TABLE OF CONTENTS

► Advanced-UI.md

· Animation.md

· Basic-UI.md

· Data.md

· Debugging.md

· Drawing.md

· Fragments.md

· Home.md

· Introduction.md

· Networking.md

· Threading.md

· Unit-Testing.md

Advanced-UI.md

Goal

Add some common UI elements, learn about databinding and multi-page applications.

Table of Contents

- Enable Databinding
- Layout Changes
  - findViewById
- RecyclerView
  - RecyclerView.ViewHolder
  - Reading List UI
  - View Models
  - Adapter
  - Setting up the RecyclerView
  - Divider
  - Multiple View Types
- Click Handling
- Two-Way Databinding with EditText
- Saving Changes
  - Reloading changes
  - Add new entries

Up until now, our application has still resembled a Hello World app. Throughout this chapter, you will transform it into our Reading List application.

Enable Databinding

Open your module-level `build.gradle` and add a new `dataBinding` closure inside the `android` closure. Remember that it is case sensitive.

```
android {
    dataBinding {
        enabled = true
    }
}
```

Sync your application.

Layout Changes

The most obvious change you will see initially is around the layout files. Let's take our existing layout and do a minimal conversion for it.

Open your `activity_book_list.xml`.

Currently, it looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/black"
        tools:context=".BookListActivity">
...your text and image views...
</androidx.constraintlayout.widget.ConstraintLayout>
```

We are going to make a new top-level element, and this ConstraintLayout will become a child of it (remember, only one top-level element can be present).

The databound format looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns-from-original-layout-here>
    <data>

    </data>
    <your original layout here... />
</layout>
```

- Select everything (except the `<?xml ... ?>` line) and Ctrl+X to cut it.
- Add a new `<layout></layout>` element and press enter between the start and end tags.
- Add a new `<data></data>` element inside of that; and press enter between those start and end tags.
- Between `</data>` and `</layout>` use Ctrl+V to paste the old layout back inside.
- Highlight all the `xmlns` attributes on the `ConstraintLayout` and cut them
- Go inside the `<layout>` element and press space or enter. Paste the `xmlns` attributes (like `<layout xmlns...`)

When we are done, the end result looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/black"
        tools:context=".BookListActivity">
...your text and image views...
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Make sure `app` is selected and redeploy your application. Let's make sure we haven't broken anything yet.

Congratulations! You've just auto-generated your first databound model.

findViewById

So what can it do for us?

Remember our conversation about `findViewById`? The way that function works is by iterating over all your views until it finds the one with the correct identifier. When your application calls that function over and over and over again, you introduce a lot of lag.

We can improve the speed and readability at the same time using databinding.

When we modified that layout, we instructed the build to auto-generate a new class, `ActivityBookListBinding` (based on `activity_book_list.xml`). This auto-generated class acts as a view holder for any variables defined in the layout and keeps a list of all the ids we specified.

If you `Navigate | Class` and enter `ActivityBookListBinding` you will see the newly auto-generated class. You will also notice a few familiar `public final` variables at the top.

For now, close that class and just remember that re-deploying our application re-generates those files.

Let's see how we can use it.

Open your `BookListActivity`. First we are going to add a new variable

```
private lateinit var binding: ActivityBookListBinding
```

> If you try to copy/paste that line and it doesn't recognize the class, remove it and try typing it out instead.

The `lateinit` tells the system that we *will* give it a value later. It's not set yet, but we can not set it to null.

Then, in our `onCreate`, replace this line

```
setContentView(R.layout.activity_book_list)
```

with

```
binding = DataBindingUtil.setContentView(this, R.layout.activity_book_list)
```

While we didn't really *have* to save that binding for later, it can come in really handy.

Now, our first findViewById looks like this:

```
findViewById<TextView>(R.id.hello_text).text = getBooksLoadedMessage(it.books.size)
```

We are going to replace that line with

```
binding.helloText.text = getBooksLoadedMessage(it.books.size)
```

Notice that it replaced the underscores with camel cases.

Let's do the next one. Replace

```
findViewById<TextView>(R.id.login_text).text = resources.getString(R.string.last_login, displayForma
t.format(time))
```

with

```
binding.loginText.text = resources.getString(R.string.last_login, displayFormat.format(time))
```

If you ever catch yourself writing `findViewById`, stop and ask yourself if it could be databound instead.

Optimize your imports and re-deploy to validate that everything is still working.

RecyclerView

It's time to try to do something more useful with our application.

A `RecyclerView` is like a list, but when rows are off-screen, it removes them from memory. As such, they suffer less from large amounts of data than a normal list would.

There are multiple parts to setting up a `RecyclerView`. While it is one of the more complicated UI elements, it is also one of the most used.

RecyclerView.ViewHolder

The first thing a `RecyclerView` needs is an extension of `RecyclerView.ViewHolder` to hold references to each view.

But wait... doesn't our auto-generated binding class already do that?

It does, but it doesn't extend the class required by the framework. However, we can cheat. Since we know that everything we need will be auto-generated, we can create a small proxy class that will wrap the binding class and just look like a ViewHolder. The beauty of this approach is that we only need one no matter how many different ViewHolders the framework expects.

Right-click on the `readinglist` package (`com` version) and let's make a new subpackage called `ui`. It should be a sibling to your `model` package.

In it, create a new class called `DataboundViewHolder`. We are going to rely on the fact that we are using Databinding to create a generic ViewModel.

```
package com.aboutobjects.curriculum.readinglist.ui

import android.content.Context
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.annotation.LayoutRes
import androidx.annotation.NonNull
import androidx.databinding.DataBindingUtil
import androidx.databinding.ViewDataBinding
import androidx.recyclerview.widget.RecyclerView

class DataboundViewHolder<BINDING : ViewDataBinding> : RecyclerView.ViewHolder {
    val binding: BINDING
    fun context(): Context = binding.root.context

    constructor(binding: BINDING) : super(binding.root) {
        this.binding = binding
    }

    constructor(@LayoutRes layoutId: Int, @NonNull parent: ViewGroup)
            : this(
        DataBindingUtil.inflate<BINDING>(
            LayoutInflater.from(parent.context),
            layoutId,
            parent,
            false
        )
    )
}
```

Although short, it looks a little complicated. Let's walk through it.

First, the DataboundViewHolder class has a generic defined. Normally, you see these as simply `T`. We are using `BINDING` here so it is more obvious when using it. That parameter references the auto-generated binding class, like `ActivityBookListBinding` from earlier.

Next, we specify a val (not var) to store the `binding` variable. This is possible, even though it is not set on that line, because it is set from the constructors. If you had a constructor that did not set it, you would not be able to do this.

We then specify a convenience function for anyone using this class to get the `Context` associated with the root view. This could be a DecorView or some other class, but should be sufficient for things like `getString`.

We then specify two constructors. The first constructor takes an already constructed binding class (like `ActivityBookListBinding`) and just passes it in.

The second constructor is a little more complicated. It asks for the layout id and parent, inflates the xml, creates the binding class, then continues as normal.

Now, how do we use this?

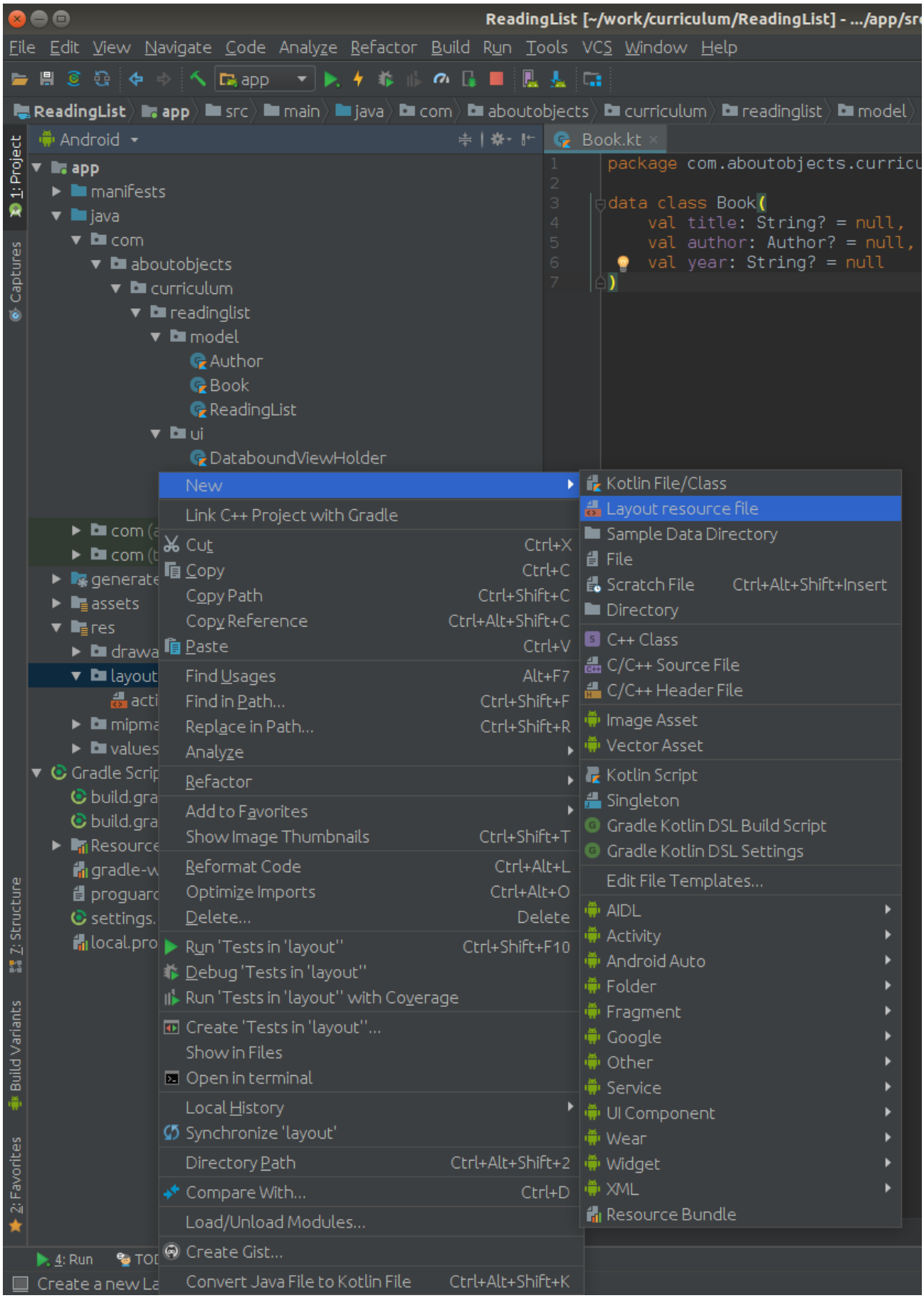
Reading List UI

Since it requires us to specify the layout id, maybe we should create the layout next?

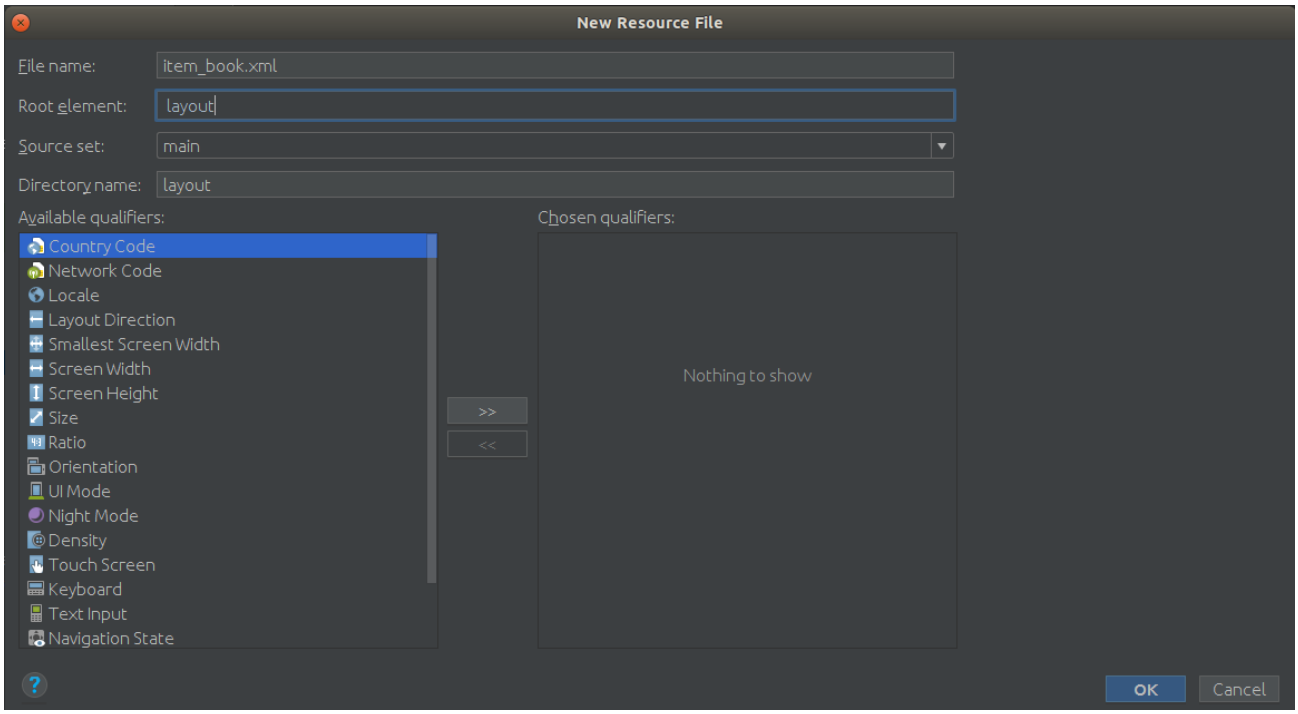
Open your `Book` model and let's re-review.

```
package com.aboutobjects.curriculum.readinglist.model

data class Book(
    val title: String? = null,
    val author: Author? = null,
    val year: String? = null
)
```

Each row in our new UI will represent a book, and thus the information in our Book model - title, author and year.



Right-click on your `res/layout` folder and `New | Layout resource file`.



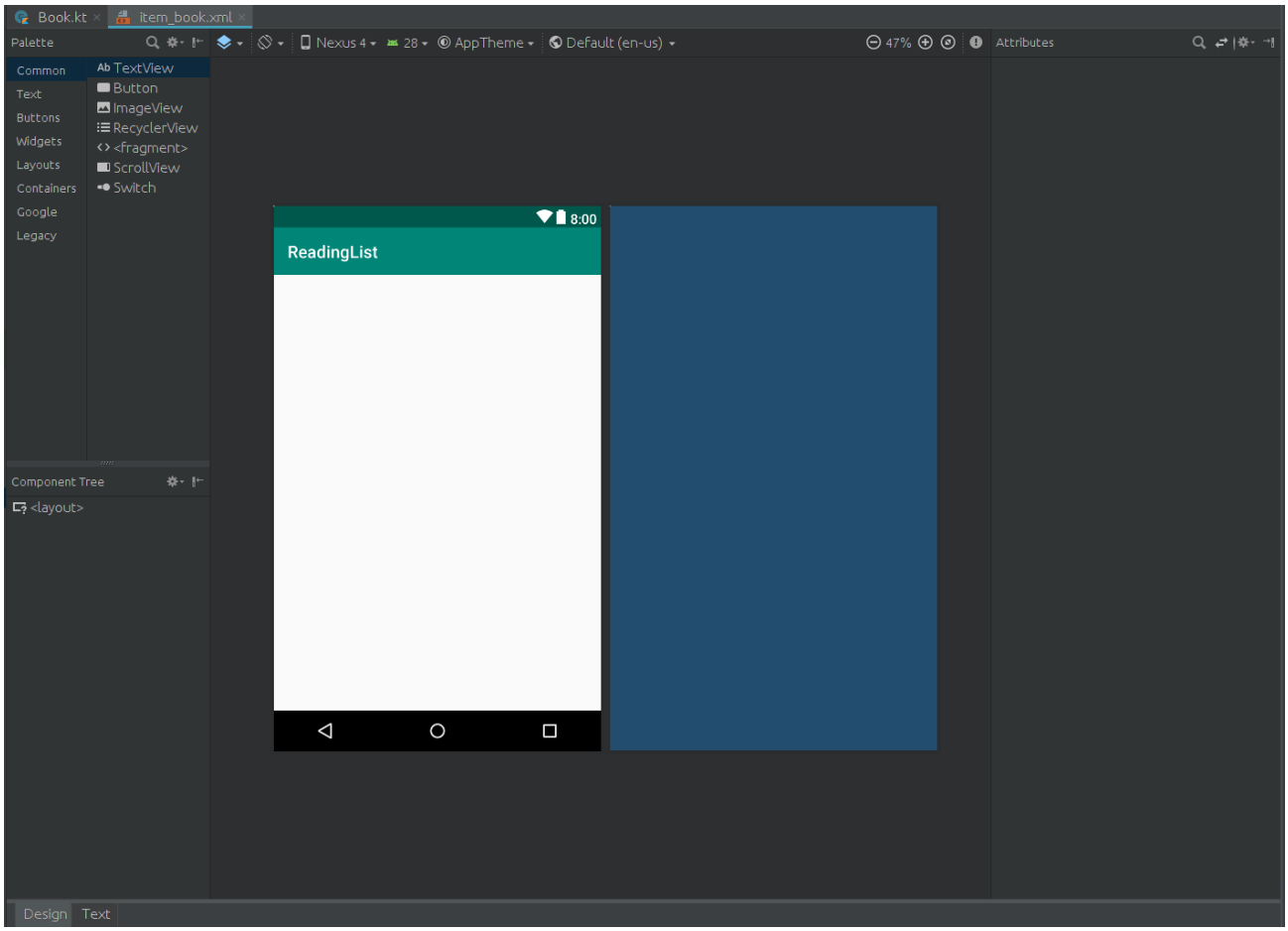
Since we are creating a layout that represents a single book, we want to name it as such. We will use `item_` to represent single rows since we could end up with different types over time. How about `item_book.xml`?

There are different types of layouts available, however we are going to be using databinding anyway. For the 'Root element' put `layout`. Note that it will provide suggestions as you type.

At this stage, we are only using the `main` source set; but this is where you would specify that a layout is only for your debug build, for example.

> While we are not going to add any qualifiers at this time, it is a good opportunity for you to learn about them. Find the `Orientation` in the list and click it. Then click the `>>` button. If it is too collapsed to make out, you can drag the bottom right corner of the window to make it larger. You will notice that as you change between `Portrait` and `Landscape`, it changes the Directory name to `layout-land` and `layout-port`. Click the '<<' to get rid of the qualifier and your directory should go back to `layout`. In this way you can specify that your UI looks different in different orientations, on different screen sizes, in different countries, on different mobile carriers, or even on different versions of Android. You can play around with these. When you are done, make sure none are selected for this exercise.

