We can also save a reference to this Activity for later, to save some time:

```
public void onAttach(Context context) {
    super.onAttach(context);

    try {
        callback = (OnMovieSelectedListener)context; //attempt to cast
    } catch (ClassCastException e) {
        throw new ClassCastException(context.toString() + " must implement OnMovieSelected
    }
}
```

- Then when an action occurs in the Fragment (e.g., a movie is selected), you call the interface's method on the `callback` reference.
- Finally, you will need to make sure that the Activity `implements` this callback. Remember that a class can implement multiple interfaces!

  In the Activity's implementation of the interface, you can handle the information provided. For example, use the `FragmentManager` to create a `replace()` transaction to load a new `DetailFragment` for the appropriate data.

In the end, this will allow you to have one Fragment cause the application to switch to another!

This is not the only way for Fragments to communicate. It is also possible to have a Fragment send an `Intent` to the Activity, who then responds to that as appropriate. But using the Intent system is more resource-intensive than using interfaces.

Parcelable

It's not uncommon to want to pass multiple or complex data values as arguments to a new Fragment, such as a `Movie` object. However, the `arguments` for a new Fragment must be passed in through a `Bundle`, which are only able to store primitive values (e.g., `int`, `float`, `String`)… and a special data type called **Parcelable**. An object that implements the `Parcelable` interface includes methods that allow it to be *serialized* into a value that is simple enough to place in a Bundle—in effect, it's values are converted into a formatted String (similar in concept to `JSON.stringify()` in JavaScript).

Implementing the `Parcelable` interface involves the following steps: 1. provide a `writeToParcel()` method that adds the object's fields to a given `Parcel` object 2. provide a constructor that can re-create the object from a `Parcel` by reading the values from it (*in the EXACT SAME order they were added*!) 3. provide a `describeContents()` that returns whether the parcel includes a file

descriptor (usually `0`, meaning not) 4. provide a `Parcelable.Creator` *constant* that can be used to create the Parcelable.

These steps seem complex but are fairly rote: as such, it's possible to "automate" making a class into a Parcelable. My favorite strategy is to use **http://www.parcelabler.com/**, a website that allows you to copy and paste a class and provides you a version of that class with the Parcelable interface implemented!

Bundles are only supposed to store small amounts of information (a few variables); and by extension Parcelable objects should also only include a few attributes. **Never** make an object that contains an array of a list into a Parcelable! Complex or variable-length data should instead be persisted separate from the Activity (e.g., in a database), with identifiers passed in Bundles instead.

The Back Stack

But what happens when we hit the "back" button? The Activity exits! *Why?* Because "back" normally says to "leave the Activity"—we only had one Activity, just multiple fragments.

Recall that the Android system may have lots of Activities (even across multiple apps!) with the user moving back and forth between them. As described in Lecture 3, each new Activity is associated with a "task" and placed on a **stack**[4]. When the "back" button is pressed, that Activity is popped off the stack, and the user is taken to the Activity that is now at the top.

Fragments by default are not part of this "back-stack", since they are just components of Activities. However, you *can* specify that a transaction should include the Fragment change as part of the stack navigation by calling `FragmentTransaction#addToBackStack()` as part of your transaction (e.g., right before you `commit()`):

```
getSupportFragmentManager().beginTransaction()
                           .add(detailFragment, "detail")
                           // Add this transaction to the back stack
                           .addToBackStack(null)
                           .commit();
```

Note that the "back" button will cause *the entire transaction* to "reverse". Thus if you performed a `remove()` then an `add()` (e.g., via a `replace()`), then hitting "back" will cause the the previously added Fragment to be removed *and* the previously removed Fragment to be added.

- `FragmentManager` also includes numerous methods for manually manipulating the back-stack (e.g., "popping" off transactions) if you want to include custom navigation elements such as extra "back" buttons.

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

6.3 Dialogs

We have previously provided feedback to users via simple pop-ups such as Toasts or Snackbars. However, sometimes you would like to show a more complex "pop-up" View—perhaps one that requires additional interaction.

A ***Dialog***[5] is a "pop-up" modal (a view which doesn't fill the screen) that either asks the user to make a decision or provides some additional information. At it's most basic, Dialogs are similar to the `window.alert()` function and its variants used in JavaScript.

