This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Android

What to Do

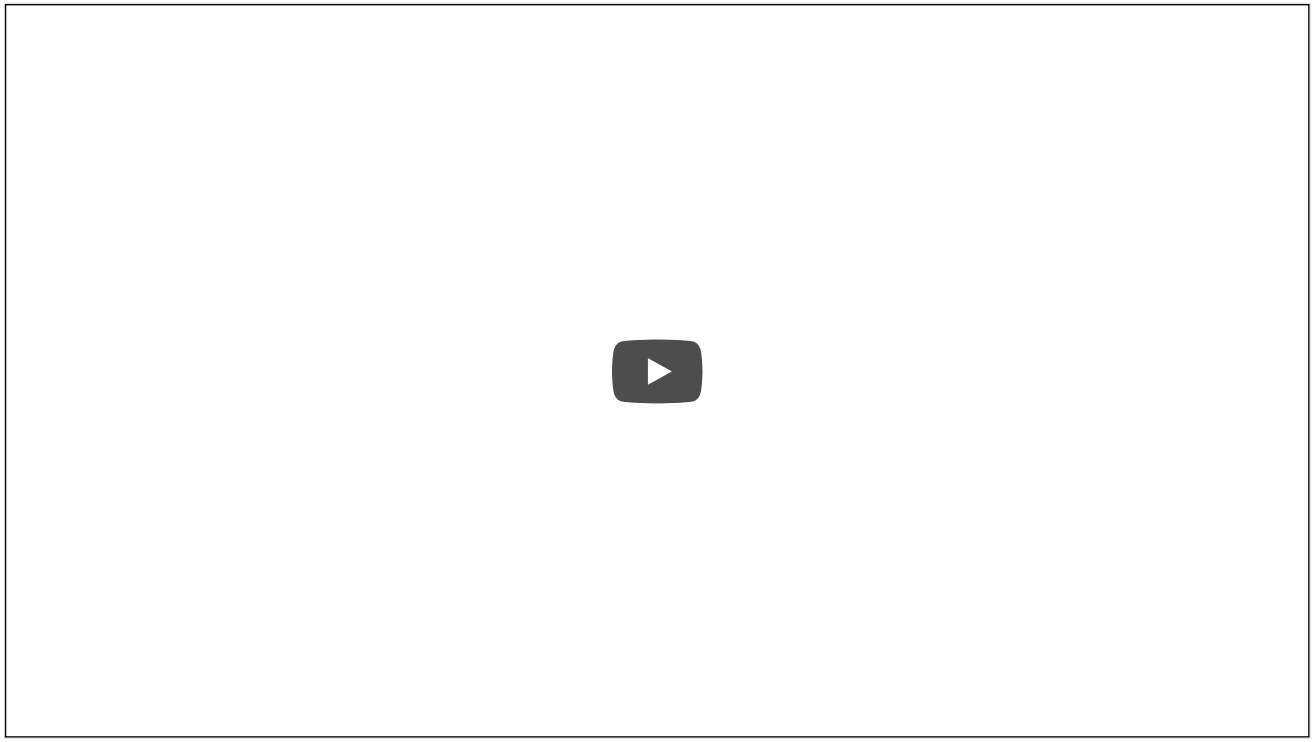
1. After watching Lessons 1, 2, and 3, submit Pokédex.
2. After watching Lesson 4, submit Fiftygram.
3. After watching Lesson 5, submit Notes.

When to Do It

By 11:59pm on 31 December 2020.

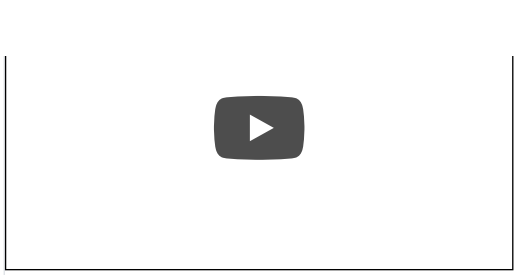
How to Do It

▼ **Introduction**



- We'll learn to write mobile apps for Android with a new language, Java, and build three apps: one that loads data and displays it; one that applies filters to images; one that lets you take notes and save them.