With Directory name set to `layout` go ahead and click `OK`.



You will notice that it starts in the Design view. If it didn't, click on the Design tab now.

Designing with Blueprints

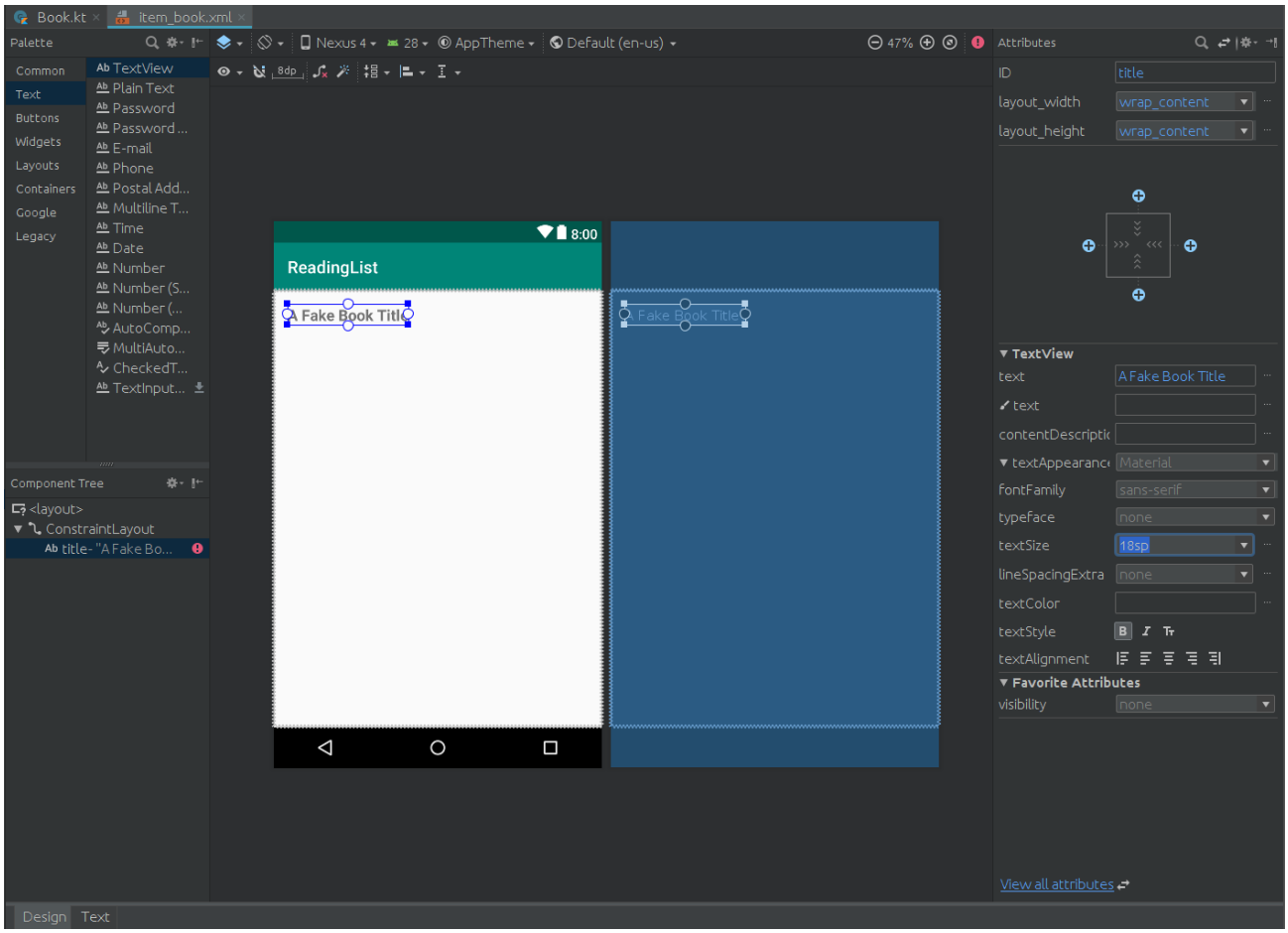
While we did all of our editing through the text editor earlier, we will use the Designer this time.

On the top-left of the designer, you will see a section called "Palette" that is divided into two columns. Below that is a "Component Tree". On the right side of the designer is the "Attributes". There is also a small toolbar above the designer. The designer itself is split into two views, the Design and the Blueprint.

In the Palette, select `Layouts` from the first column and `ConstraintLayout` in the second column. Now, click-drag the `ConstraintLayout` into the large white section of the Design view. The first thing you will notice is that there is now a border around both the Design and Blueprint. The attributes now have some information in them. The Component Tree now shows that the Constraint Layout is a child of the root layout element.

In the same way, drag a `Text` | `TextView` into the top-left corner of the white section of the Design view. You can leave a little bit of padding to make it look nice. Like before, the Component Tree now shows the TextView is a child of the Constraint Layout.

The Attributes view has been updated to reflect the information about this particular view. Let's tweak some of that.

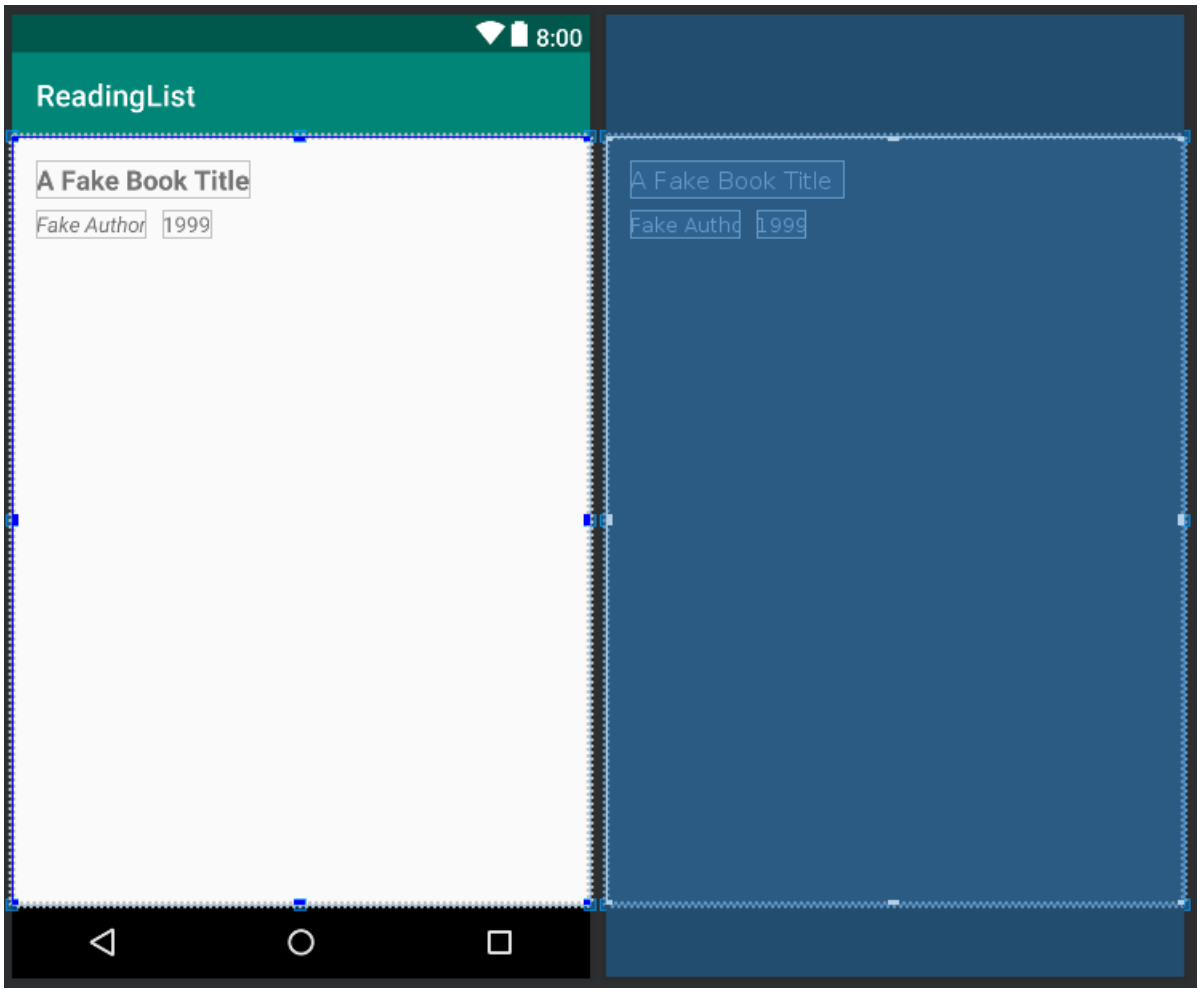


Let's change the ID from `textView` to `title`. Let's also temporarily change the text from `TextView` to `A Fake Book Title` (or it will auto-collapse and be hard to adjust). Where you see textSize, change that from `14sp` to `18sp`. Lastly, click on the little "`B`old" symbol.

In a similar way, let's create another TextView below it. This one will have the ID `author`, text will be `Fake Author`. We'll leave the size at `14sp` but click on "`I`talic".

> Note that the designer will show guidelines to help you line things up.

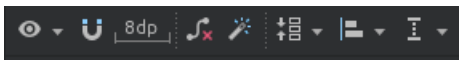
Lastly, make one more TextView next to the author. Give it an ID `year`, set the text to `1999`, leave it at `14sp` and do not click either bold or italic.



If you look at the Component Tree, you will notice a red exclamation mark on each of the new views. Hovering your mouse over these will tell you that `This view is not constrained. It only has designtime positions, so it will jump to (0,0) at runtime unless you add the constraints`

Let's do that.

Click on your `A Fake Book Title` view. In the design toolbar, you will see some icons.



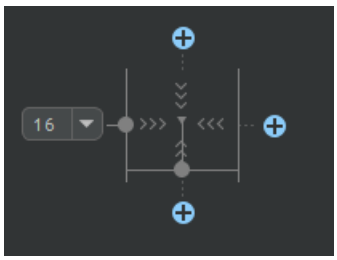
First, you will see something that looks like a magnet. It has a tooltip about "Autoconnect". If it has a line through it, then click it once to remove the line and enable auto-connect.

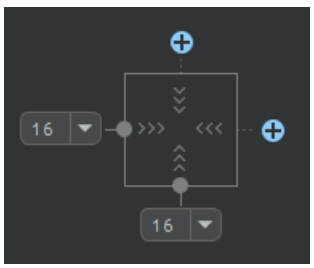
The other icon looks like a magic wand shooting sparks, with a tooltip that says "Infer Constraints". With your text view selected, click that icon.

You might have also noticed that the ConstraintLayout takes up the entire screen real estate. In reality, if this is just one row, we need that to collapse down. Click on the white background (which selects the Constraint Layout in the Component Tree) and change the layout_height in the Attributes panel to `wrap_content`.

That looks better? It would be nice if the top margin and bottom margin matched though...

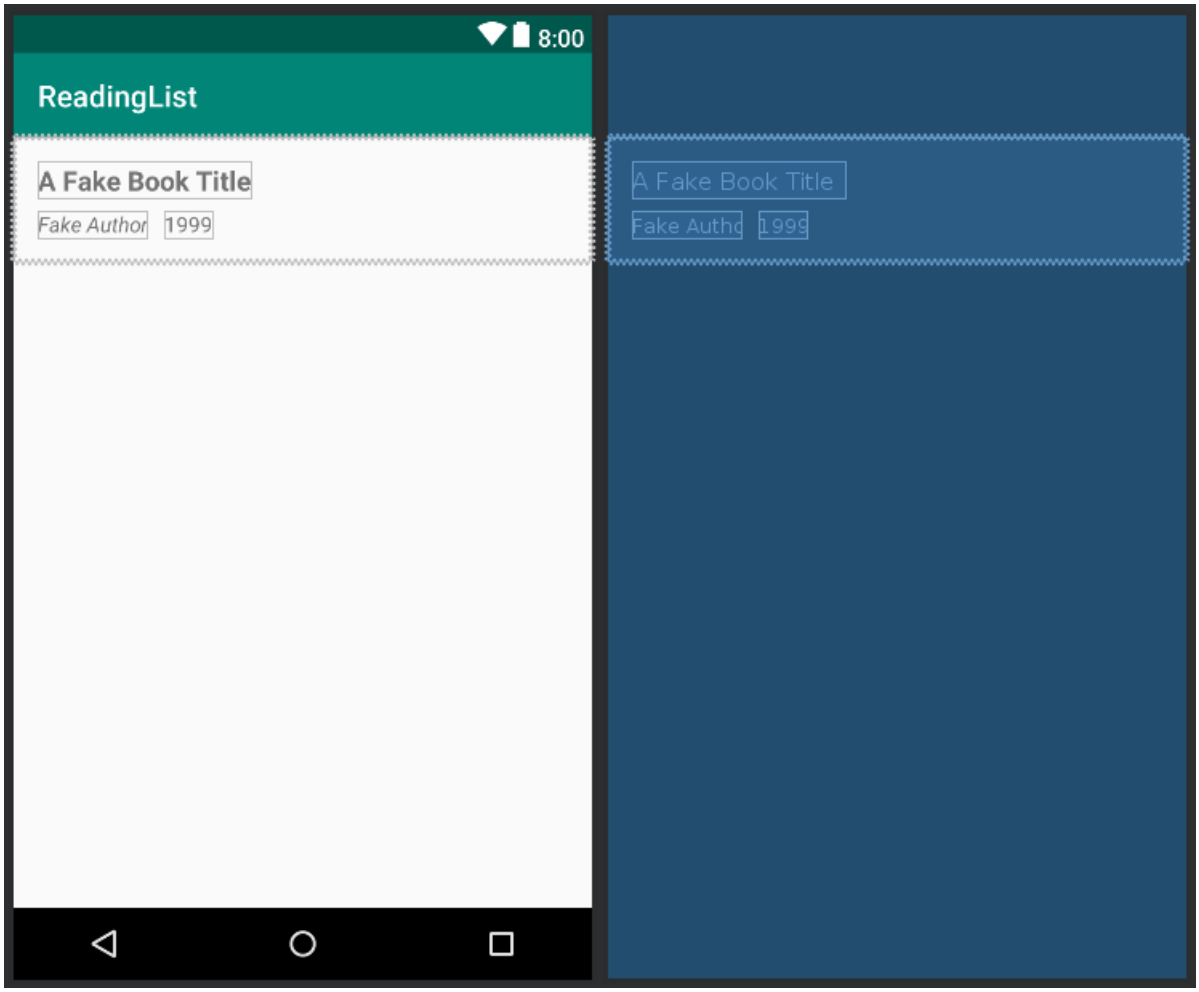
If you click on the title view, you will see in the Attributes panel that it has a `16` top margin.





The author view has a left margin, but no bottom margin. Click that `+` sign which will give you a new drop-down box. Change it

from `0` to `16`. Do the same for the year view.



In your Component Tree view, you will notice that the red exclamations have been replaced with yellow warnings. The new warning is telling us that we should be using `@string` resources, as we learned earlier.

It's time to switch back to the `Text` view and do some final editing.

You will notice that those three strings are highlighted by the IDE as needing attention. We don't ever want to show those values to the end-user. What we DO want, however, is to continue showing them in the designer.

On those three views, change the `android:text` to `tools:text`.

The highlights will go away, indicating that the problem has been dealt with. In the design view, everything looks the same, except the warnings are now gone as well.

Back in the Text view, inside the `<layout>` tag, but above our `<ConstraintLayout>` tag, add in our empty data block:

```
<data>
</data>
```

End result should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <data>

    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:layout_marginTop="16dp"
            tools:text="A Fake Book Title"
            android:textSize="18sp"
            android:textStyle="bold"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/author"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:layout_marginBottom="16dp"
            tools:text="Fake Author"
            android:textStyle="italic"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent" />

        <TextView
            android:id="@+id/year"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="11dp"
            android:layout_marginTop="8dp"
            android:layout_marginBottom="16dp"
            tools:text="1999"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toEndOf="@+id/author"
            app:layout_constraintTop_toBottomOf="@+id/title" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

View Models

Currently, if we wanted to populate this layout we would need to set the title, author and year. Wouldn't it be nice if we could just set the Book model instead?

We can.

Inside the data block, let's specify a variable. Remember, auto-completion is your friend here.

```
<data>
    <variable
        name="book"
        type="com.aboutobjects.curriculum.readinglist.model.Book" />
</data>
```

> Note that the type is fully-qualified

Now, inside the title view, add a new attribute

```
android:text="@{book.title}"
```

The `@{...}` format tells the framework to execute that instruction; thus, it gets the title from the book variable. Technically, if you are familiar with Java, it is calling `book.getTitle()`. Kotlin in our model just hides that syntax.

Do similar with the author and year.

Now, we can just set the `book` and everything else should fall into line.

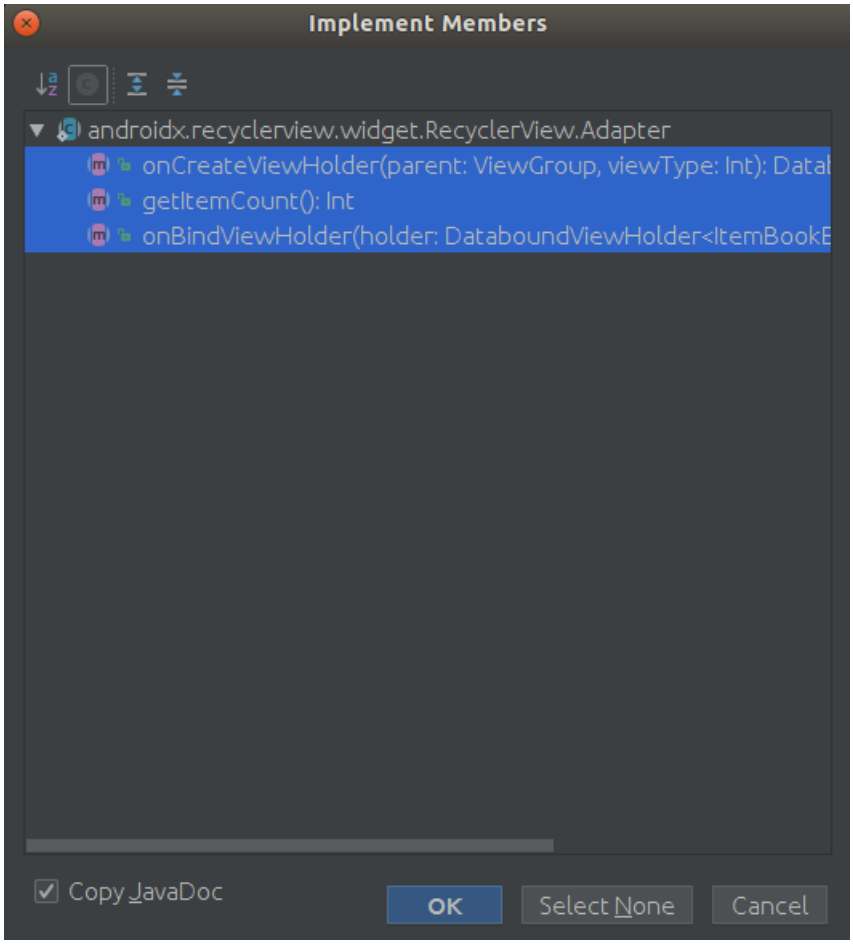
Adapter

The next step is to create our Adapter. Create a new class in our `ui` package called `ReadingListAdapter`. It has to extend `RecyclerView.Adapter<T>`, but we have our new `DataboundViewHolder` so what we end up with is:

```
package com.aboutobjects.curriculum.readinglist.ui

import androidx.recyclerview.widget.RecyclerView
import com.aboutobjects.curriculum.readinglist.databinding.ItemBookBinding

class ReadingListAdapter : RecyclerView.Adapter<DataboundViewHolder<ItemBookBinding>>() {
}
```

When you extend `RecyclerView.Adapter`, you will notice that your new `ReadingListAdapter` is underlined in red. Move your mouse over it and it will tell you that your class is not abstract and it doesn't implement certain functions.