There is a base `Dialog` class, but almost always we use a pre-defined subclass instead (similar to how we've use `AppCompatActivity`). `AlertDialog`[6] is the most common version: a simple message with buttons you can respond with (confirm, cancel, etc).

We don't actually instantiate an `AlertDialog` directly (in fact, it's constructors are `protected` so inaccessible to us). Instead we use a helper *factory* class called an `AlertDialog.Builder`. There are a number of steps to use a builder to create a Dialog:

1. Instantiate a new builder for this particular dialog. The constructor takes in a `Context` under which to create the Dialog. Note that once the builder is initialized, you can create and recreate the same dialog with a single method call—that's the benefits of using a factory.
2. Call "setter" methods on the builder in order to specify the title, message, etc. for the dialog that will appear. This can be hard-coded text or a reference to an XML String resource (as a user-facing String, the later is more appropriate for published applications). Each setter method will return a reference to the builder, making it easy to chain them.
3. Use appropriate setter methods to specify callbacks (via a `DialogInterface.OnClickListener`) for individual buttons. Note that the "positive" button normally has the text `"OK"`, but this can be customized.
4. Finally, actually instantiate the `AlertDialog` with the `builder.create()` method, using the **`show()`** method to make the dialog appear on the screen!

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Alert!")
       .setMessage("Danger Will Robinson!");
builder.setPositiveButton("I see it!", new DialogInterface.OnClickListener() {
  public void onClick(DialogInterface dialog, int id) {
    // User clicked OK button
  }
```

[5] https://developer.android.com/guide/topics/ui/dialogs.html
[6] https://developer.android.com/reference/android/support/v7/app/AlertDialog.html

```
});

AlertDialog dialog = builder.create();
dialog.show();
```

An important part of learning to develop Android applications is being able to read the API to discover effective options. For example, can you read the AlertDialog.Builder API and determine how to add a "cancel" button to the alert?

While `AlertDialog` is the most common Dialog, Android supports other subclasses as well. For example, `DatePickerDialog` and `TimePickerDialog` provide pre-defined user interfaces for picking a date or a time respectively. See the Pickers guide for details about how to utilize these.

DialogFragments

The process described above will create and show a Dialog, but that dialog has a few problems in how it interacts with the rest of the Android framework—namely with the lifecycle of the Activity in which it is embedded.

For example, if the device changes configurations (e.g., is rotated from portrait to landscape) then the Activity is destroyed and re-created (it's `onCreate()` method will be called again). But if this happens while a Dialog is being shown, then a `android.view.WindowLeaked` error will occur and the Dialog is lost!

To avoid these problems, we need to have a way of giving that Dialog its own lifecycle which can interact with the the Activity's lifecycle… sort of like making it a *modular* piece of an Activity… that's right, we need to make it a Fragment! Specifically, we will use a subclass of Fragment called `DialogFragment`, which is a Fragment that displays as a modal dialog floating above the Activity (no extra work needed).

Just like with the previous Fragment examples, we'll need to create our own subclass of `DialogFragment`. It's often easiest to make this a *nested class* if the Dialog won't be doing a lot of work (e.g., shows a simple confirmation).

Rather than specifying a Fragment layout through `onCreateView()`, we can instead override the `onCreateDialog()` callback to specify a `Dialog` object that will provide the view hierarchy for the Fragment. This Dialog can be created with the `AlertDialog.Builder` class as before!

```
public static class MyDialogFragment extends DialogFragment {

    public static HelloDialogFragment newInstance() {
        Bundle args = new Bundle();
        HelloDialogFragment fragment = new HelloDialogFragment();
```