▼ **Lesson 1**



- We'll use Android Studio, an IDE provided by Google to help us write Android apps. We'll download and open it, and start a new project. We'll select the Empty Activity template for our app, and use `JavaExample` for our app name. A convention for the package name is the domain name in reverse, plus the app name, like `edu.harvard.cs50.javaexample` . We'll use Java and support Android 5.0 or above, so most devices can use our app.
- [2:20] Inside Android Studio we'll see a lot of files that have been generated for us. We'll want to first create an AVD, or Android Virtual Device, so we can run our app on our laptop instead of a separate device.
- [4:00] We'll take a look at the syntax for Java, which is similar to C and has familiar data types. We can initialize and change variables, have conditions, arrays, for loops, Lists (like a dynamically-sized array),
  - We can create a List object with `List<String> values = new ArrayList<>()` , specifying that the type of data it will hold is `String` . Java supports many different types of Lists, but we'll just use `ArrayList` .
  - We can also iterate over the elements in our List with `for (String value : values) {` .
- [9:40] Java has the concept of generics, a way for a type like List to understand the type inside, in this case String. Java also has maps, also called dictionaries in Python or objects in JavaScript, which stores key-value pairs.
- [12:45] Lists and Maps are examples of classes in Java, which we can think of as structs with functions attached to them. And we call functions inside classes *methods*.
- [15:25] We can add a method to our `Person` class, and use a variable in our method that the constructor in the class already saved.
- [17:10] We can also have static methods, or methods we can call without an instance (one that we constructed) of our class.
- [17:50] Another feature of classes is inheritance, where we can inherit fields and methods from a parent class, and optionally modify some of them.
- [20:40] Interfaces are like a list of methods that any class implementing them has to have. If a method is missing, the compiler will tell us. It turns out that declaring a list with `List<String> strings = new ArrayList<>()` is actually using an interface, where `List` is the interface, and `ArrayList` is the class that will implement the behaviors of a list. With `new ArrayList<>()` , we're creating an instance of the class.
- [24:20] Java also has packages, which helps us organize and namespace our files. Like in Python, we also need to `import` certain packages we want to use.
- [26:10] We'll come back now to Android Studio and look at the project we created. Inside the `java` folder, there's our package with a `MainActivity.java` file that has one class and a method, `onCreate` , that calls `super.onCreate` first (which is the parent class' implementation), and then calls `setContentView` , which we won't worry about now.
- [29:20] We can start by creating a new Java Class by right-clicking on our package folder on the left. We'll name it Track, to represent the tracks in our course, and now we can add fields and a constructor that saves its arguments.
- [32:00] Back in our `onCreate` function, we'll create a new `List` of `Track` s and add some tracks. We'll create a list of strings with a static method, `Arrays.asList` , to represent student names.
- [35:40] Now we'll use a map, of strings to tracks, to represent track assignments for students. We'll iterate over every string in our list of strings and use the `Random` class to get a random track for each of them.
- [39:00] To print everything, we can iterate over the map, and use the `Log` class in Android Studio to print out a log. We'll press the play button on the top right, and the device shows the default "Hello, world". But in Android Studio, on the bottom right we can click Logcat, inside which we can see our logs.
- [43:00] We'll add getter methods to our `Track` class to return its fields, so they can be private to the class and so other code can't change

them directly. Our getter might also have other logic.