Use the IDE to help us. Click on `ReadingListAdapter` and Alt+Enter (or Option+Enter on Mac) to use the IDE quick fix.



Select all of the method it recommends and check the `Copy JavaDoc` then click OK.

Take a moment to look at what was generated.

Before we replace the TODOs that were created, we need some data to work with. Our adapter will use our `ReadingList` model that we created earlier.

At the top of the class, let's add a new variable to hold that model.

```
var readingList: ReadingList? = null
    set(value) {
        field = value
        notifyDataSetChanged()
    }
```

Here, we set it to `null` by default since it is not loaded yet. Then, whenever the data is set, we notify listeners that the data has changed.

getItemCount

Now, looking at our three TODO, it seems that the `getItemCount` might be the easiest? We are just going to return the number of books in our list.

We can replace

```
override fun getItemCount(): Int {
    TODO("not implemented") //To change body of created functions use File | Settings | File Templat
es.
}
```

with

```
override fun getItemCount(): Int {
    return readingList?.books.orEmpty().size
}
```

And actually, we can even reduce it a *little* more...

```
override fun getItemCount(): Int = readingList?.books.orEmpty().size
```

onCreateViewHolder

What about the `onCreateViewHolder`? This method is used to create a new view holder for each row.

For this one, we want to take an extra preparatory step.

Let's add another method first. Using Ctrl+O for Override, double click on `getItemViewType`.

What we want is a unique integer that represents how we want to display the item at a specific index. What better way than to return the layout resource id?

For now, we only have one row type, so just return it.

```
    override fun getItemViewType(position: Int): Int =  R.layout.item_book
```

> Later, we will look at how to intermix different types.

Now, back to `onCreateViewHolder`. The viewType passed into that method will now be our layout id, which means we can do this:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DataboundViewHolder<ItemBookBindi
ng> {
    return DataboundViewHolder(layoutId = viewType, parent = parent)
}
```

onBindViewHolder

That leaves the `onBindViewHolder`. This one takes the view holder created above and binds our data to it.

```
override fun onBindViewHolder(holder: DataboundViewHolder<ItemBookBinding>, position: Int) {
    readingList?.let {
        holder.binding.book = it.books[position]
    } // else... what do we do?
}
```

If the reading list is loaded, we get the specified book model and set it on the binding class that was auto-generated; which will in turn set the title, author and year.

> I left a question there for us to evaluate once the new UI is deployed to the emulator.

Setting up the RecyclerView

Now that all the pieces are in place, it's time for us to actually use them.

Open your `activity_book_list.xml` and replace the ImageView with the new RecyclerView:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/black"
        tools:context=".BookListActivity">

        <TextView
            android:id="@+id/hello_text"
            style="@style/LightTitle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/hello"
            app:layout_constraintBottom_toTopOf="@+id/login_text"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

        <TextView
            android:id="@+id/login_text"
            style="@style/LightInfo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            tools:text="@string/last_login"
            app:layout_constraintBottom_toTopOf="@+id/recycler"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/hello_text"/>

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/recycler"
            android:background="@color/cornsilk"
            android:scrollbars="vertical"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/login_text"/>

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

We borrow the constraints from the old ImageView, and make sure to adjust the constraint on the login text to point to the new

recycler.

> layout_height of `0dp` tells it to use the remaining space.

Try to deploy. Doesn't matter if it succeeds or not - you are doing this step to get the compiler to regenerate the binding class.

Open your `BookListActivity`.

First, add two more variables

```
private val viewAdapter = ReadingListAdapter()
private lateinit var viewManager: RecyclerView.LayoutManager
```

We are able to define the adapter already because we chose to allowing setting the ReadingList in it after the fact.

Now, in our `onCreate`, immediately after we define `binding` let's define the `viewManager` and setup our `recyclerView`.

```
viewManager = LinearLayoutManager(this)
binding.recycler.apply {
    setHasFixedSize(true)
    layoutManager = viewManager
    adapter = viewAdapter
}
```

The other common layout manager is the GridLayoutManager; but LinearLayoutManager is by far the most commonly used.

Lastly, inside our `loadJson()?.let {` closure, call

```
viewAdapter.readingList = it
```

That will update our adapter (and notify any listeners) once the data is loaded.

Debugging Databinding

Try to deploy the app.

You will notice an error that looks like this

```
Found data binding errors.
****/ data binding error ****msg:Cannot find the setter for attribute 'android:text' with parameter
type com.aboutobjects.curriculum.readinglist.model.Author on android.widget.TextView. file:/home/mal
achid/work/curriculum/ReadingList/app/src/main/res/layout/item_book.xml loc:34:28 - 34:38 ****\ data
 binding error ****
```

So what does this mean?

It says that you are calling `android:text` on a `TextView`, but passing an `Author` model. An `Author` model is not a String, so it doesn't know what to do.

There are a couple options on how to address this. If the object you are referencing is a simple one, you might get away to `toString()`, but in our case that wouldn't look very good.

You can also use BindingAdapters so that the system knows how to use `Author` models in `TextView` classes. While this can seem really tempting, it is best to leave those to things that are used often (like converting colors or urls, etc). Otherwise it starts to grow out of hand very quickly.

Another option for us is to write a function that handles it specifically. These are often contained in the view model classes, or within extension functions.

We'll take a simpler approach.

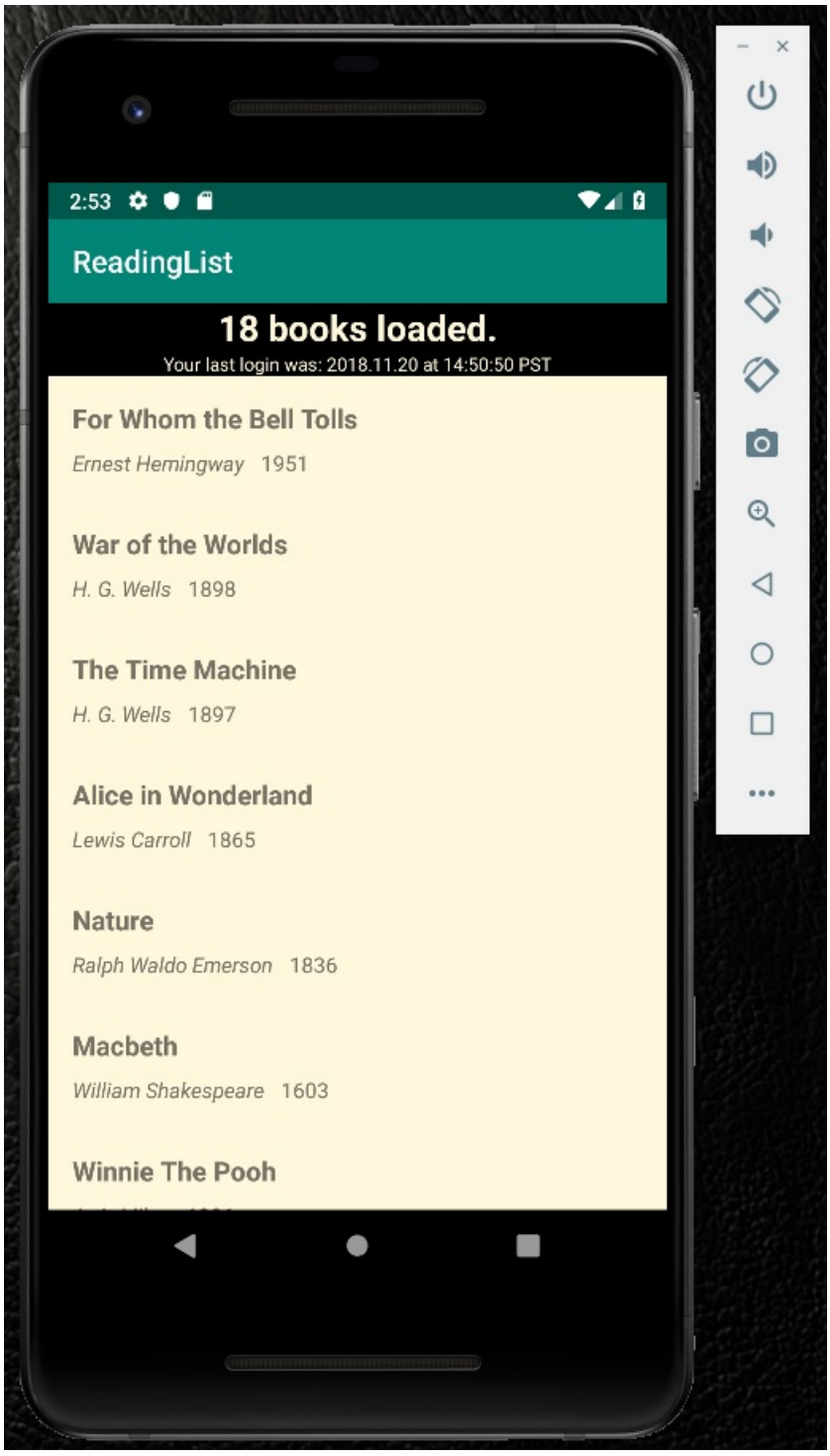
Let's update our `Author` model to have a `displayName` function.

```
package com.aboutobjects.curriculum.readinglist.model

data class Author(
    val firstName: String? = null,
    val lastName: String? = null
) {
    companion object {
        const val UNKNOWN = "Unknown"
    }

    fun displayName(): String? {
        return when {
            firstName == null && lastName == null -> UNKNOWN
            firstName == null -> lastName
            lastName == null -> firstName
            else -> "$firstName $lastName"
        }
    }
}
```

Then in our `item_book.xml` and change the author binding to

```
android:text="@{book.author.displayName()}"
```

Deploy the application.



Divider

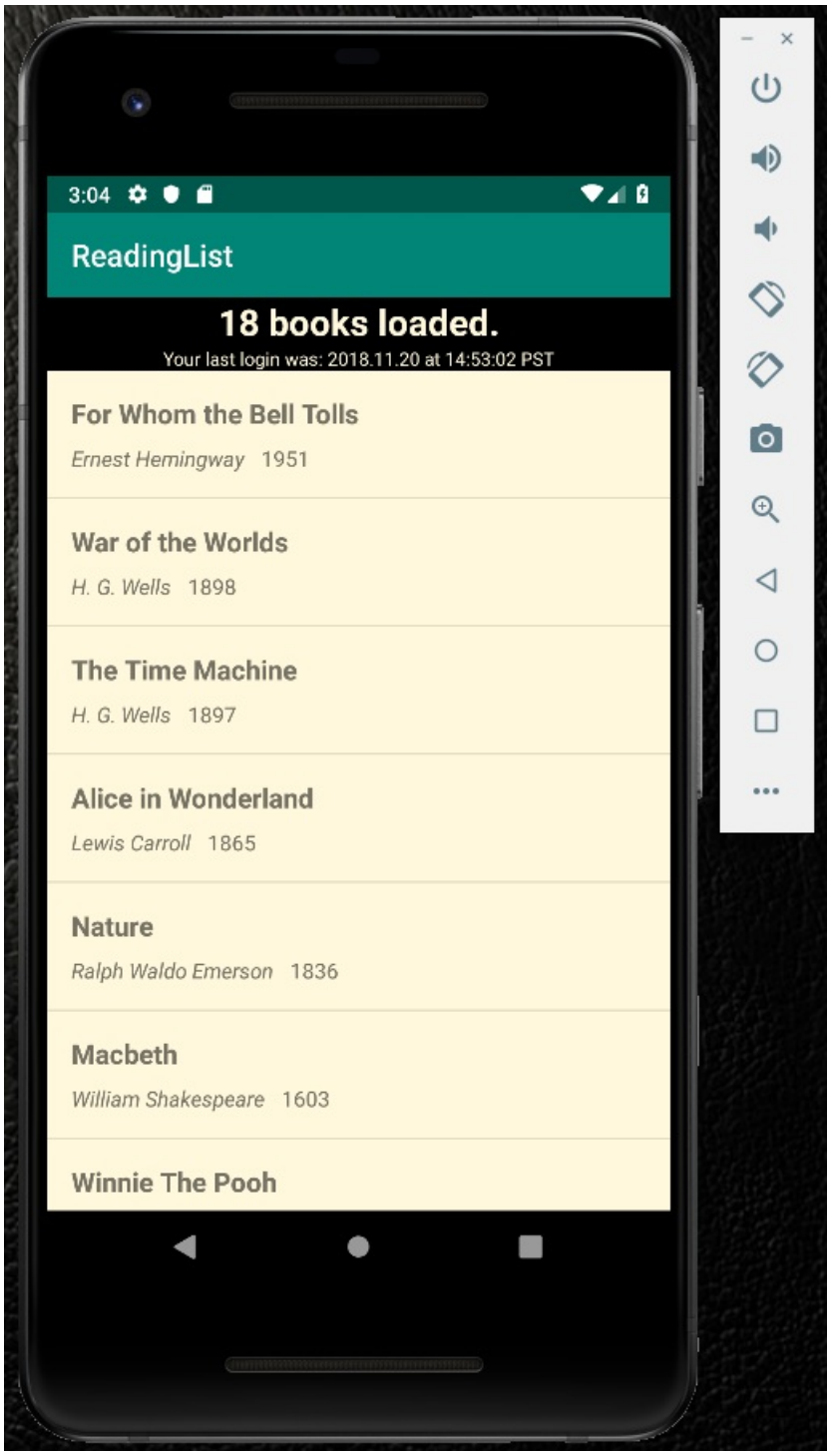
That's starting to look a lot better. Can we add a divider in between the rows?

Edit the `recycler` closure in our `BookListActivity` and add a DividerItemDecoration

```
binding.recycler.apply {
    setHasFixedSize(true)
    layoutManager = viewManager
    addItemDecoration(DividerItemDecoration(this@BookListActivity, DividerItemDecoration.VERTICAL))
    adapter = viewAdapter
}
```

We specify `this@BookListActivity` because `this` refers to the `binding.recycler` itself.

Redeploy your application.



Multiple View Types

It's starting to look pretty good. Our `ReadingList` model also has its' own title. Maybe we can add that to our list?

To do that, we are going to update our `ReadingListAdapter` to handle more than one type of row.

Like before, let's start with defining the layout for the new row. Let's call it `item_title.xml` with a `layout` Root element.

Add a TextView and we'll use our `ReadingList` model as the variable.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <variable
            name="readinglist"
            type="com.aboutobjects.curriculum.readinglist.model.ReadingList" />
    </data>

    <TextView
        android:id="@+id/title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@{readinglist.title}"
        tools:text="ReadingList Title"
        android:textSize="24sp"
        android:textStyle="bold" />
</layout>
```

Next we will make some tweaks to our `ReadingListAdapter`.

First, we will now be referencing both `ItemBookBinding` and `ItemTitleBinding`, so change our class signature to be more generic:

```
class ReadingListAdapter : RecyclerView.Adapter<DataboundViewHolder<ViewDataBinding>>() {
```

Now, starting with `getItemCount`, we know that we will have 1 extra element for the title.

```
override fun getItemCount(): Int {
    return readingList?.let {
        it.books.size + 1 // +1 for title
    } ?: 0 // no title if not loaded
}
```

Next we look at `getItemViewType`. We know that the first (item 0) will be the title and that any other item will be a book.

```
override fun getItemViewType(position: Int): Int {
    return when(position) {
        0 -> R.layout.item_title
        else -> R.layout.item_book
    }
}
```

Since we are using the `getItemViewType` to specify the layout id, the `onCreateViewHolder` doesn't have to change anything but the signature.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DataboundViewHolder<ViewDataBindi
ng> {
    return DataboundViewHolder(layoutId = viewType, parent = parent)
}
```

The `onBindViewHolder` has to change both the signature and the implementation.