```
        fragment.setArguments(args);
        return fragment;
    }

    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        //...
        AlertDialog dialog = builder.create();
        return dialog;
    }
}
```

Finally, we can actually show this `DialogFragment` by instantiating it (remember to use a `newInstance()` factory method!) and then calling the `show()` method on it to make it show as a Dialog. The `show()` method takes in a `FragmentManager` used to manage this transaction. By using a DialogFragment, it is possible to change the device configuration (rotate the phone) and the Dialog is retained.

Here's the other neat trick: a `DialogFragment` is just a `Fragment`. That means we can use it *anywhere* we normally used Fragments… including embedding them into layouts! For example if you made the `DetailsFragment` subclass `DialogFragment` instead of `Fragment`, it would be able to be used in the exact same as before. It's still a Fragment, just with extra features—one of which is a `show()` method that will show it as a Dialog!

- Use `setStyle(DialogFragment.STYLE_NO_TITLE, android.R.style.Theme_Holo_Light_Dialog)` to make the Fragment look a little more like a dialog.

The truth is that Dialogs are not very commonly used in Android (compare to other GU systems). Apps are more likely to just dynamically change the Fragment or Activity being shown, rather than interrupt the user flow by creating a pop-up modal. And 80% of the Dialogs that *are* used are AlertDialogs. Nevertheless, it is worth being familiar with this process and the patterns it draws upon!

Part II

Additional Topics (Labs)

Chapter 7

Styles & Themes

In this chapter you will learn to use Android **Styles & Themes** to easily modify and *abstract* the appearance of an app's user interfaces—that is, the XML resource attributes that define what your `Views` look like.

This tutorial will walk you through creating and using styles to modify views, though you should also reference the official documentation for more details and examples.

The code for this tutorial can be found at https://github.com/info448/lab-styles.

You will be working almost exclusively with the **XML resources** (e.g., `res/layout/activity_main.xml`) in the provided code, so make sure to look those over before you begin. The main layout describes a very simple screen showing a pile of `TextViews` organized in a `RelativeLayout`.

7.1 Defining Styles

If you look at the `TextViews`, you'll see that they share a lot of the same attributes: text sizing, width and height, boldness, etc. If you decided that all of the text should be bigger (e.g., for readability), then you'd need to change 6 different attributes—which is a lot of redundant work.

Enter **Styles**. Styles ***encapsulate a collection of XML properties***, allowing you to define a set of properties once and then use a single attribute to apply all of those properties to a view. This provides *almost* the same functionality as a CSS rule describing a class declaration, but without the "cascading" part.

Styles are themselves defined as an XML resource&mdsah;specifically a `<style>` element inside the **`res/values/styles.xml`** file.

- This XML file was created for us when Android Studio created the project. Open the file, and you can see that it even has some initial content in it!
- Style resource files, like String resource files, use `<resource>` as a top-level element, declaring that this XML file contains (generic-ish) resources to use.

Styles are declared with a `<style>` tag, which represents a **single** style. You can *almost* think of this as a "class" in CSS (though again, without the cascading behavior).

- The `<style>` element is given a `name` attribute, similar to how we'd define a CSS class name. Names are normally written using PascalCase: they need to be legal Java identifiers since they will be compiled into `R`, and since they are "classes" we capitalize them!
- We'll discuss the `parent` attribute in the starter code in the next section.

We define `<item>` elements as *nested children* of the `<style>` element. Each `<item>` represents a *single* attribute we want our style to include, similar to a single *property* of a CSS rule.

- `<item>` elements get a `name` attribute which is the the name of the property you want to include. For example: `name="android:layout_width"` to specify that this item refers to the `layout_width` attribute. The *content* of the `<item>` tag is the value we want to assign to that attribute, e.g., `wrap_content` (not in quotes, because the content of an XML tag is already a String!)

Finally, you can specify that you want a particular `View` (e.g., in your `layout`) to have a style by giving that View a `style` attribute, with a value that references the style that you've defined (using `@style/...`, since this resource has type "style").

- Note that the `style` attribute does **not** use the `android` namespace!

Practice: Define a new style (e.g., `TextStyle`) that specifies the attributes *shared* by the 6 `TextViews`: the text size, the width, and the height. Additionally, have the style define the text color as UW purple. Then, refactor these TextViews so that they use the style you just defined *instead of* duplicating attributes. See how much code you've saved?

- After you've done that, go ahead and change the size of *all* the `TextViews` to be `22sp`. You should be able to make this change in exactly one place!

Style Inheritance

This is a good start, but we still have some duplicated attributes—in particular, the "labels" share a couple of attributes (e.g., they are all bold). Since each View can only have a single style and there is no cascading, if we wanted to create a

separate `LabelStyle` style, it would end up duplicating some attributes of the `TextStyle` (size and color). Ideally we would like to not have to redefine a style if it only changes a little bit.

Luckily, while styles don't cascade, they can **inherit** items from one another (a la Java inheritance, e.g., `extends`). We can establish this inheritance relationship by specifying the `parent` attribute for the `<style>`, and having it reference (with `@`) the "parent" style:

```
<style name="ChildStyle" parent="@style/ParentStyle"> ... </style>
```

This will cause the `ChildStyle` to include all of the `<item>` elements defined in the `parent` style.

- We can then "override" the inherited properties by redefining the `<item>` you want to change, just like when inheriting and overriding Java methods.

When inheriting from our own *custom styles* (e.g., ones that we've defined within the same package), it's also possible to use **Dot Notation** *instead of* the `parent` attribute. For example, naming a style `ParentStyle.ChildStyle` will define a style (`ChildStyle`) that inherits from `ParentStyle`. This would be referenced in the layout as `@style/ParentStyle.ChildStyle`. The dot notation is used to "namespace" the inherited class as if it were a "nested" class we wanted to reference.

- We can chain these together as much as we want: `MyStyle.Red.Big` is a style that inherits from `Red` and `MyStyle`. However, this style cannot also be referenced as `MyStyle.Big.Red` style—it's not using a CSS class selector, but Java class inheritance!
- Note that often name style classes based on this namespaced inheritance, so the "child class" is named after an adjective (e.g., `Big`) that is used to describe the appearance change of the parent element. `Text.Big` would be an appropriate style naming convention.

Pratice: Define another style (e.g., `Label`) that **inherits** from your first style to encapsulate attributes shared by the labels (e.g., boldness). Refactor your `layout` so that the labels use this new style.

Define *another* style (e.g., `Gold`) that **inherits** from your `Label`'s style and has a background color that is UW Gold and a text color of black. Apply this style to *one* of your labels.

It is best to utilize styles for elements that share **semantic meaning**, not just specific attributes! For example, buttons and labels that will be duplicated, headers shared across screens, etc. This will help you avoid needing to frequently change or overwrite styles in the future just because you want to make one button look different; changes to the style should reflect changes to the appearance of semantic elements. This is the same guideline that is used for determining whether you should define a CSS class or not! This blog post has

a good summary.

Built-in Styles

Android also includes a large number of built-in platform styles that we can apply and/or inherit from. These **must** be inherited via the `parent` attribute (you can't use dot notation for them). They are referenced with the format:

```
<style name="MyStyle" parent="@android:style/StyleName">...</style>
```

There are a bunch of these, all of which are defined in the R.style. This makes discoverability difficult, as not all of the styles are documented. To understand exactly what style effects you're inheriting, Android recommends you browse the source code and seeing how they are defined.

- Yes, this is like trying to learn Bootsrap by reading the CSS file.
- Author's opinion: most of the styles are not very effective bases for inheritance; you're often better using your own.

Practice: Define a new style for the `Button` at the bottom of the screen that inherits from the built-in `MediaButton` style (but give it a text size of `22sp`). What does the inheritance do to the appearance?

Styles for Text Views

If your style primarily is defining text appearance attributes such as font size or text color, you alternatively have the option to utilize the `android:textAppearance` attribute instead. This attribute is available only on `TextView` elements (including subclaseses such as `EditText` or `Button`), and should be assigned a reference to a `<style>` element.

Moreover, a View can have **both** a `style` *and* a `textAppearance`—in effect letting you apply two different styles to the same element!

```
<!-- for example, inherit from built-in TextAppearance style -->
<style name="MyTextAppearance" parent="TextAppearance.AppCompat">
    <item name="android:textColor">#0F0</item>
    ...
</style>
```