▼ **Lesson 2**

- Now we'll add UI to our Android app. The build system is called Gradle, which helps us by downloading libraries and compiling our code.
- MVC, Model-View-Controller, is a general design pattern in which we separate our concerns, or types of code, into three categories. Models, like the Track class we created in lesson 1, stored our data. The view takes care of displaying the data when it gets it. And finally, the controller is the bridge between the model and the view, with logic deciding what data to pass to the view and when.
- An Activity in Android is like a base class for each of our screens in the app, representing a single thing we're trying to do. For example, in a contacts app the first activity might be the list of contacts we see, and the second activity is the view of a single contact.
- Our app will also have Resources, non-Java code such as Layouts that describe how a view should look. Layouts are in a language called XML, which looks similar to HTML, with tags and attributes. For example, we can specify a `LinearLayout` with a `TextView`.
- [6:15] Another concept we'll see is called Intents, that let us move from one activity to another. We'll also have Recycler Views, which displays a list of items that we can scroll through like a feed.
- [8:15] We'll create a new project again, an Empty Activity, to display a list of Pokemon and details about each of them. We'll set up our project and take a look at the generated files:
  - `AndroidManifest.xml` contains some configuration for our application, like an icon and the activities in our app.
  - In the `java` folder, we'll have our `MainActivity` file, but also packages for test files.
  - In the `res` folder, we'll see resources. In particular, the `layout` folder will have view for each activity, and if we open `activity_main.xml`, and we can see a UI to drag and drop component. We can also click the Text tab at the bottom to see the source XML. Important attributes include `layout-width` and `layout_height`, so we can choose to fill the entire screen or only some fraction of the parent view.
  - In the `values` folder, we can also add constants like strings for translation.
- [17:15] We'll look at some Gradle scripts, which specify flags and dependencies that's like a configuration file for the Java compiler. In the `dependencies` section, we can add more libraries as we use them.
- [19:15] Now we'll come back to `MainActivity` and add a `RecyclerView`. Android's developer documentation has a lot of details and examples that we can learn from. We'll add the library in the Gradle file, and click Sync in Android Studio to automatically download the package. Then we'll change our `activity_main.xml` layout file to use a `RecyclerView` instead of the `TextView`. We'll add an identifier to the view so we can reference it from our controller with `android:id="@+id/..."`.
- [23:55] We also need to define what each row looks like, so we'll need to create a new Layout resource file in the same folder, and create a new `LinearLayout`. We'll add a `TextView` inside, and give both IDs.
- [26:20] Our view is ready, so we'll create some classes for our models. First, we'll create a `Pokemon` class with properties and a constructor to save them.
- [28:35] It turns out that a `RecyclerView` uses another class, an adapter, to control what data will be displayed, so we'll create a new class `PokemonAdapter` that extends the `Adapter` class. We'll also need what's called a view holder so we can modify the view and layout as needed. In our adapter, we'll define the `PokedexViewHolder`, which will have the generic row view, but also the `LinearLayout` and `TextView` we added earlier inside each row. We'll need a class `R.id` to get the unique ID for each of those views. Then we'll override

`onCreateViewHolder`, after our view holder is created, to create our view that represents a row. We'll also need `onBindViewHolder` to set the values of each row, given a position of the row.

- [40:35] In our `MainActivity` file, we'll add fields for our view and adapter, and also a `LayoutManager`. Now we can get the view in our main layout and connect our adapter to it. We can run our project now, and see that each row takes up the whole screen, so we'll change the layout's width and height to be `wrap_content`.
- [44:00] We'll create another activity, so we can display each Pokemon when they're selected. We'll create a new Activity > Empty Activity, and generate a layout file. We'll change the layout to a simpler `LinearLayout`, and add some `TextView`s inside, setting the text size and padding.
- [47:45] In our `PokemonActivity` view, we'll set the values we get from an Intent onto those views. We can call `getIntent().getStringExtra()`, built into `AppCompatActivity`, to get the variables passed into our view. We'll need to pass along data in our adapter in the `onBindViewHolder` method, with `setTag` on the view to set the current `Pokemon` object to our view holder. And we'll add an event listener, `setOnClickListener`, to get our `Pokemon` back from the tag and create an Intent we can pass to the next view with `startActivity.`
- [55:25] We can make our view nicer in the layout XML file with some more attributes like padding, built-in animations, and centered text. And like `printf` in C, `String.format` in Java can take in a format string to give our number a certain number of digits.

▼ **Lesson 3**

- Now we'll load data from the internet for our app.
- We can use an API, application programming interface, to load data from the internet in our app. An API is like a set of code that someone else has written, designed for you to use too.
- In this case, we'll be making requests to a website and getting data back in a format called JSON, JavaScript Object Notation.
- An object in JSON might look like a dictionary of key-value pairs:

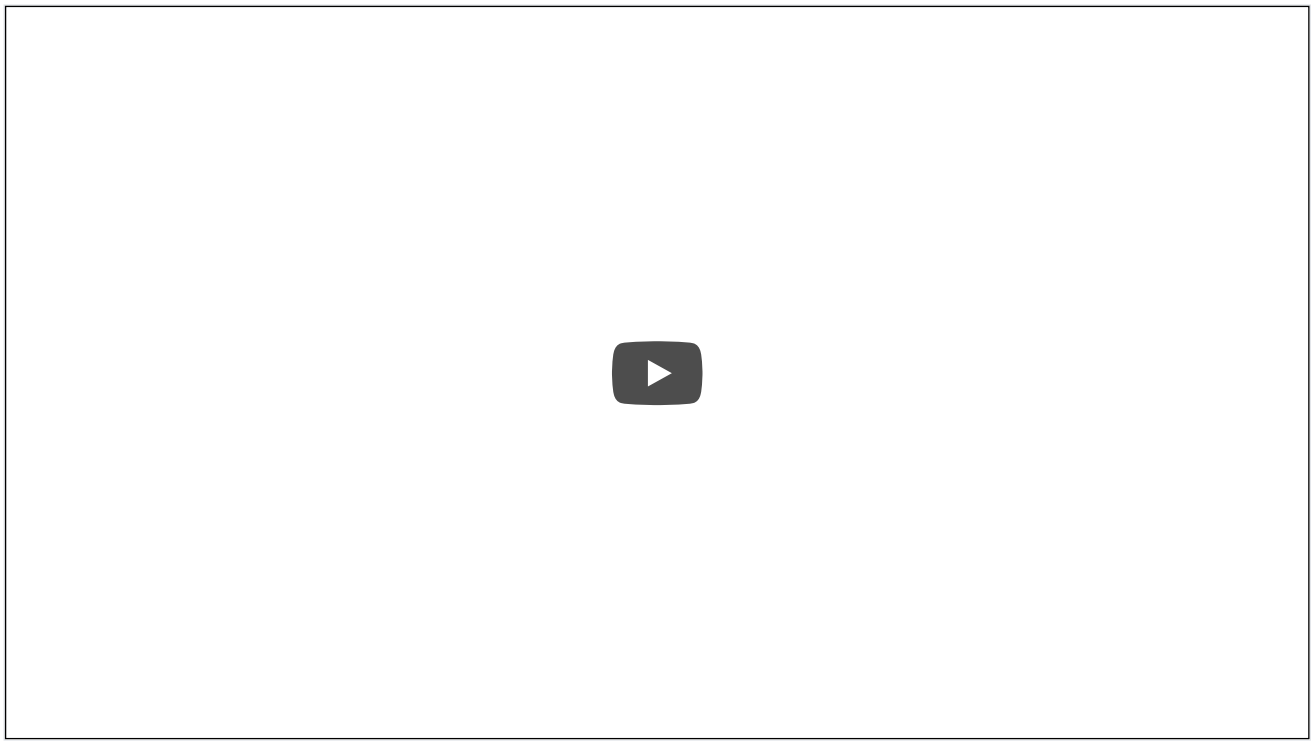
```
{
  "course": "cs50",
  "tracks": ["mobile", "web", "games"],
  "year": 2019,
}
```

  - The values can be a string, an array, or a number as we see here, or some other data types.
- [2:10] We'll check out PokeAPI at pokeapi.co, and see that a URL we put in will return a lot of data in the format of JSON. The documentation for the website has information about how we can get a list of data, so we try that.
- [5:25] Android has a library called Volley for making requests, and we'll include it in our `build.gradle` file so we can use it.
- [6:35] We'll also need to use a new pattern called try, catch, where a function that might fail or have an exception, can be "caught", or recovered from. When we catch an exception, we get an object of the `Exception` type, and we'll be able to print out details of exactly what happened.
- [8:30] In our adapter, we'll load our `pokemon` list with a new method, `loadPokemon`, and use the Volley library to make a `JsonObjectRequest`. We'll have an anonymous method that will be called once we get a response from the API, with a `JSONObject` that

we can parse into an array of `pokemon`. Since the response might not be what we expect, we'll need to catch any exception we might get.