```
override fun onBindViewHolder(holder: DataboundViewHolder<ViewDataBinding>, position: Int) {
    readingList?.let {
        when(holder.binding) {
            is ItemTitleBinding -> holder.binding.readinglist = it
            is ItemBookBinding -> holder.binding.book = it.books[position - 1]
            else -> {
                // @TODO add logging
            }
        }
    } // else... what do we do?
}
```

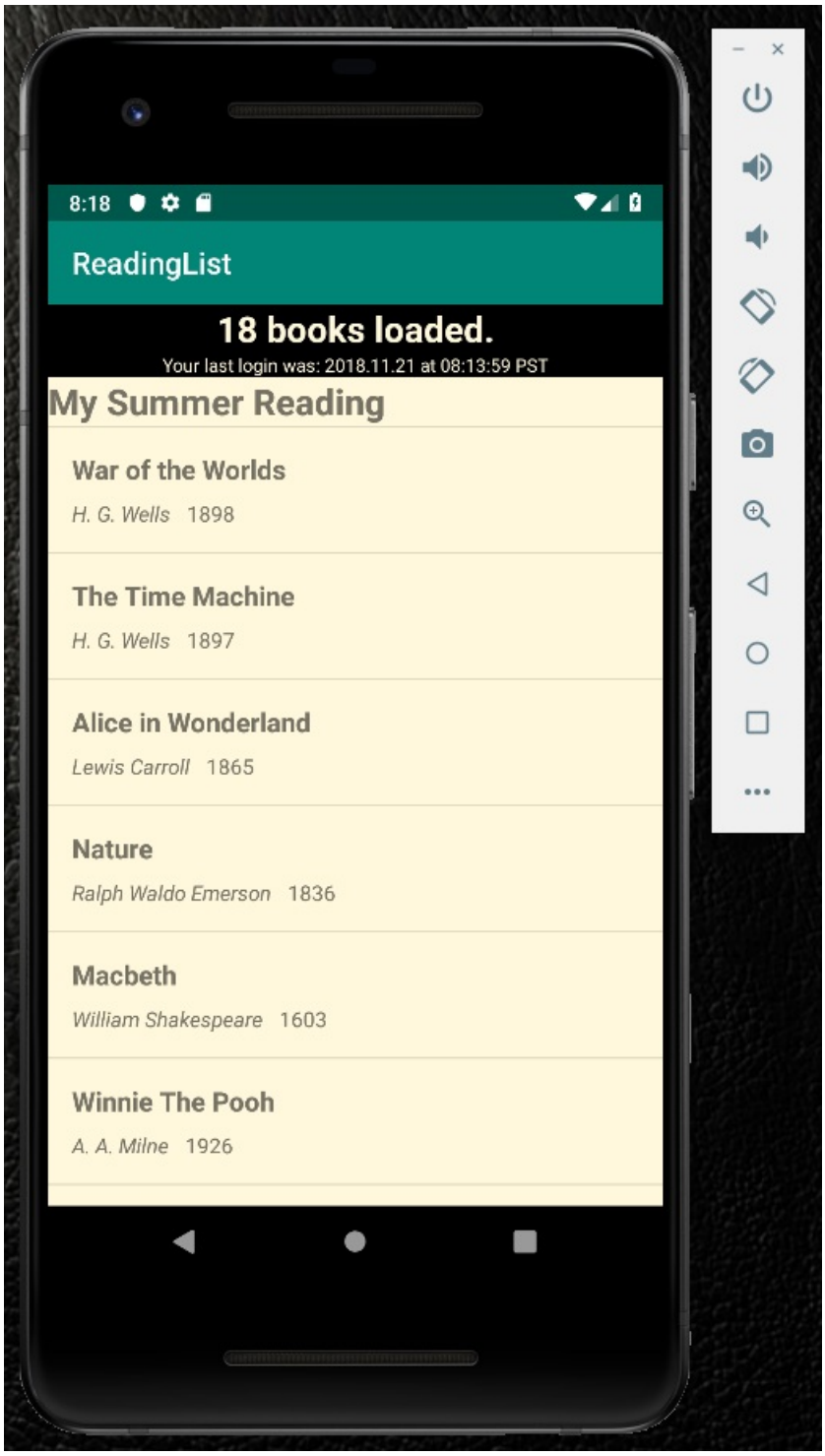
Here, we are relying on our `DataboundViewHolder` having knowledge of what type it is. We *could* use the position index like in `getItemViewType`, but why duplicate logic?

Also, the IDE is telling you that `is ItemBookBinding` is always true. So why not just make that row the `else` instead of having the TODO section? Down the road, if you were to add additional types (say, Magazines) this approach would help catch a bug that would cause it to be shown with the wrong layout.

Note that we changed the array position from `position` to `position - 1` to account for position 0 now being the title.

We'll get to how to add logging in the next chapter.

Deploy your application.



Cleanup

It's starting to look a lot better... let's remove our early work from the top of the page.

activity_book_list.xml

Update your `activity_book_list.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/cornsilk"
        tools:context=".BookListActivity">

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/recycler"
            android:scrollbars="vertical"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Try (unsuccessfully) to deploy your application so that it re-generates the `ActivityBookListBinding`.

BookListActivity.kt

Open the `BookListActivity`. You'll notice that the `helloText` and `loginText` are now red. That's because these fields are no longer available.

Remove:

```
binding.helloText.text = getBooksLoadedMessage(it.books.size)
```

Since we are no longer showing the timestamp, you can also remove:

```
prefs.getString(KEY_TIMESTAMP, null)?.let {
    val time = timestampFormat.parse(it)
    binding.loginText.text = resources.getString(R.string.last_login, displayFormat.format(time))
}

prefs.edit {
    putString(KEY_TIMESTAMP, timestamp())
}
```

Now, scrolling up to the top of the page, you will notice some variables (like `timestamp`) are grayed out. That's because they are no longer in use. Let's remove some more. Remove these.

```
private val displayFormat: SimpleDateFormat by lazy {
    SimpleDateFormat(displayPattern, Locale.getDefault())
}

private fun timestamp(): String {
    return timestampFormat.format(Calendar.getInstance().time)
}

private val prefs: SharedPreferences by lazy {
    getSharedPreferences(PREF_FILE, Context.MODE_PRIVATE)
}
```

More is grayed out, remove this:

```
private val timestampFormat: SimpleDateFormat by lazy {
    // Don't set Locale.getDefault() in companion because user may change it at runtime
    SimpleDateFormat(timestampPattern, Locale.getDefault())
}
```

If you look at the `companion object` some of those variables are also grayed out. Remove these.

```
// from https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
const val timestampPattern = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
const val displayPattern = "yyyy.MM.dd 'at' HH:mm:ss z"
val KEY_TIMESTAMP = "${BookListActivity::class.java.name}::timestamp"
```

We're almost done with this file.

Your imports are probably collapsed to look like `import ...`. Click on those dots to expand that section. You'll notice quite a few of those imports are also no longer required. We'll take a shortcut to clean those up. Use `Ctrl+Alt+O` (or `Ctrl+Option+O` on Mac) to Optimize Imports.

Let's do one last piece of cleanup while we are here.

You'll notice that we have `loadJson` as its' own method, but the save json logic is inline. Let' move that out.

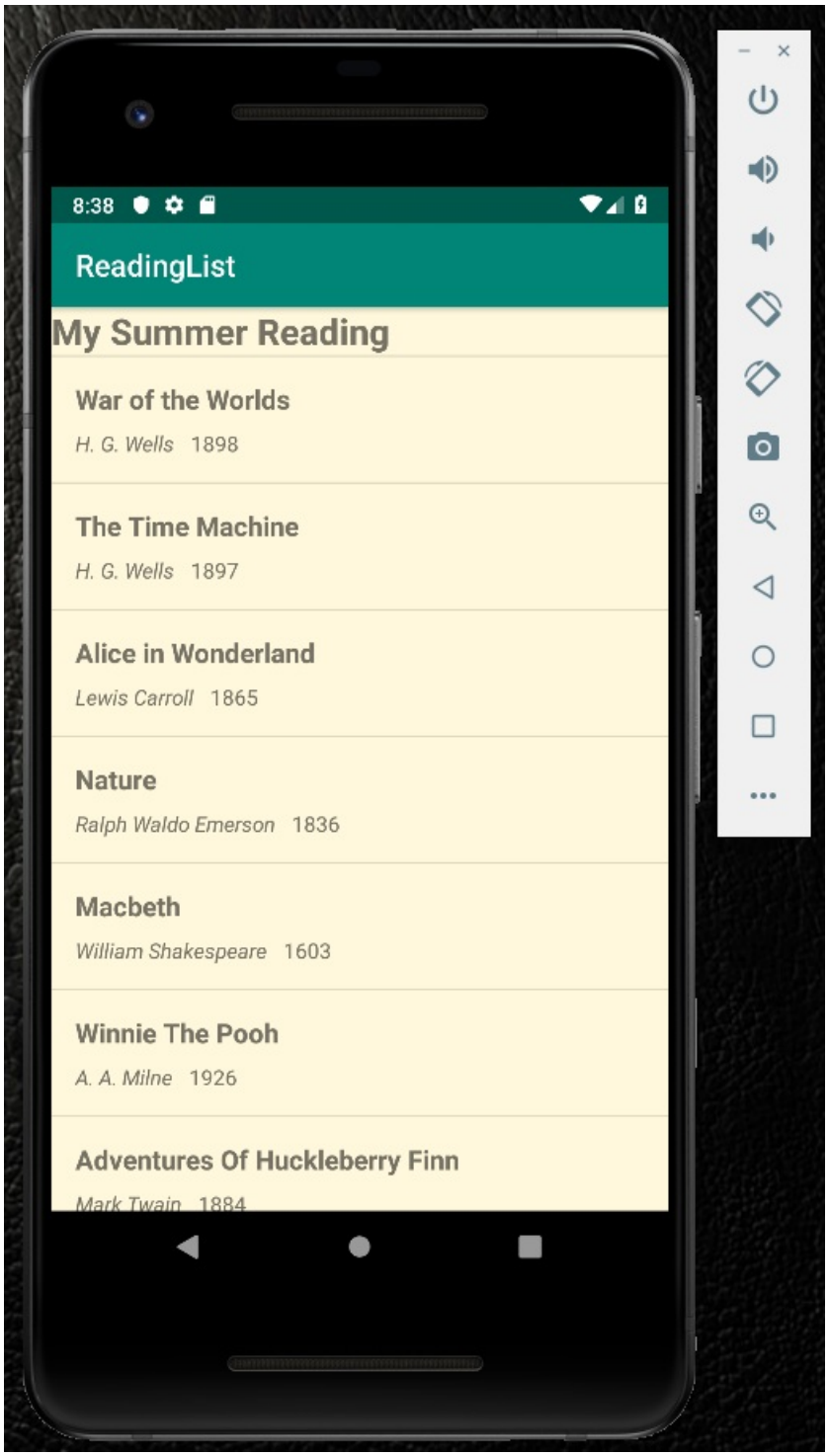
First, we will copy the logic from `onCreate` into a new method

```
private fun saveJson(readingList: ReadingList) {
    try{
        val writer = FileWriter(File(filesDir, JSON_FILE))
        app.gson.toJson(readingList, writer)
        writer.flush()
        writer.close()
    }catch(e: Exception){
        // @TODO introduce logging
    }
}
```

Then update our `onCreate`

```
loadJson()?.let {
    viewAdapter.readingList = it
    saveJson(it)
}
```

We can deploy our application successfully now, and it's looking a lot better. But we are missing something. Do you know what it is?



Unit Tests

Right-click on your `com (test)` folder and `Run 'Tests in 'com''`.

Good news, they all pass.

You should make a habit of running these tests regularly.

Integration and UI Tests

Right-click on your `com (androidTest)` folder and `Run 'Tests in 'com''`.

```
/home/malachid/work/curriculum/ReadingList/app/src/androidTest/java/com/aboutobjects/curriculum/readinglist/BookListActivityUITests.kt: (31, 28): Unresolved reference: login_text
```

Our UI tests made assumptions that `login_text` existed - but it no longer does.

Open `BookListActivityUITests`. You'll notice that, like in our `BookListActivity`, `login_text` is now red because it no

longer exists.

Although not a very useful test, let's replace `lastLogin_isDisplayed` with one that will pass for now

```
@Test
fun recycler_isDisplayed() {
    val scenario = ActivityScenario.launch(BookListActivity::class.java)

    onView(withId(R.id.recycler))
        .check(matches(isDisplayed()))
}
```

Run your tests and verify everything passes now.

Click Handling

Now that we can see a list of books, it would be nice if we could click on one. Maybe to edit it?

While it is possible to use a generic click handler on UI elements; with databinding we can make it a bit more manageable.

Let's start by defining a new interface to be used when a `Book` model is clicked.

BookClickListener

Right-click on your `ui` package and select `New | Kotlin File/Class` and name it `BookClickListener`. We then define the callback method we want to be used.

```
package com.aboutobjects.curriculum.readinglist.ui

import com.aboutobjects.curriculum.readinglist.model.Book

class BookClickListener(val bookClicked: (Book) -> Unit) {
    fun onBookClicked(book: Book) {
        bookClicked.invoke(book)
    }
}
```

The weird constructor here allows us to pass a Kotlin lambda function. Similar redirects can be used to wrap generics, for example, to simplify the definitions in the XML.

item_book.xml

We'll update our xml to use the callback.

Add a new variable in the `data` block

```
<variable
    name="listener"
    type="com.aboutobjects.curriculum.readinglist.ui.BookClickListener" />
```

Then we will update our layout to add the `focusable`, `clickable` and finally `onClick`.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:focusable="true"
    android:clickable="true"
    android:onClick="@{() -> listener.onBookClicked(book)}">
```

> Note, if we only wanted access to the `view` and not the `book`, there is a much simpler way of doing it using method references, but the `Book` model is what really matters to us.

Redeploy your application to make sure the binding classes are re-generated.

ReadingListAdapter.kt

Our adapter now needs to be updated to populate the new `listener` variable.

First, we'll update our class definition to take a lambda in the constructor.

```
class ReadingListAdapter(
    val bookClicked: (Book) -> Unit
) : RecyclerView.Adapter<DataboundViewHolder<ViewDataBinding>>() {
```

And then update the `onBindViewHolder` for the book case (we don't currently allow clicking on the title)

```
is ItemBookBinding -> {
    holder.binding.book = it.books[position - 1]
    holder.binding.listener = BookClickListener(bookClicked)
}
```

BookListActivity

Now, the our `BookListActivity` will actually do something when the book is clicked. You are probably wondering why all the levels of indirection? Each piece of the puzzle has their own job to do, and we don't want any one class becoming either a kitchen sink or a bottleneck. Anotherwords, only ask those classes to do what they were meant to do. The `ReadingListAdapter` is only meant to select which layout to show.

If you look closely at our line that says `private val viewAdapter = ReadingListAdapter()` you will notice that there is now a red mark in the parens. Hovering over it will tell you that `No value passed for parameter 'bookClicked'`.

Let's do that now.

```
private val viewAdapter = ReadingListAdapter(
    bookClicked = {
        // @TODO do something with `it`
    }
)
```

We need to create a new screen to launch when the book is clicked.

> For now we will have the click launch a new Activity. When we get to the Fragments chapter, we will update this code to use Fragments instead.

EditBookActivity

Make a new class in the same package as `BookListActivity` called `EditBookActivity`. Like our `BookListActivity`, it will also extend `AppCompatActivity`.

```
package com.aboutobjects.curriculum.readinglist

import androidx.appcompat.app.AppCompatActivity

class EditBookActivity: AppCompatActivity() {
}
```

You'll notice that EditBookActivity is highlighted. Hovering over it tells you that it is not in the manifest. Click on `EditBookActivity` once. You'll see a lightbulb appear to the left of the editor. Click it then choose `Add activity to manifest`.

> Note: You could have used the Quick Fix shortcut (Alt+Enter or Option+Enter on Mac) instead of clicking the lightbulb.

If you look at your `AndroidManifest.xml` you will notice that it added

```
<activity android:name=".EditBookActivity" />
```

It did not, however, add the MAIN LAUNCHER intent filter, which is why it doesn't look like an additional application in your app

drawer.

activity_edit_book.xml

We'll need a new UI to edit the book. We have a couple options on how to do the new layout. We could mimic what we have just done and create a new item type for each piece of information (title, author, year) that we want to edit.

Let's use the Designer for quick prototyping.

Start by creating another layout file `activity_edit_book.xml` with a `layout` root element (because we are using databinding).

We know we want to be able to edit 3 fields. We probably also want labels for those 3 fields. So, we probably want 3 rows with 2 columns each.

Let's try adding a `Layouts | TableLayout`. Now, although there are TableRows automatically added, they may not show up in the designer. If that's the case, go ahead and drop a `Layouts | TableRow` onto the `TableLayout` in the Component Tree. That should force them to expand.

Drag a `Text | TextView` onto the first `TableRow` in the Component Tree. Do the same thing to the second and third `TableRow`.

Click on the 4th `TableRow` and click your delete button. Do the same for the 5th one.

Now, drag a `Text | Plain Text` onto the `textView` in the first `TableRow`. Do the same thing to the other two.

Now we are going to go down the list and start styling.

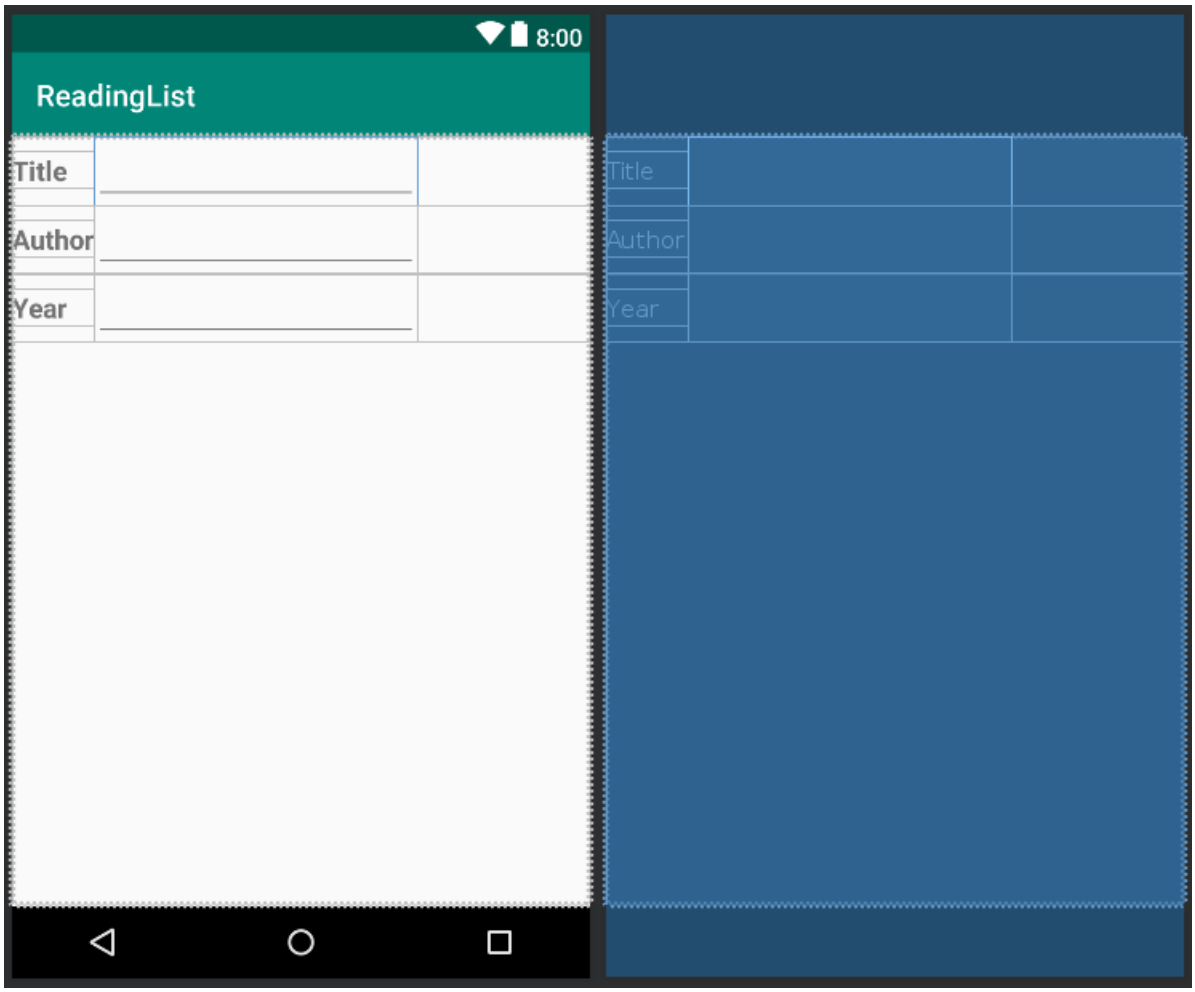
Click the `textView` in the first row, give it an id of `title_label`, set the text to `Title`, make it `18sp` and `B`old.