```
<TextView
    style="@style/MyStyle"
    android:TextAppearance="@style/MyTextAppearance" />
```

The intention of this attribute is to let you specify different broad styles for Views (e.g., whether buttons should be rounded, the backgrounds and padding of cards), but still be able to have unified text styling (font a size), as well as

different "types" of fonts (e.g., `Large` or `Small` text). In practice, styles that purely modify text appearance should be applied view `textAppearance`, while all other styles should be applied via the `style` attribute/

7.2 Themes

Unlike CSS, Android styling is *NOT* inherited by child elements: that is, if you apply a style to a `ViewGroup` (a layout), that style will not be applied to all the child components of that `ViewGroup`. Thus you can't "style" a layout and have the styling rule apply throughout the layout.

The option that is available is to apply a that style as a **Theme**. Themes are styles that are applied to ***every*** `View` in a `Context` (an *Activity* or the whole *Application*). You can't get any finer granularity of style sharing (without moving to per-View *Styles*). Theme styles will apply to *every* View in the context, though we can overwrite the styling for a particular View as normal.

Themes *are* styles, and so are defined the exact same way (as `<style>` elements inside a resource XML file). You can define them in either `styles.xml`, `theme.xml`, or any other `values` file—resource filenames are arbitrary, and their content will still be compiled into `R` no matter which file the elements are defined in.

Themes are applied to an Activity or Application by specifying an `android:theme` attribute in the `Manifest` (where the Activity/Application is defined). If you look at the starter project's `Manifest` created by Android Studio, you'll see that it already has a theme (`AppTheme`). In fact, this is the `<style>` that was provided inside `styles.xml`!

Practice: Experiment with removing the theme attribute from the application. How does your app's appearance change? *NOTE*: you will need to change `MainActivity` to subclass `Activity`, not `AppCompatActivity`, in order to fully adjust the theme.

- You might also try commenting out the stylings you applied to the individual `TextViews` to *really* see what happens.

Material Themes

Along with built-in styles, Android provides a number of platform-specific themes. And again, the somewhat unhelpful recommendation is to understand these by browsing the source code, or the list in R.style (scroll down to constants that start with `Theme`).

One of the most useful set of Android-provided themes are the **Material Themes**. These are themes that support Google's Material Design, a visual

design language Google uses (or aims to use) across its products, including Android. Material themes were introduced in Lollipop (API 21) and so are only available on devices running API 21+ (though there are compatibility options).

There are two main Material themes:

- `@android:style/Theme.Material`: a Dark version of the theme
- `@android:style/Theme.Material.Light`: a Light version of the theme

And many variants:

- `@android:style/Theme.Material.Light.DarkActionBar`: a Light version with a Dark action bar (Dark backgroud, Light contents)
- `@android:style/Theme.Material.Light.LightStatusBar`: a Light version with a Light action bar (Light background, Dark contents)
- `@android:style/Theme.Material.Light.NoActionBar`: a Light version with no action bar.
- … etc. See R.style for more options (do a ctrl-f "find" for `Material`)

Practice: Experiment with applying different material themes to your application How does your app's appearance change? Give your application a Dark Material theme!

Theme Attributes

One of the big advantages of Themes is that they can be used to define **theme attributes** that can be referenced from inside individual, per-View Styles. This allows the Theme to effective "skin" the View. For example, a Theme could define a "color scheme" as a set of attributes; these attributes can then be referenced by the Style as e.g., "the primary color *of the current theme*".

Theme-level attributes are referenced in the XML using the `?` symbol (in place of the `@` symbol). For example: `?android:textColorPrimary` will refer to the value of the `<item name="textColorPrimary">` element inside the Theme.

Indeed, one of the advantages of the Material Themes is that they are implemented to utilize a small set of color theme attributes, making it incredibly easy to specify a color scheme for your app. See Customize the Color Palette for a diagram of what theme attributes color what parts of the screen.

- Note it is possible to apply a Theme to an individual `View` or `ViewGroup`, causing the theme attributes (and *ONLY* the theme attributes!) to be "inherited" by any child elements. This allows you to specify color palettes for specific parts of your layout.

Practice: Redefine the colors in your custom Styles (from the first practice steps) so that they reference the theme attribute colors instead of purple and gold. What happens now when you change the application's Material Theme between light and dark?

- Can you have the logo image reference those color theme attributes as well? Hint: use the `tint` attribute.

Practice: Modify the provided* `AppTheme` *style (in* `styles.xml`*) with the following changes:

- Have it **inherit** from a Material Theme (your choice of which)
- Have it define theme attribute colors using the UW colors (purple and gold). Use these theme attribute colors to "Huskify" your app (including the colors in your custom Styles)

Finally, set the theme of your app back to `AppTheme`—and you should now have a UW flavored app!

Resources

- Styling Views on Android Without Going Crazy (blog post)

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/appendix-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):

```
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 :

```
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. ***Prac-***

tice: 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:

```
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:

```
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

Java 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, it can be useful to consider how to build simple graphical applications in Java using a different GUI framwork: the Swing library.

This appendix references code found at https://github.com/info448/appendix-java-swing. Note that this tutorial involves Java Programming: you can either do this in Android Studio, or just using 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 definition (including methods to override) 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.