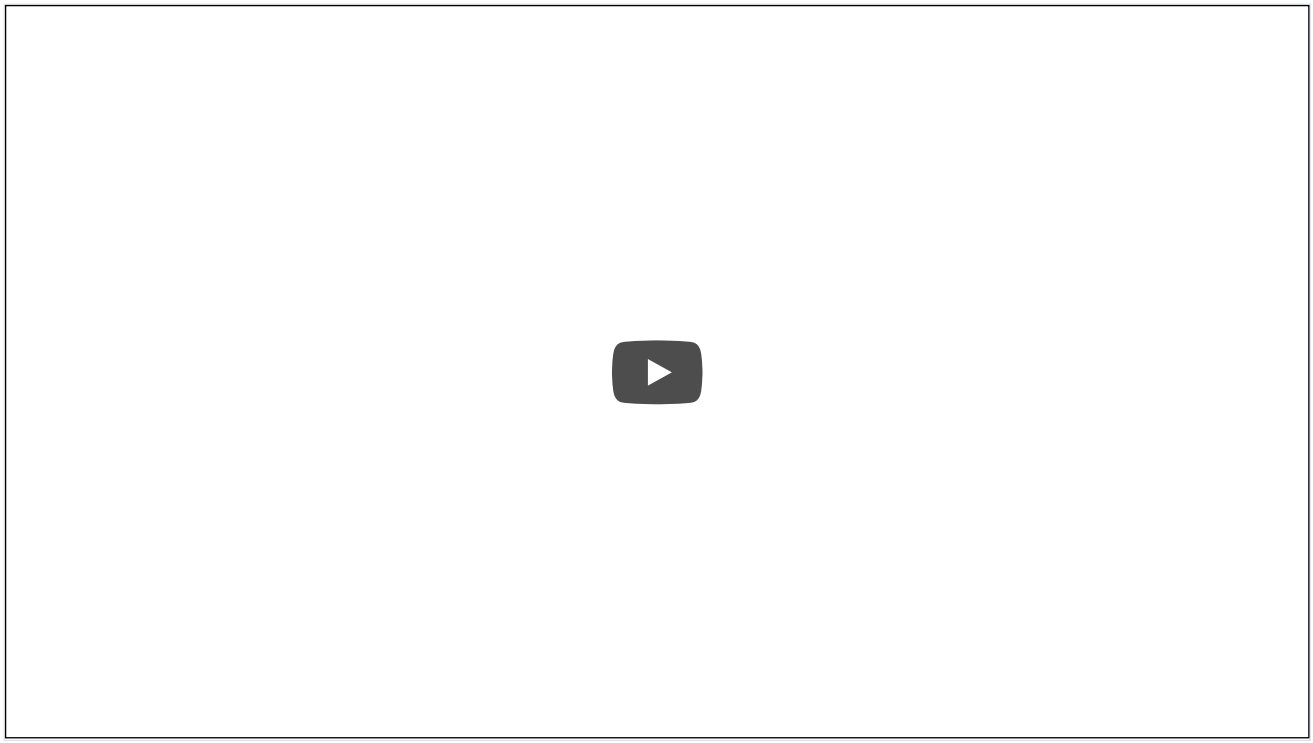
- [14:40] Each result in the `JSONArray` will be a `JSONObject`, and similarly we can try to get the `name` and `url` from each of them and put them in our `Pokemon` class.
- [17:50] Once we've defined our request, we also need a `RequestQueue` that has the `Context` of the app, so the Volley library can make requests on behalf of our app properly. We'll add our request to the queue, so we can actually load data.
- [20:25] After we load our data, we need to refresh our RecyclerView with `notifyDataSetChanged()` from our adapter, and we also need to add the permission to use the internet for our app in `AndroidManifest.xml`. We'll fix the capitalization of the name.
- [24:40] We'll use the `url` on the Pokemon data we got back to make another request when we want details about a particular Pokemon. We'll look at the view, and add two more `TextView`s to display the types of the Pokemon. In the Activity class for the Pokemon details view, we'll make another request, and parse the object in the response for the `types` array. Then, we can set the values of the all the `TextView`s based on the data.
- [33:30] The `PokemonActivity` class will need the `url` from our adapter, so our click listener can pass that in with the `Intent` object. And our activity also needs to call our new `load` function, and actually make the request.

▼ **Lesson 4**



- We'll build another app now, one that allows us to apply filters to images. We'll create a new project with an Empty Activity in Android Studio, and start by adding more views in our `activity_main.xml` layout. Since we'll need to scroll, we'll change the parent layout to a `ScrollView`, and inside have a `LinearLayout` for our `ImageView` and a `Button`.
- [4:40] In our activity, we'll add the functionality for our button to load an image and display it. We'll write a `choosePhoto` method, which will call the built-in Android image gallery for selecting a photo. We'll create an `Intent` and set the action and type. We'll add a `requestCode`, so we know how to handle the selected image in our app when the other activity finishes. Finally, we'll have our `Button` call the method when it's clicked.
- [10:15] In our activity, we'll need to override the `onActivityResult` method to actually handle the data (image file) we get back. We'll make sure that the `resultCode` is okay, and that we got data back. Then, we'll have some steps to get an image from the `data` object, by getting the URI (like a URL), trying to open the file from it, and loading the file as a bitmap image.
- [15:40] We'll use the `BitmapFactory` to create our image object by decoding the file, and then close the file. Finally, we can set the image on our `ImageView` to show what we've picked.
- [18:05] We'll use some third-party libraries, like `glide-transformation`, by adding them to our Gradle file by following their documentation. And we need another library for some of the filters we want, so we'll add that too.
- [20:40] We'll add a button to our view for applying a filter, and create methods in our activity by following the documentation. We'll use the example of loading an image into the `Glide` library, applying a transformation, and loading it into the `ImageView`.
- [25:05] We'll add two more in the same way, and factor out the common code, and just pass in different transformations depending on which filter we want to apply. We also have to be careful with importing the right classes from the right packages. Now we can apply different filters to our images.