The second one will be `author_label` and `Author`. The third `year_label` and `Year`.

Now, the `editText` entries.

The first one will be `title`. Change the `inputType` to text (make sure to unselect any other options). Remove the value in the `text` field.

Do the same for the `author` and `year` fields, except make the `year` field a `number` instead of `text`.

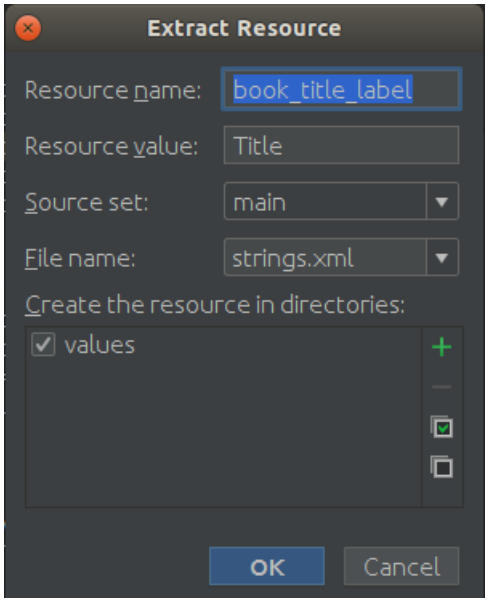


Back in the Text view, add our empty data block

```
<data>

</data>
```

You'll notice our warning about the three strings. Let's fix that with the Quick Fix (either the keyboard shortcut or the lightbulb) by clicking in the string itself (ie inside the word `Title`, not just anywhere on the line) and choosing `Extract string resource`.

We'll give them reasonable resource names that won't conflict down the line, like `book_title_label`.



When you have fixed all three strings, rebuild to regenerate the binding classes.

Back to EditBookActivity

Update the `EditBookActivity` to show the new UI.

```
private lateinit var binding: ActivityEditBookBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = DataBindingUtil.setContentView(this, R.layout.activity_edit_book)
}
```

> Remember, if you are copy/pasting the binding class may not auto-complete. If that is the case, just type that class name.

Now, how are we going to tell it *which* book to display information about? We need some way to pass the `Book` model from our `BookListActivity` to our `EditBookActivity`.

We will add a function to our `companion object` that can be called statically (before we create the EditBookActivity instance).

```
companion object {
    val EXTRA_BOOK = "${EditBookActivity::class.java}.name::book"

    fun getIntent(context: Context, book: Book): Intent {
        val gson = (context.applicationContext as ReadingListApp).gson
        return Intent(context, EditBookActivity::class.java).apply {
            putExtra(EXTRA_BOOK, gson.toJson(book))
        }
    }
}
```

Here, we ask the caller to provide their context (because we do not have one yet) and the `Book` they would like to edit. Since the Intent doesn't know how to handle that type of model, we are converting it to something it can handle, in this case a json string.

> You could also convert it to a Parcelable or a Serializable, if that would be more convenient for you -- but since we have already examined how to do this conversion, we'll use it.

We need to update our `BookListActivity` to call it

```
private val viewAdapter = ReadingListAdapter(
    bookClicked = {
        startActivity(EditBookActivity.getIntent(
            context = this,
            book = it
        ))
    }
)
```

> Note that in the IDE, the `bookClicked = {` line also includes a hint that says `it: Book`.

That will suffice to call the new edit activity. But we still need to consume it.

Back in the `EditBookActivity` we add a new variable to read the new parameter from the intent.

```
private val extraBook: String? by lazy { intent?.getStringExtra(EXTRA_BOOK) }
```

That, of course, is still reading it as a String.

Add our convenience variable from before

```
private val app: ReadingListApp by lazy { application as ReadingListApp }
```

Then in `onCreate` we can do

```
extraBook?.let {
    val book = app.gson.fromJson(it, Book::class.java)
    // @TODO use the book
}
```

We could just bind that book to our xml like before, but then it is just readable. We want it to be editable. How are we going to do that?

Two-Way Databinding with EditText

What we have done so far is one-way databinding. We let the XML view the data in our model.

What we want to do now is also have the model update when the data in the UI changes. We call that two-way databinding.

EditableBook

In our edit UI, we have three fields (title, author, year) that are editable. Let's create a view model to handle this.

In your `model` package, create a new class called `EditableBook`. We'll seed it with the `Book` we want to edit.

```
package com.aboutobjects.curriculum.readinglist.model

class EditableBook(val source: Book) {
}
```

Now, we need to mimic the three pieces of data that we want to have edited. This will be especially interesting for the `Author` field because we have a single text input for two `Book.Author` fields.

To do this, we are going to use `ObservableField`. The Android databinding library has a few handy classes to use when you want to allow the XML to observe data changes. By far, the most useful one is `ObservableField` which allows us to specify a generic, like `String` as the type.

The ObservableField variables themselves will never change - the contents *inside* those variables will, so we will use `val` instead of `var`.

We will also initialize them with values from our source Book.

```
package com.aboutobjects.curriculum.readinglist.model

import androidx.databinding.ObservableField

class EditableBook(val source: Book) {
    val title = ObservableField<String>(source.title)
    val author = ObservableField<String>(source.author?.displayName())
    val year = ObservableField<String>(source.year)
}
```

activity_edit_book.xml

Add a `EditableBook` variable to the `activity_edit_book.xml` called `book`.

```
<data>
    <variable
        name="book"
        type="com.aboutobjects.curriculum.readinglist.model.EditableBook" />
</data>
```

Like before, we will update the `android:text` value for each of the `EditText` classes; however, this time we will add an equals sign to signify two-way databinding.

```
android:text="@={book.title}"
...
android:text="@={book.author}"
...
android:text="@={book.year}"
```

Rebuild to regenerate the binding classes.

EditBookActivity

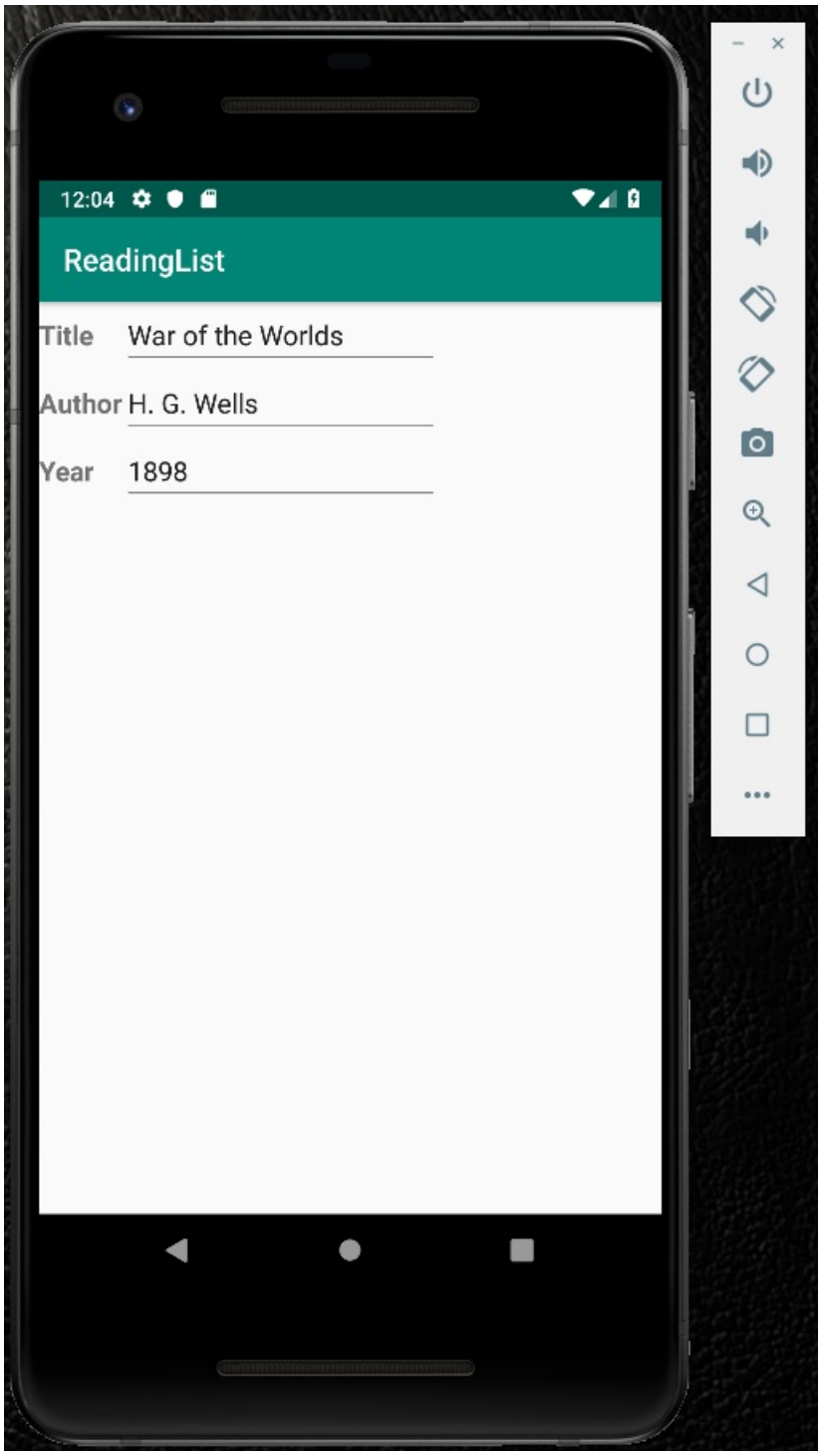
`EditBookActivity` can now be updated to bind the new `EditableBook` model.

Let's update our `onCreate`

```
extraBook?.let {
    binding.book = EditableBook(
        source = app.gson.fromJson(it, Book::class.java)
    )
}
```

Rebuild and deploy your application.

Click on one of the books.



Saving Changes

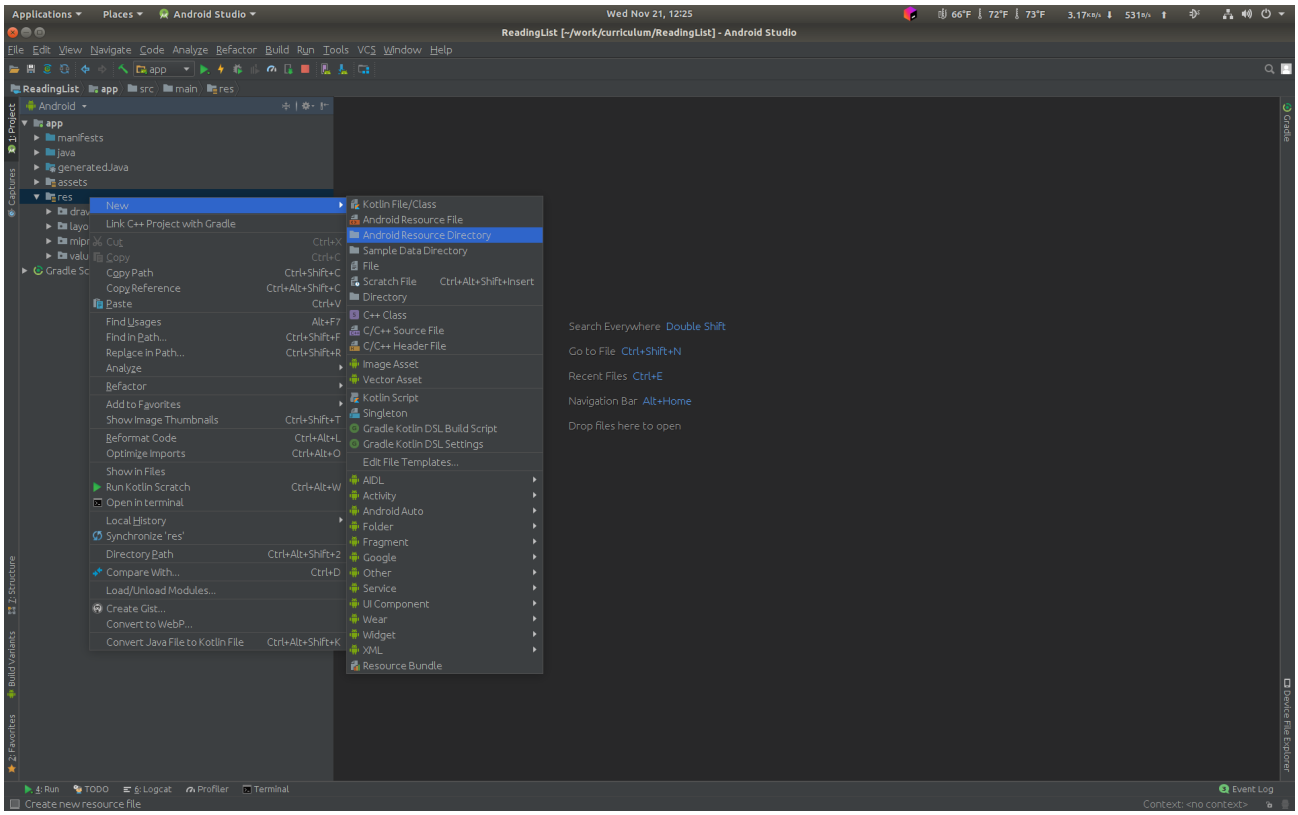
It now populates and lets you make changes; but those changes are lost when you leave the screen.

We could automatically persist any changes you make as soon as you make them, but then how would you change your mind or cancel?

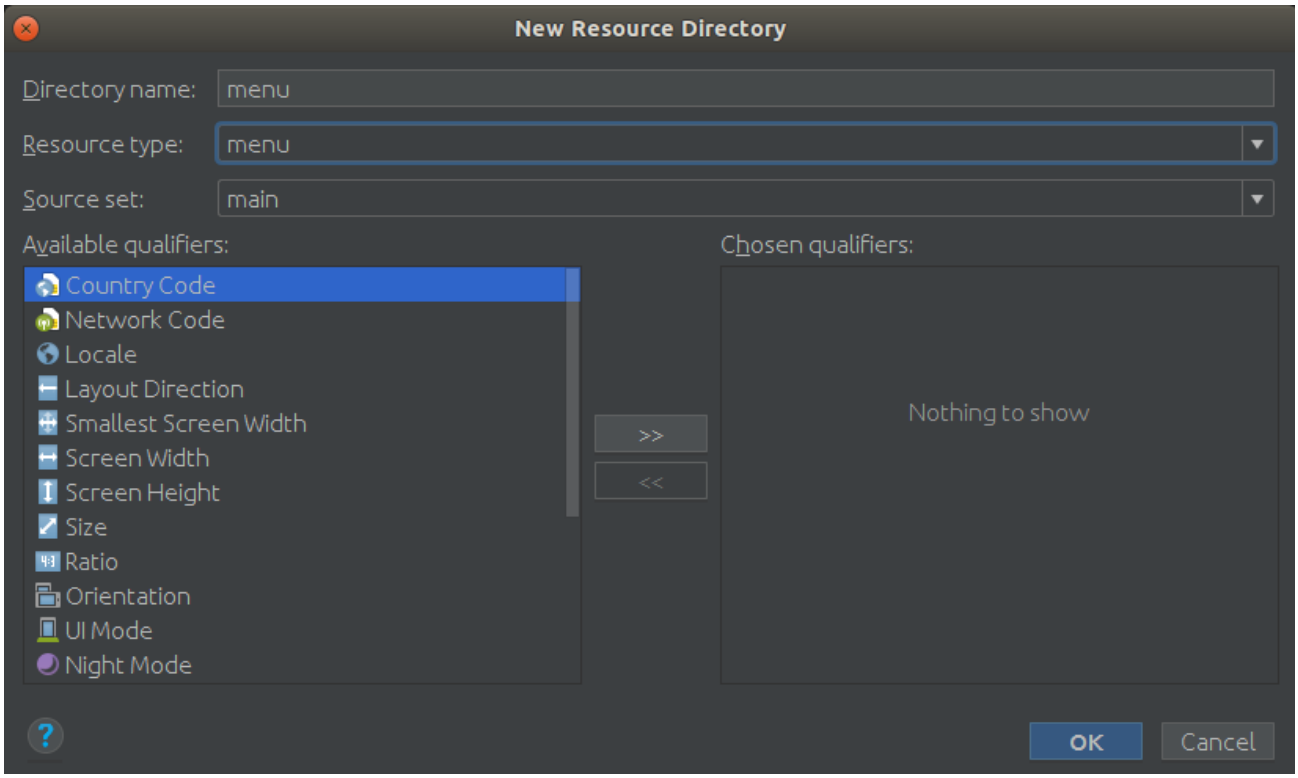
A more intuitive approach might be to have a save button, and continue to allow leaving without saving to result in cancellation.

To add a save button we have a few choices. A common approach might be to add a big button at the bottom of the page. A less disruptive approach might be to add something to the toolbar. Let's go with that approach.

menu_edit_book.xml



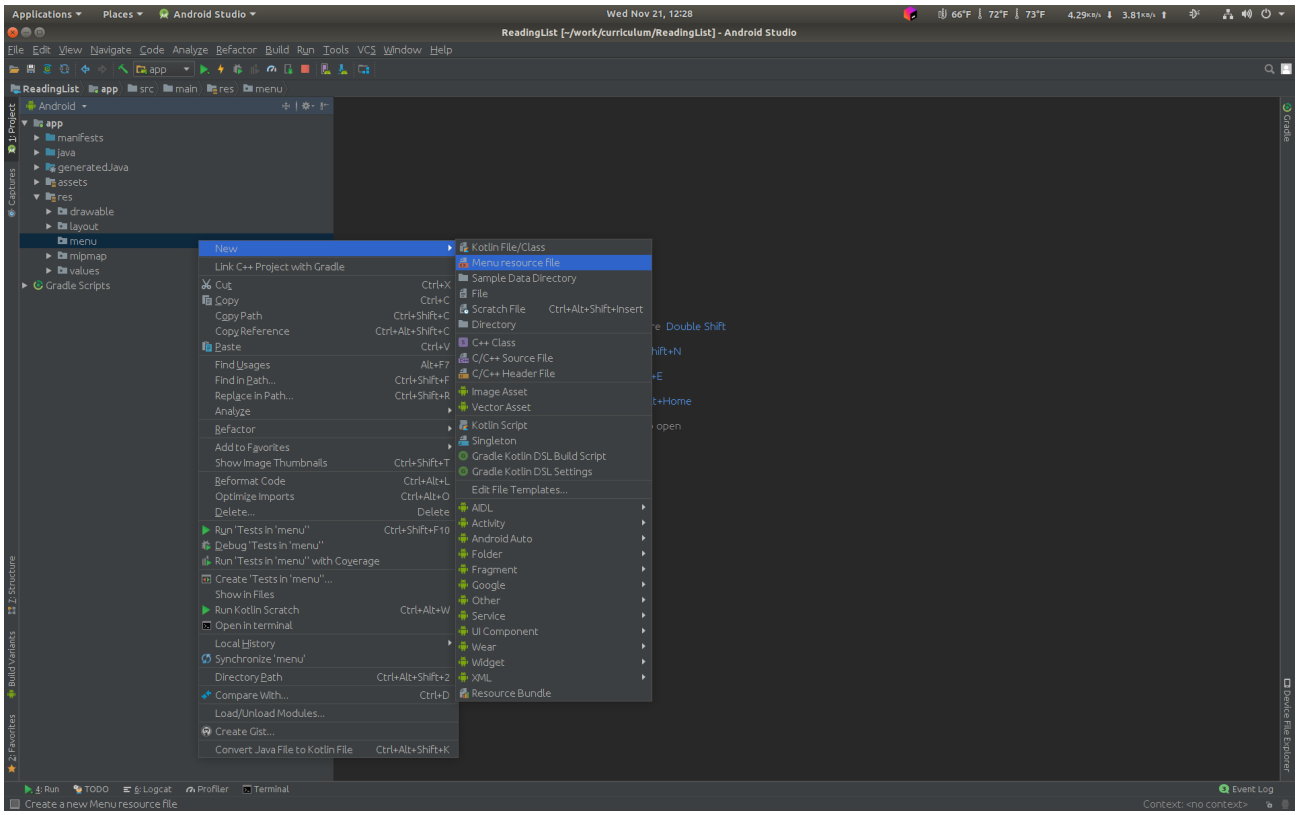
Right-click on your `res` folder and select `New | Android Resource Directory`.



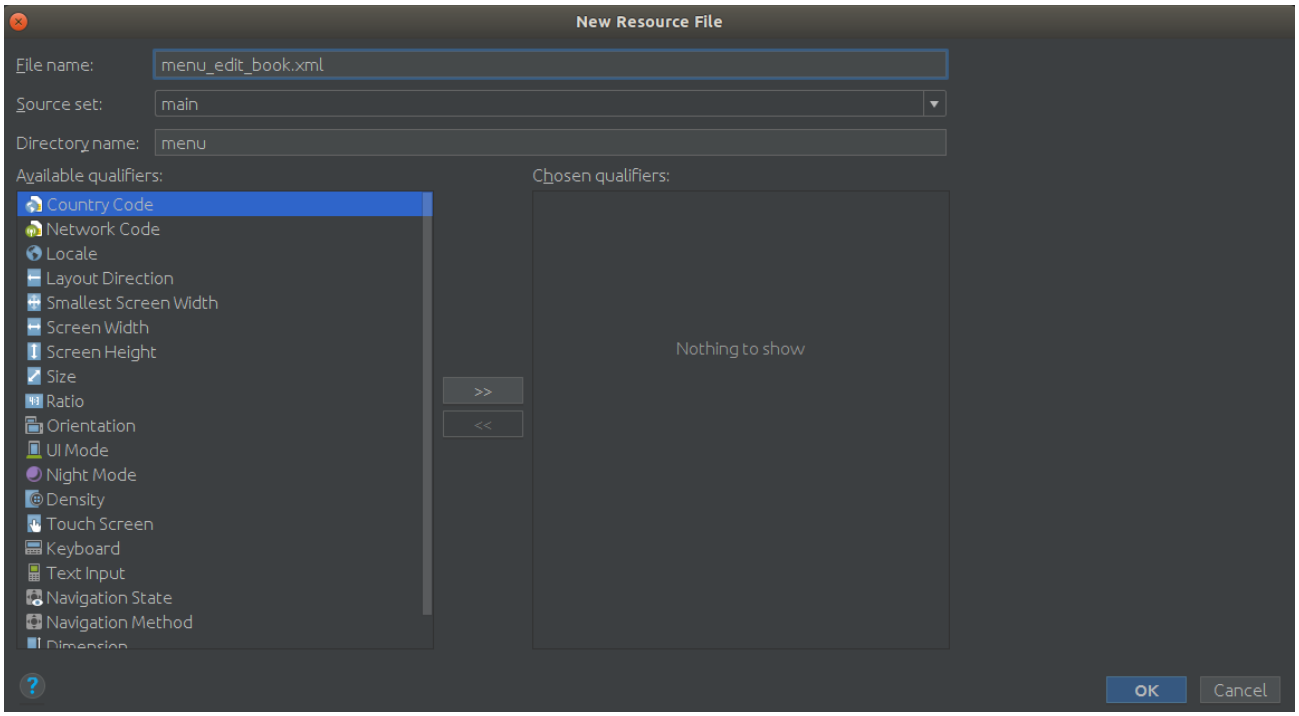
On the next screen, change the Resource type from `values` to `menu`.

> Note: We could specify qualifiers here as well.

Click `OK`.



Right-click on the new `menu` folder and select `New | Menu resource file`.



Name it `menu_edit_book.xml` and click `OK`.

Add the following `<item/>` to your new menu.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_save"
        android:icon="@android:drawable/ic_menu_save"
        android:title="Save"
        app:showAsAction="always" />
</menu>
```

The icon is built into Android. If you don't have your own, a good starting place is the ones that start with `ic_`.

Let the IDE help you convert the hardcoded `Save` text into a string resource named `save`, resulting in:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_save"
        android:icon="@android:drawable/ic_menu_save"
        android:title="@string/save"
        app:showAsAction="always" />
</menu>
```

EditBookActivity

To make your new menu show up, edit your `EditBookActivity`.

Use the Override command (`Code | Override Methods` in the toolbar, or Ctrl+O) to implement `onCreateOptionsMenu`.

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.menu_edit_book, menu)
    return true
}
```

The `R.menu` just means the `res/menu` folder, and `menu_edit_book` is the name of our xml. The last `menu` is the parent to add it to, which was passed into the function.

Redeploy your application. When you go to the edit screen, you'll now see a save icon. It doesn't do anything yet, but it is there.

For that, we need to override another method.

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    return when(item?.itemId) {
        R.id.action_save -> {
            // @TODO do something
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Here we are referencing the `android:id` we set in the menu file. We return `true` to signify that we handled the event. If we didn't, we rely on the underlying framework to make a decision.

So... what should we do when it is clicked?

We want to take the `EditableBook` and convert it to a `Book` model, and return it back to the calling activity, which has a reference to the entire reading list.

Converting the title and year should be no problem. How are we going to convert the `Author`? We have a single string but the `Author` object takes two.

Author

What we can do is build that business logic into the `companion object` of the `Author` model itself, which can then be used to create instances of the `Author` class.

Open your `Author` class. We will add a new function inside the `companion object`

```
fun from(name: String?): Author {
    return when {
        name.isNullOrEmpty()-> Author()
        name.contains(",") -> {
            val index = name.indexOf(",")
            Author(
                firstName = name.substring(index + 1),
                lastName = name.substring(0, index)
            )
        }
        name.contains(" ") -> {
            val index = name.indexOf(" ")
            Author(
                firstName = name.substring(0, index),
                lastName = name.substring(index + 1)
            )
        }
        else -> Author(lastName = name)
    }
}
```

> Are there any missing use cases? What would happen if the comma was at the end of the `name`? This is a very good example of something you might want to write unit tests for. There is no need for integration or ui tests here -- you just want to validate the business logic.

EditableBook

Let's add one more convenience for ourselves. Open `EditableBook`. Add a new function

```
fun edited(): Book {
    return Book(
        title = title.get(),
        author = Author.from(author.get()),
        year = year.get()
    )
}
```

What we are doing here is converting the ObservableField values into a new `Book` model and returning it. The ObservableField class has a `set(value)` and a `get(): value` function which we are utilizing. We are also using our new `Author.from(name)` function that we just created.

> While we could listen for changes and automatically re-create an edited book, that could cause many unnecessary models. This function is only called once, during save, which is much more efficient than every time the user types something.

BookListActivity

How do we return a value to `BookListActivity`?

Open it up and look at our `bookClicked` method. Instead of `startActivity` we are going to want to use `startActivityForResult`. It requires that we specify a request code.

Add a request code to your `companion object`

```
const val EDIT_REQUEST_CODE = 1234
```

> The number can be somewhat arbitrary. Just be aware that it is a 16-bit number, so do not use hashCode() directly.

Now, we can update our `bookClicked` handler.

```
private val viewAdapter = ReadingListAdapter(
    bookClicked = {
        startActivityForResult(EditBookActivity.getIntent(
            context = this,
            book = it
        ), EDIT_REQUEST_CODE)
    }
)
```

Go ahead and redeploy just to make sure nothing has broken.

Back to EditBookActivity

Now we can update our `onOptionsItemSelected` to return the result.

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    return when(item?.itemId) {
        R.id.action_save -> {
            binding.book?.let {
                setResult(RESULT_OK, getIntent(
                    context = this,
                    book = it.edited()
                ))
                finish()
            }
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

If the `binding.book` (our `EditableBook`) is not null, we set the result to OK and call the function we created earlier to serialize it.

This results in the calling activity getting a callback. We should probably write that too.

And back to BookListActivity

Override the method `onActivityResult`.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    // We ignore any other request we see
    if (requestCode == EDIT_REQUEST_CODE) {
        // We ignore any canceled result
        if (resultCode == Activity.RESULT_OK) {
            // Grab the book from the data
            data?.getStringExtra(EditBookActivity.EXTRA_BOOK)?.let {
                app.gson.fromJson(it, Book::class.java)?.let { book ->
                    // Grab the old reading list since it is read-only
                    viewAdapter.readingList?.let { oldReadingList ->
                        // Make a writable copy of the books
                        val booklist = oldReadingList.books.toMutableList()
                        // and add the new book to the list -- @TODO what's wrong here?
                        booklist.add(book)
                        // Create a new ReadingList
                        val newReadingList = ReadingList(
                            title = oldReadingList.title,
                            books = booklist
                        )
                        // Save the results
                        saveJson(newReadingList)
                        // And update our view
                        viewAdapter.readingList = newReadingList
                    }
                }
            }

        }
    }
    super.onActivityResult(requestCode, resultCode, data)
}
```

While it looks overly complex, half of it is comments. We grab the `Book` that was returned to us and make a new `ReadingList` model to contain that book. We then save our new list locally and update our UI.

Look through it and see if you know what the problem is at the TODO.

> I'll give you a hint. Edit one of the books. Change the title. Hit the save button. Go back through the list and look for your updated book. Did you find it? Did you also find the original?

We added a book, but we didn't remove the original book. How can we know which book to remove?

In general, there are two ways. We either *also* return the original source book, or we alter the data to include identifiers.

For now, let's take the first approach. When we get to networking, we can revisit the second approach.

To do this, we only need to alter our two activities.

EditBookActivity

First, let's alter the `EditBookActivity`.

We use `EXTRA_BOOK` as the key the incoming source `Book`. Let's change this to use two keys.

Replace `EXTRA_BOOK` with:

```
val EXTRA_SOURCE_BOOK = "${EditBookActivity::class.java}.name::source"
val EXTRA_EDITED_BOOK = "${EditBookActivity::class.java}.name::edited"
```

Then we'll update the `getIntent` to handle both parameters. Since we are using this method both coming into the editing as well as returning from it, we'll make the `edited` book optional.

```
fun getIntent(context: Context, source: Book, edited: Book? = null): Intent {
    val gson = (context.applicationContext as ReadingListApp).gson
    return Intent(context, EditBookActivity::class.java).apply {
        putExtra(EXTRA_SOURCE_BOOK, gson.toJson(source))
        edited?.let {
            putExtra(EXTRA_EDITED_BOOK, gson.toJson(edited))
        }
    }
}
```

Cleaning up any of the red error messages, we update our `extraBook`

```
private val extraBook: String? by lazy { intent?.getStringExtra(EXTRA_SOURCE_BOOK) }
```

And finally our `onOptionsItemSelected`

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
    return when(item?.itemId) {
        R.id.action_save -> {
            binding.book?.let {
                setResult(RESULT_OK, getIntent(
                    context = this,
                    source = it.source,
                    edited = it.edited()
                ))
                finish()
            }
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

BookListActivity

Now to clean up the errors in `BookListActivity` caused by our changes.

```
private val viewAdapter = ReadingListAdapter(
    bookClicked = {
        startActivityForResult(EditBookActivity.getIntent(
            context = this,
            source = it
        ), EDIT_REQUEST_CODE)
    }
)
```

Next, `onActivityResult` will get more complicated, so let's extract some of that functionality to make it easier.

Let's start by moving the logic out that gets the Json from the `Intent` and converts it to a `Book` object.

```
private fun getBook(data: Intent?, key: String): Book? {
    return data?.getStringExtra(key)?.let {
        app.gson.fromJson(it, Book::class.java)
    }
}
```

That's going to allow us to do

```
val source = getBook(data, EditBookActivity.EXTRA_SOURCE_BOOK)
val edited = getBook(data, EditBookActivity.EXTRA_EDITED_BOOK)
```

So we can now tweak our onActivityResult

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    // We ignore any other request we see
    if (requestCode == EDIT_REQUEST_CODE) {
        // We ignore any canceled result
        if (resultCode == Activity.RESULT_OK) {
            // Grab the books from the data
            val source = getBook(data, EditBookActivity.EXTRA_SOURCE_BOOK)
            val edited = getBook(data, EditBookActivity.EXTRA_EDITED_BOOK)
            if (source != null && edited != null) {
                // Grab the old reading list since it is read-only
                viewAdapter.readingList?.let { oldReadingList ->
                    // Create a new ReadingList
                    val newReadingList = ReadingList(
                        title = oldReadingList.title,
                        books = oldReadingList.books
                            .toMutableList()
                            .filterNot {
                                it.title == source.title
                                && it.author == source.author
                                && it.year == source.year
                            }.plus(edited)
                            .toList()
                    )
                    // Save the results
                    saveJson(newReadingList)
                    // And update our view
                    viewAdapter.readingList = newReadingList
                }
            }
        }
    }
    super.onActivityResult(requestCode, resultCode, data)
}
```

If neither the `source` or `edited` are null, then we go ahead and remove the old book and add the new one. Since we do not have some database identifier, we are filtering based off of all the data fields. Not the most efficient, but it will do for now.

Since we have currently corrupted the data, clear your application data before re-deploying your application.

Edit a book, save it, and double check your list.

All looks good, right?

Close the application and re-launch it. Where's your changes?

Take a look at the `BookListActivity.loadJson`. Do you see the problem?

Reloading changes

Our application reloads the UI when we change data, but when we restart the application, it starts over from the version of the data distributed with the application.

Let's change this to only fallback to the assets, and use our saved version by default.

We'll start by making a copy of `loadJson()` called `loadJsonFromAssets()`.

```
private fun loadJsonFromAssets(): ReadingList? {
    return try{
        val reader = InputStreamReader(assets.open(JSON_FILE))
        app.gson.fromJson(reader, ReadingList::class.java)
    } catch (e: Exception) {
        // @TODO introduce logging
        null
    }
}
```

Then we'll make a `loadJsonFromFiles()`.

```
private fun loadJsonFromFiles(): ReadingList? {
    return try {
        val reader = FileReader(File(filesDir, JSON_FILE))
        app.gson.fromJson(reader, ReadingList::class.java)
    } catch (e: Exception) {
        // @TODO introduce logging
        null
    }
}
```

You'll notice that it's almost identical - but we change the reader to reflect our prior `saveJson` method.

Finally, update our `loadJson()` function to call both of these.

```
private fun loadJson(): ReadingList? {
    return loadJsonFromFiles() ?: loadJsonFromAssets()
}
```

This will call `loadJsonFromFiles()` and return the results; however, if the results are null, then it will instead return the results of `loadJsonFromAssets()`.

Redeploy your application. Make some changes, exit the app and relaunch it.

Add new entries

To finish off this chapter, let's add the ability to not just edit existing books, but add entirely new ones.

First, we'll need a button to click.

In your module-level `build.gradle` add a new dependency on the material components

```
implementation 'com.google.android.material:material:1.1.0-alpha01'
```

Then, in your `activity_book_list.xml`, we'll insert a new `FloatingActionButton` at the end

```
<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/cornsilk"
        tools:context=".BookListActivity">

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/recycler"
            android:scrollbars="vertical"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

        <com.google.android.material.floatingactionbutton.FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="end|bottom"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            android:src="@android:drawable/ic_input_add"
            android:layout_margin="16dp" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

That will add our FAB icon to the bottom right corner using the `colorAccent` specified in our styles.

Deploy to see what it looks like.

Now, we need to handle the button click. Let's go ahead and define this one programmatically.

Open `BookListActivity`.

In your `companion object` add another request code.

```
const val NEW_REQUEST_CODE = 1235
```

This will allow us to differentiate between new book requests vs edit book requests.

In your `onCreate`, after configuring the `recycler` let's add a snippet.

```
binding.fab.setOnClickListener {
    startActivityForResult(EditBookActivity.getIntent(
        context = this
    ), NEW_REQUEST_CODE)
}
```

Uh oh, there is a red warning. What does it say? `No value passed for parameter source`.

Ctrl+Clicking on `getIntent` will take you to that method in `EditBookActivity`. We can see that the `source: Book` is currently listed as required. Let's make some changes so that can be optional.

First, add a `?` to the `Book`; then we will need to wrap the `putExtra` in a check like we did for `edited`.

```
fun getIntent(context: Context, source: Book? = null, edited: Book? = null): Intent {
    val gson = (context.applicationContext as ReadingListApp).gson
    return Intent(context, EditBookActivity::class.java).apply {
        source?.let {
            putExtra(EXTRA_SOURCE_BOOK, gson.toJson(source))
        }
        edited?.let {
            putExtra(EXTRA_EDITED_BOOK, gson.toJson(edited))
        }
    }
}
```

If we follow the trail, we will see that `extraBook` already handles being null

```
private val extraBook: String? by lazy { intent?.getStringExtra(EXTRA_SOURCE_BOOK) }
```

However, inside `onCreate` does not. We'll need to adjust that.

```
extraBook?.let {
    binding.book = EditableBook(
        source = app.gson.fromJson(it, Book::class.java)
    )
} ?: let {
    binding.book = EditableBook()
}
```

There's another one. `EditableBook` requires the `source` to be non-null. Let's fix that as well.

Changing the constructor to take a `Book?` makes red errors appear in the ObservableFields. We need to add null checks there as well.

```
class EditableBook(val source: Book? = null) {
    val title = ObservableField<String>(source?.title)
    val author = ObservableField<String>(source?.author?.displayName())
    val year = ObservableField<String>(source?.year)
```

So what's missing?

Go back to `BookListActivity`. Notice that our `onActivityResult` will only handle `EDIT_REQUEST_CODE`.

We could make a new section to handle `NEW_REQUEST_CODE`, but for now it seems like the code can handle both use cases.

```
if (requestCode == EDIT_REQUEST_CODE || requestCode == NEW_REQUEST_CODE) {
```

If we do start having if/else here, we might consider switching to a `when` clause instead.