▼ **Lesson 5**



- We'll build a note-taking app that can save data to the device.
- We'll use SQLite, a simple database that saves data to a file but supports SQL queries.
- We'll need queries like:
  - `CREATE TABLE`
  - `INSERT INTO`
  - `SELECT ... FROM`
  - `UPDATE ... SET`
- [4:25] We'll open a new project again, with Empty Activity, and start by creating two views, one with a list of notes, and one for editing an individual note. We'll make a `RecyclerView` as before, and create an adapter to provide data for the view. We'll also need another view for each row, `note_row`, similar to our Pokemon app. Inside our adapter we'll create a view holder to be able to set data on the views.
- [11:05] We'll look at the documentation for Android's persistence (data storage) library, called Room. We'll need to add the dependencies to our Gradle file, including an `annotationProcessor` for our compiler to generate the library's code.
- [12:50] We'll make a new model class, `Note`, with an `id` and `contents`. We'll also add some annotations, like `PrimaryKey` and `ColumnInfo` to specify how these fields will be stored in our database by the Room library. This is essentially the definition of the table.
- [14:55] In our adapter, we'll override `onBindViewHolder` to get the contents of our note from the notes list, and `getItemCount` for the total size of the list. In our main activity, we'll connect our view, layout, and adapter with each other.
- [18:15] Now that our view is ready, we'll write a new class, DAO, for data access object, with the Room library, so we can actually load and save note objects to our database. We'll make a new class, `NoteDao`, which will actually be an interface that we annotate, and the Room library will generate the actual code implementing these queries. For example, we'll add an annotation, `@Query`, to the `create()` method in our interface, without actually writing any code for it. Instead, we'll write our SQL query in the annotation. Similarly, we can write a method `getAllNotes` to return a list of notes. And in our `save` method's query, we can easily use `:contents` and `:id` to safely substitute variables into our query, avoiding SQL injection attacks. This class is how we'll interact with our table.
- [23:50] To use our DAO, we need a database class, and we'll call it `NoteDatabase`. This will specify the database that our DAO can use. It turns out that our database class is an abstract class, which means it has some methods that are implemented, and some methods that are not, or abstract. Again, the Room library will generate the code that implements our abstract class.
- [28:55] We can use all of this code in our main activity by first connecting to this database and storing the connection as a `public static` variable so all of our activities can share it. In our adapter, we can write a `reload` method to access the database and use the noteDao's `getAllNotes` method.
- [32:35] And in our activity, after our view loads, we'll run this `reload` method. We'll also add a button to our layout with Google's Android Material UI package, and specify some attributes of it so it looks the way we want.
- [37:00] We'll add an `onClickListener` in our main activity to the button, which should create a new note, and reload the recycler view to show it.
- [38:35] Now we can create a `NoteActivity` to allow us to edit a specific note, and we'll use a `EditText` component in the layout to hold the contents of our note. In our activity, we'll want to load the contents of the note into the text editing container, and we'll also save the contents to the database when we go back to the recycler view in the app. We can load the note from the intent, and override `onPause` to

save the note when we leave the activity.

- [42:45] In our recycler view's adapter, we'll set the `onClickListener` for each row's container to create an Intent with the row's note, and pass it to our `NoteActivity`. Finally, when we come back to this view, we'll also want to reload the notes by `onResume`.
- [46:30] When we build and run our app, we see a crash, and the log tells us that our current note is `null` in the adapter, and it turns out that we have to check for it after the view has been loaded, in the click event handler, not in the constructor.
- [47:55] Finally, we'll clean up the layout by adding padding and other aesthetics.

▼ **Conclusion**

- The Android documentation has lots of topics, so do use it to build even more interesting apps!