There is one more caveat here as well. Notice this line

```
if (source != null && edited != null) {
```

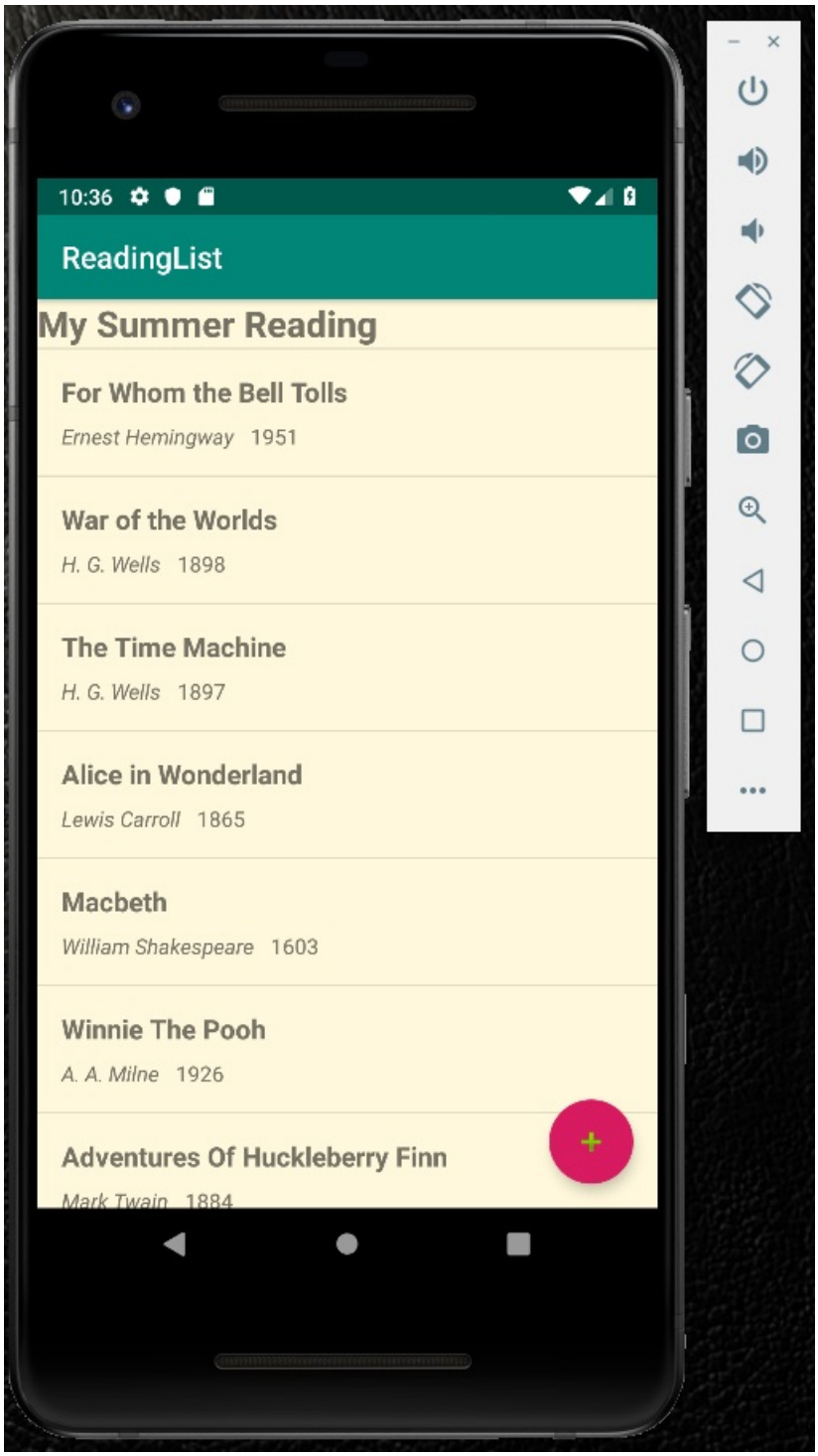
With the new changes, our `source` could be null. We need to allow that use case; which means adding more `?`s down below.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    // We ignore any other request we see
    if (requestCode == EDIT_REQUEST_CODE || requestCode == NEW_REQUEST_CODE) {
        // We ignore any canceled result
        if (resultCode == Activity.RESULT_OK) {
            // Grab the books from the data
            val source = getBook(data, EditBookActivity.EXTRA_SOURCE_BOOK)
            val edited = getBook(data, EditBookActivity.EXTRA_EDITED_BOOK)
            if (edited != null) {
                // Grab the old reading list since it is read-only
                viewAdapter.readingList?.let { oldReadingList ->
                    // Create a new ReadingList
                    val newReadingList = ReadingList(
                        title = oldReadingList.title,
                        books = oldReadingList.books
                            .toMutableList()
                            .filterNot {
                                it.title == source?.title
                                && it.author == source?.author
                                && it.year == source?.year
                            }.plus(edited)
                            .toList()
                    )
                    // Save the results
                    saveJson(newReadingList)
                    // And update our view
                    viewAdapter.readingList = newReadingList
                }
            }
        }
    }
    super.onActivityResult(requestCode, resultCode, data)
}
```

Deploy it and try clicking the new FAB button.

Make a new book and save it. Verify that it is in the list.

Now, just to be sure, exit the application and restart it. Is the book still there?



#

- Current Repo: v1-Advanced-UI
- Continue to Debugging -->

Animation.md

Goal

Explore some basic animations.

Table of Contents

- Card Flips
- Cross Fades

Let's explore some animations.

Card Flips

With a card flip animation, we can make it look like your reading list and edit book form are two sides of a card (like from a deck of cards).

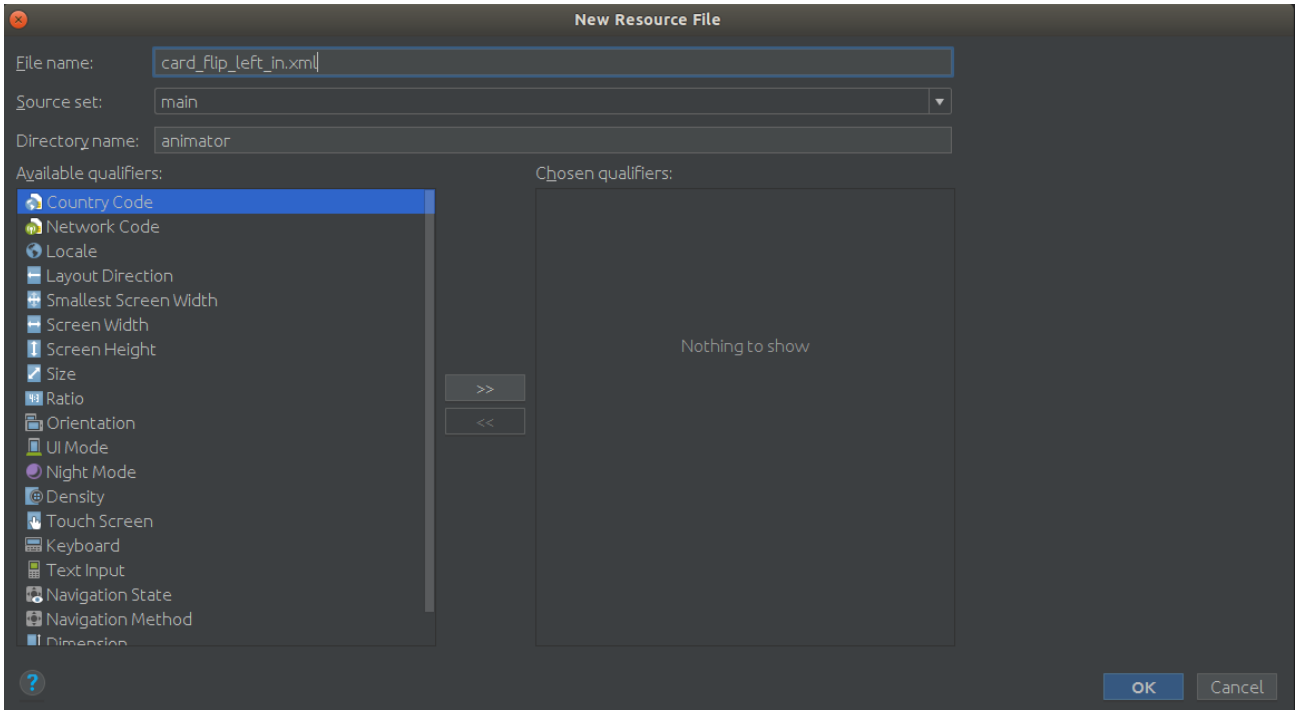
For this, we will use the XML resources in the Android Developer site tutorial. Unfortunately, that tutorial is missing a few pieces that we will need to add.

Right-click on your `res` folder and choose `New | Android resource directory`. On the `Resource type` field, choose `animator` then click `OK`.

> You may have to scroll up to see `animator`. They are alphabetical.

card_flip_left_in.xml

Right-click on the new `animator` directory and choose `New | Animator resource file`.



Give it a name of `card_flip_left_in.xml` and click `OK`.

Copy over this snippet from the Android tutorial:

```xml
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="-180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

There are three property changes that happen as part of this animation resource.

The first one changes the `alpha` (ie: transparency) from full (1.0) to none (0.0) with no delay (duration of 0).

The second one rotates around the Y axis from -180 degrees to 0 degrees. IE we are going to flip to the back of the card.

The third one sets the `alpha` back to full, but with a duration, which means we'll see it happen.

The IDE will give you an error because `@integer/card_flip_time_full` and `@integer/card_flip_time_half` are not defined.

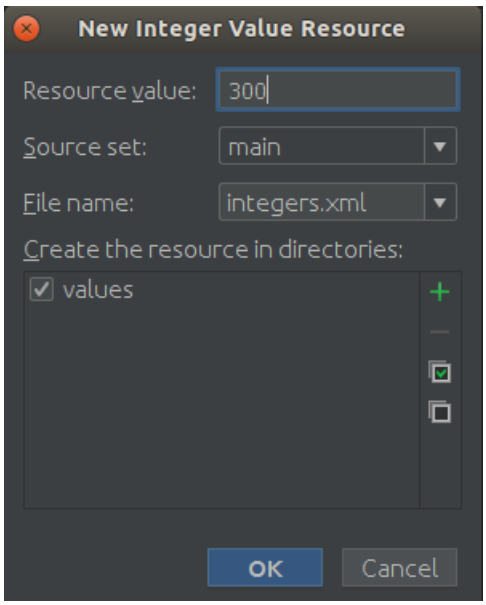
With a little bit of sleuthing, we can determine what the intention was for these values. The AOSP shows that full should be 300 and half should be 150.

integers.xml

Let's create them.

Click the `card_flip_time_full` and use the IDE Quick Fix to `Create integer value resource`.

> Remember you can access the IDE quick-fix by clicking on the light bulb or pressing Alt+Enter (Option+Enter on Mac)



Set the value to `300`. Note that this screen doesn't tell you which value you are editing, so if you are unsure, cancel and redo the process.

Do the same process for the `card_flip_time_half`, setting it to `150`.

To confirm what it created, Ctrl+Click on `card_flip_time_half` to open the new `integers.xml` file.

card_flip_left_out.xml

Create another animation resource, `card_flip_left_out.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

This one rotates the Y axes from 0 back to 180, with a delay turning off the transparency.

card_flip_right_in.xml

Next we create `card_flip_right_in.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

This one is similar to the first one but starts at 180 degrees instead of -180 degrees.

card_flip_right_out.xml

Lastly, create the `card_flip_right_out.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="-180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

As you might expect, this one is similar to our second one, but uses -180 degrees instead of 180 degrees.

Update BookListFragment

There are two places in `BookListFragment` where we call `beginTransaction`. For each of them, we will want to specify our custom animation.

Open `BookListFragment`.

our viewAdapter

Update your `viewAdapter`

```
private val viewAdapter = ReadingListAdapter(
    bookClicked = { book ->
        app?.let { readingListApp ->
            fragManager?.let {
                it.beginTransaction()
                    .setCustomAnimations(
                        R.animator.card_flip_right_in,
                        R.animator.card_flip_right_out,
                        R.animator.card_flip_left_in,
                        R.animator.card_flip_left_out)
                .replace(R.id.container, EditBookFragment.newInstance(
                    app = readingListApp,
                    source= book))
                .addToBackStack(null)
                .commit()
            }
        }
    }
)
```

our FAB

Update your `fab` in `onCreateView`

```
binding.fab.setOnClickListener {
    app?.let { readingListApp ->
        fragManager?.let {
            it.beginTransaction()
                .setCustomAnimations(
                    R.animator.card_flip_right_in,
                    R.animator.card_flip_right_out,
                    R.animator.card_flip_left_in,
                    R.animator.card_flip_left_out)
            .replace(R.id.container, EditBookFragment.newInstance(app = readingListApp))
            .addToBackStack(null)
            .commit()
        }
    }
}
```

Deploy your application and edit a book.

> You'll notice that only the reading list portion of the page swaps. The title bar remains in place. That's because the title bar is defined at the activity-level and the reading list is at the fragment level.

Cross Fades

What if you would like to cross fade between two different views?

For our application, let's look at the idea of providing some feedback while we are saving a book (after we edit it).

fragment_edit_book.xml

We'll start by editing `fragment_edit_book.xml`. We want to wrap our existing `TableLayout` in a `FrameLayout`, like so...

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <variable
            name="book"
            type="com.aboutobjects.curriculum.readinglist.model.EditableBook" />
    </data>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TableLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent">
...
        </TableLayout>
    </FrameLayout>
</layout>
```

The easiest way to do that is to add the `FrameLayout` in (`match_parent` for height and width), then cut-n-paste the `TableLayout` into it. The IDE will auto-adjust the formatting.

While we are here, add an `id` to the `TableLayout`

```
android:id="@+id/details"
```

Inside the `FrameLayout`, but after the `TableLayout`, let's add a wait message.

```
    </TableLayout>

    <LinearLayout
        android:id="@+id/please_wait"
        android:orientation="vertical"
        android:gravity="center"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            style="@style/MyTitle"
            android:text="Please wait..."
            android:gravity="center"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <ProgressBar
            style="?android:progressBarStyleLarge"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center" />
    </LinearLayout>
</FrameLayout>
```

We are just creating a standalone layout that shows a text message and a progress bar. The progress bar is getting its' style from the Android framework.

Use the IDE Quick Fix to `Extract string resource` and name it `wait_title`.

We've edited the XML, so redeploy to regenerate the binding classes.

EditBookFragment

Open your `EditBookFragment`.

We only want to show the new `pleaseWait` layout when we are saving. In your `onCreateView` let's hide it temporarily

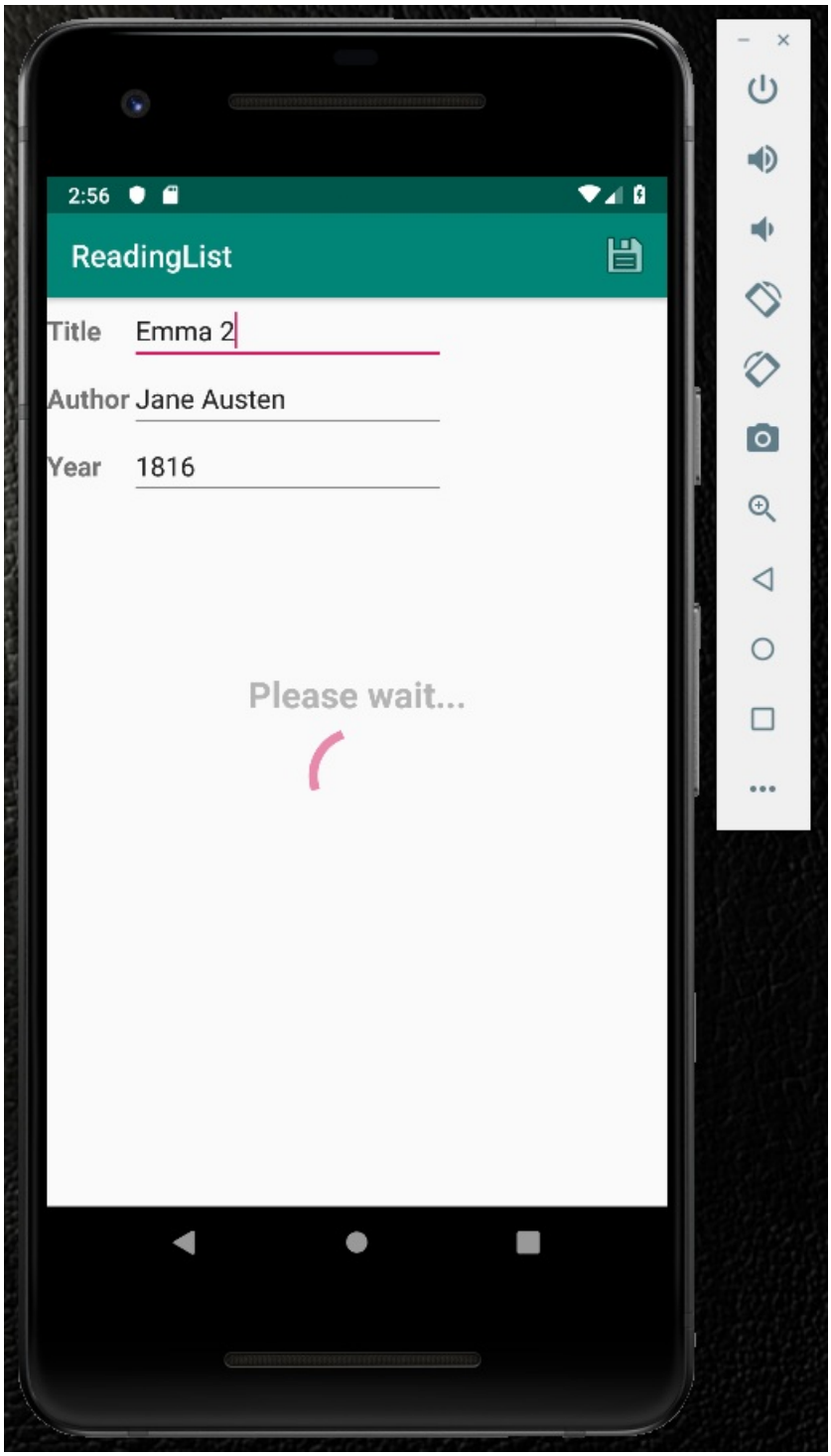
```
binding.pleaseWait.apply {
    visibility = View.GONE
    alpha = 0f
}
```

Then, in our `onOptionsItemSelected`, we will show it (by changing the alpha) over the course of... how about 3 seconds? Then, *if there is an error*, we will gradually hide it (again, change the alpha) a bit quicker... say, 1 second?

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when(item.itemId) {
        R.id.action_save -> {
            binding.book?.let { editableBook ->
                bookListService?.let { service ->
                    binding.pleaseWait.apply {
                        visibility = View.VISIBLE
                        animate()
                            .alpha(1f)
                            .setDuration(3000)
                            .setListener(null)
                    }
                    service.edit(
                        source = editableBook.source,
                        edited = editableBook.edited()
                    )
                        .subscribeOn(Schedulers.newThread())
                        .observeOn(AndroidSchedulers.mainThread())
                        .subscribeBy(
                            onComplete = {
                                // It's updated, so we are done
                                activity?.supportFragmentManager?.popBackStack()
                            },
                            onError = { t ->
                                t.message?.let { msg ->
                                    binding.pleaseWait.animate()
                                        .alpha(0f)
                                        .setDuration(1000)
                                        .setListener(object: AnimatorListenerAdapter() {
                                            override fun onAnimationEnd(animation: Animator?) {
                                                binding.pleaseWait.visibility = View.GONE
                                                Snackbar.make(binding.root, msg, Snackbar.LENGTH_LON
G)
                                                    .show()
                                            }
                                        })
                                }
                            }
                        )
                }
            }
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

If you deploy your application, you are unlikely to see the change.

You can *temporarily* test it by having `BookListService.edit` immediately return `error(IllegalArgumentException("TESTING"))`



Why don't we change the alpha back in the `onComplete` callback?

Since we are immediately popping the backstack, it seemed an unnecessary delay for the end user. This entire Fragment is about to go away. In the case of the `onError`, the Fragment stays on the screen with the error message in the Snackbar - so having the progress indicator go away gracefully is a nice touch.

More Information

For more information on Animation, please see the Android Developer site.

#

- Current Repo: v1-Animation
- Continue to Networking -->

Basic-UI.md

Goal

Generate a basic UI project, learn how the pieces fit together and deploy it to the emulator.

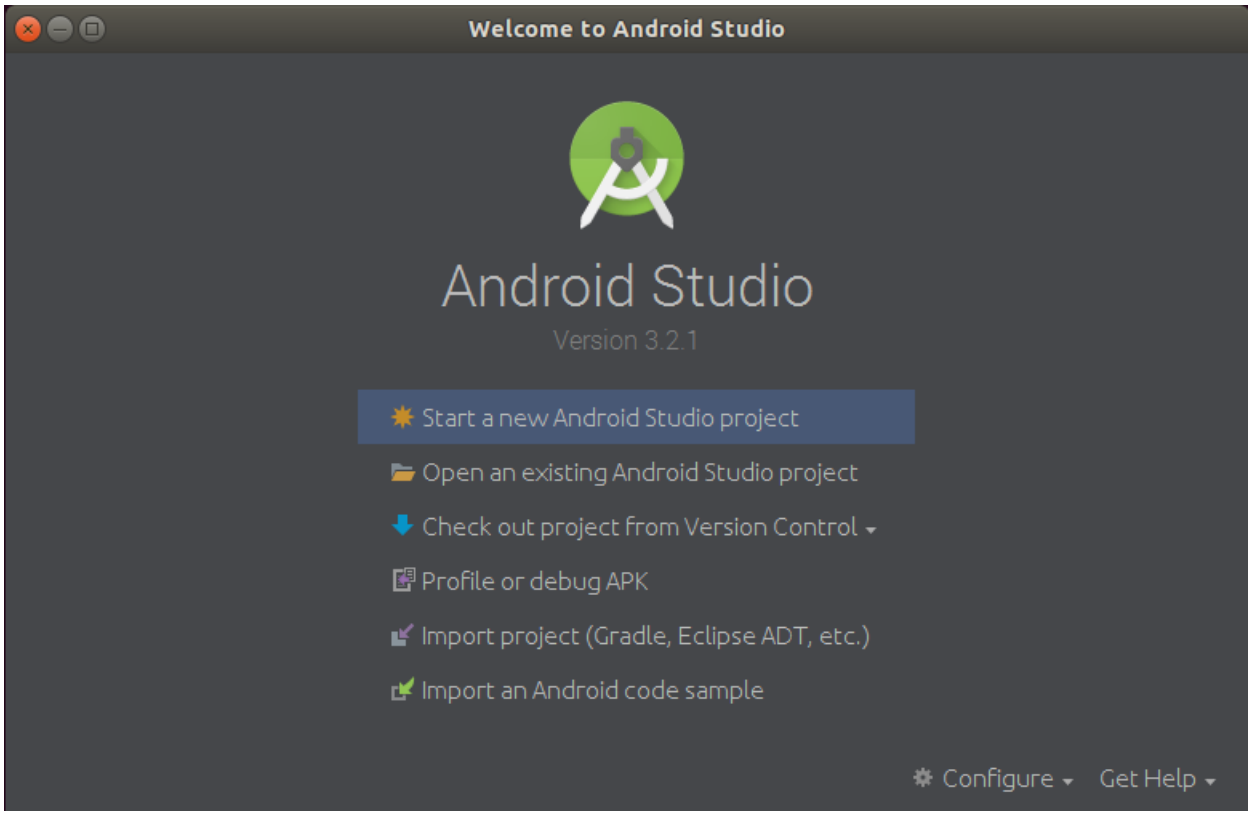
Table of Contents

- Autogenerated UI
  - Creating the Project
  - The Interface
  - What was Generated?
- Emulator
  - Setup
  - Deploy
- Tweak the UI
  - Layouts
  - Strings
  - Colors
  - Styles

Autogenerated UI

Creating the Project

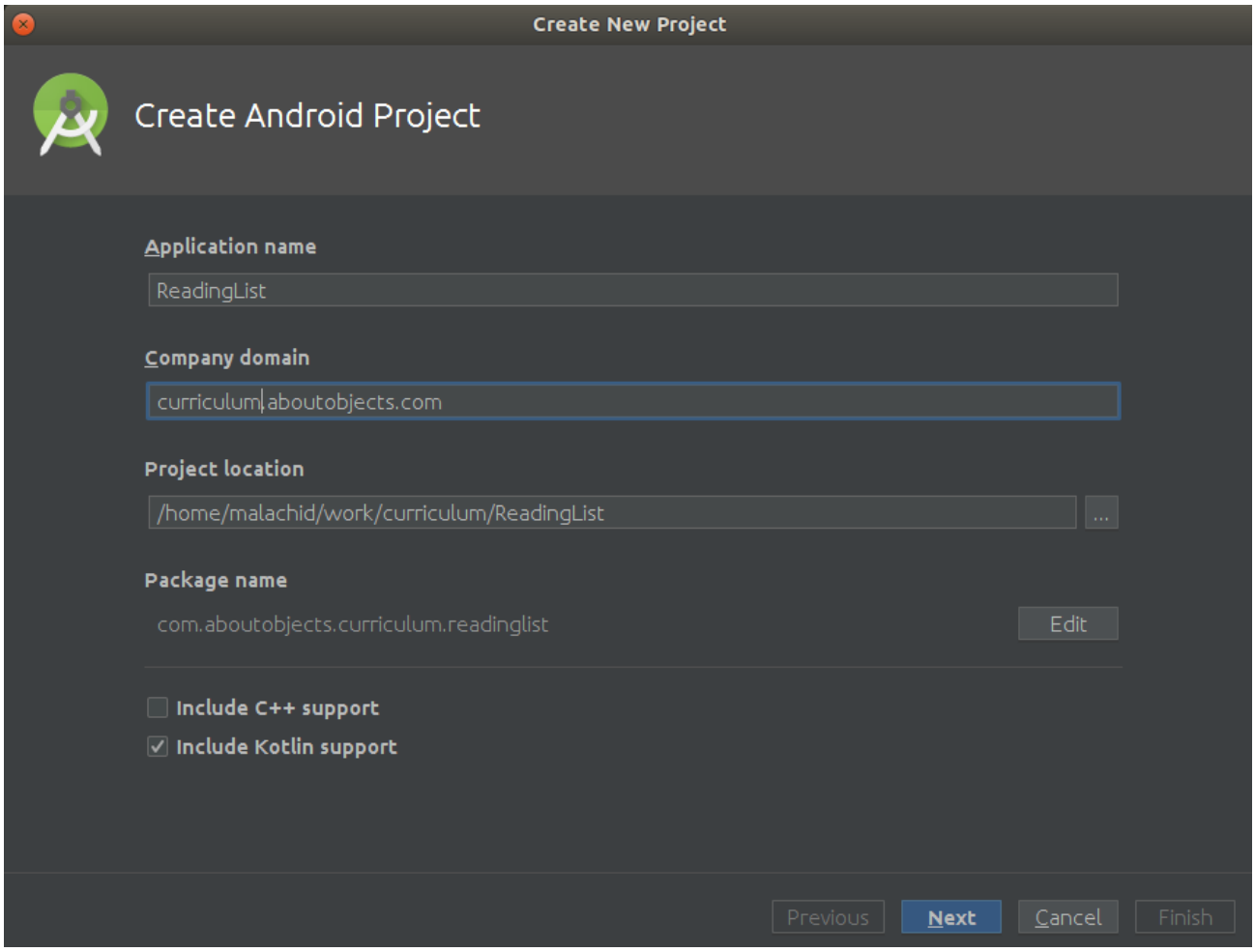
If you have not already done so, launch Android Studio.



On this screen, click `Start a new Android Studio project`.

Create Android Project

You will see a Create Android Project wizard.



Let's review the options on this page.

Application name

This is the visible name of your project. Think of it as the title of your app.

Here, we will use `ReadingList` as the name of our application.

Company domain

This is the domain of the owner of the application. It's also visible to the end users in Google Play.

I'd recommend that you use a subdomain rather than your top-level domain.

In this example, we are using `curriculum.aboutobjects.com`.

Project location

This tells the IDE where to create a new directory for your project.

You will notice that the end of the path is automatically populated with the Application name and does not end in a slash.

For this project, you will need to tailor this entry to match your personal machine. Pick a directory where you want to store your files.

Package name

By default, and convention, this is the reverse of your company name. You will note that the application name is also appended as if it were a subdomain as well.

While you *could* edit this, it is not recommended. It's not only best practice, but expected for any professional application or library.

C++ and Kotlin

At the bottom you will see options for using C++ and/or Kotlin.

We will not be using C++ for this class.

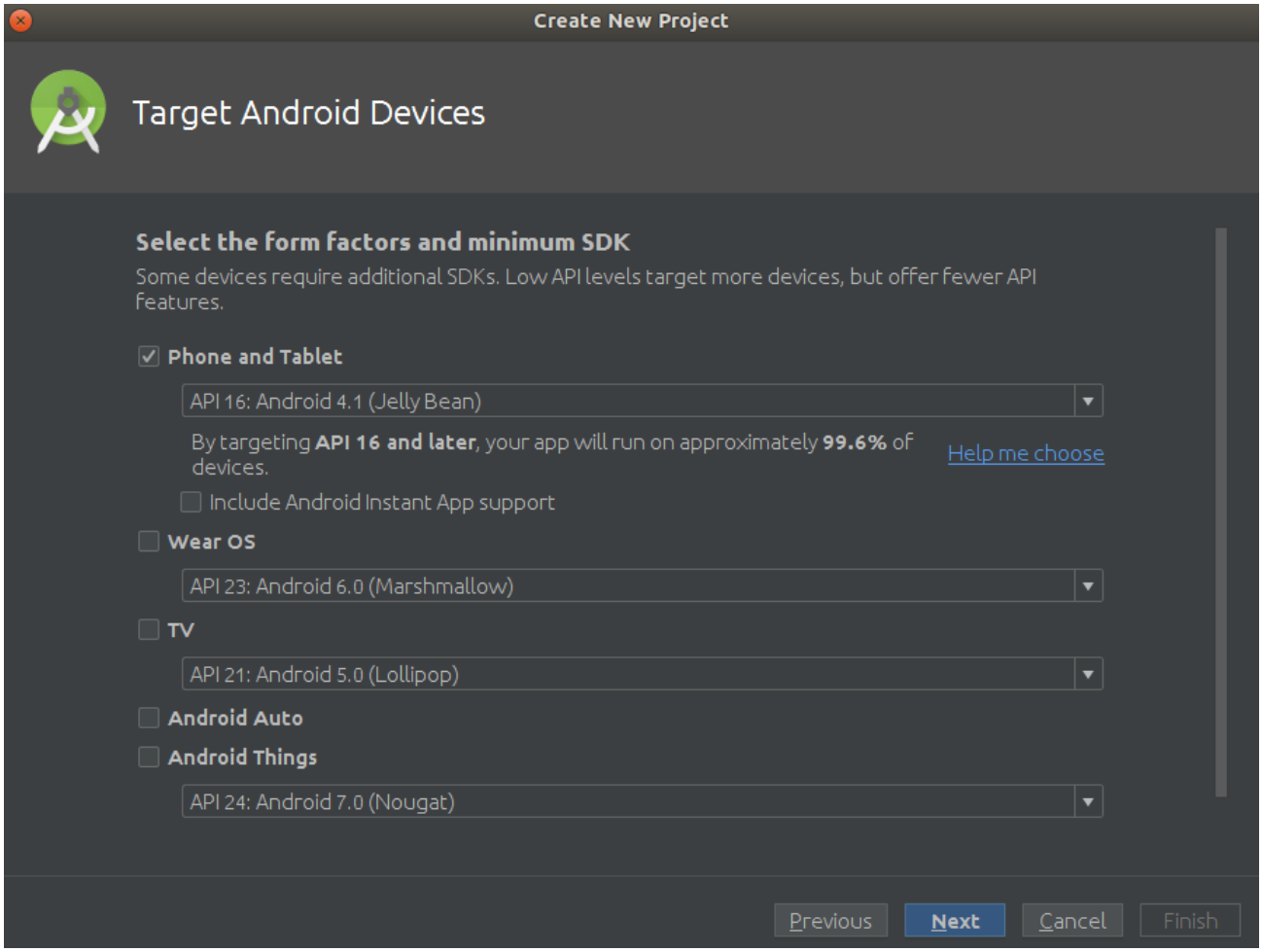
Many Android applications are written in Java. If you do not select the Kotlin checkmark, the generated project will be a Java-based project instead. You can still use both within the project if you need/want. Google is pushing pretty hard that Android developers start using Kotlin, and in many cases it does remove a lot of boilerplate code. For the purposes of this class, we will be using Kotlin to reduce the amount of boilerplate code you will need to type.

Make sure `Kotlin` is checked.

Click `Next`.

Target Android Devices

On this screen, you are asked to choose which Android devices you application is designed for.



For each section, you can select whether a particular category is supported; and if so, at what API level. For this class, we are not covering watches or TVs or cars - so only the first category, `Phone and Tablet` will be selected.

Inside of that category, we have to choose an API level. This relates directly to what you learned about Android Versioning in the last chapter.

By default, it selects `API 16` and tells you that it will support 99.6% of Phone/Tablet devices.

If you recall, API 16 only accounted for 1.1% of the market. While you may want to support all devices (and some applications may need to), it is unlikely that many employers will want to spend the time/cost/resources on maintaining the application for that version. As mentioned earlier, APIs change. You will have sections of your code that only work on API 16-20, for example; thus increasing the maintenance cost.

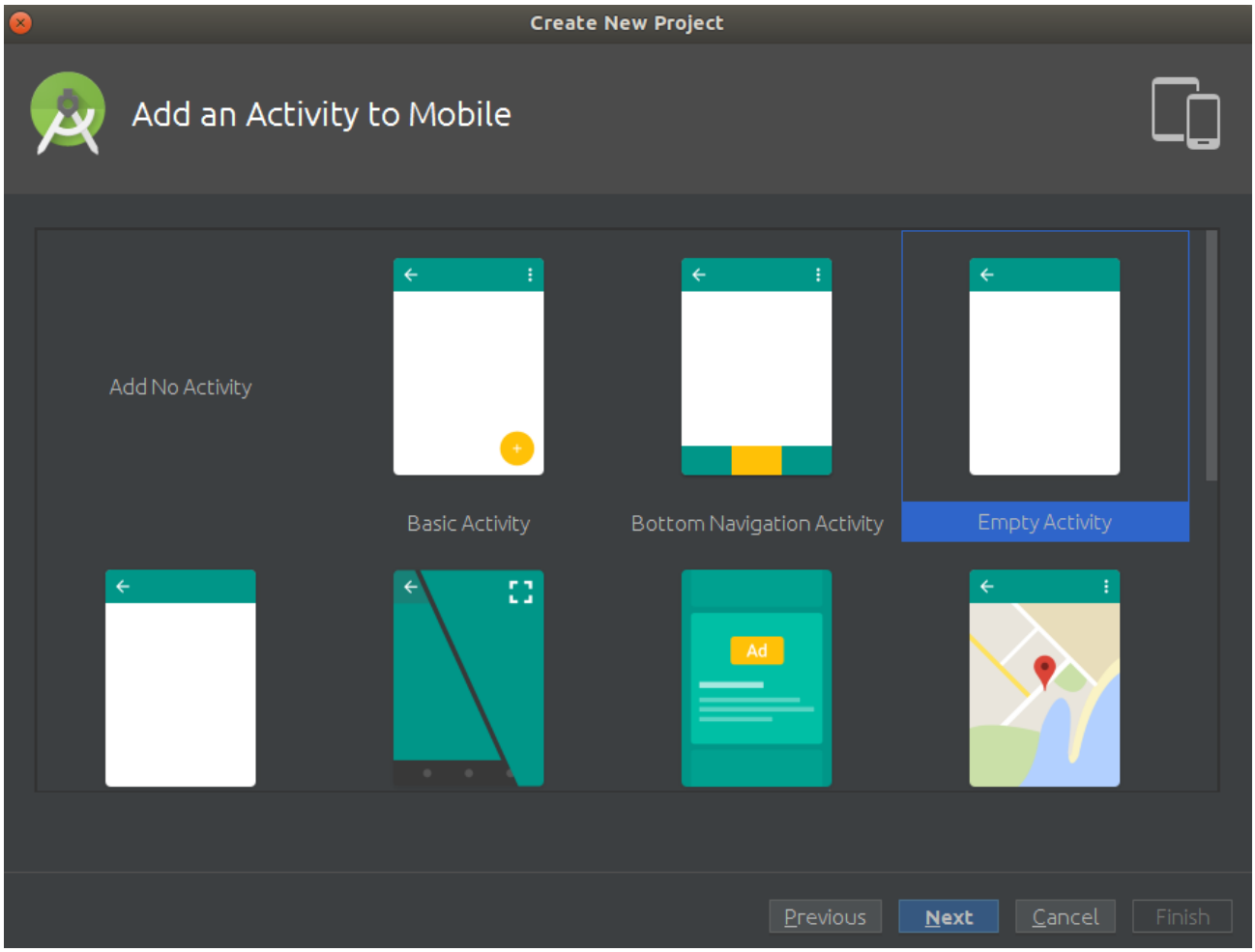
For this class, let's change that selector to `API 22`. As we discussed earlier, that was the first API level that had double-digit distribution. When we do, we see that it still reaches 80.2% of the market.

These numbers change over time. You will need to work with your product owners to determine what the correct break-off point is for your own application.

For now, once `Phone/Tablet` is selected and set to `API 22`, click `Next`.

Add an Activity to Mobile

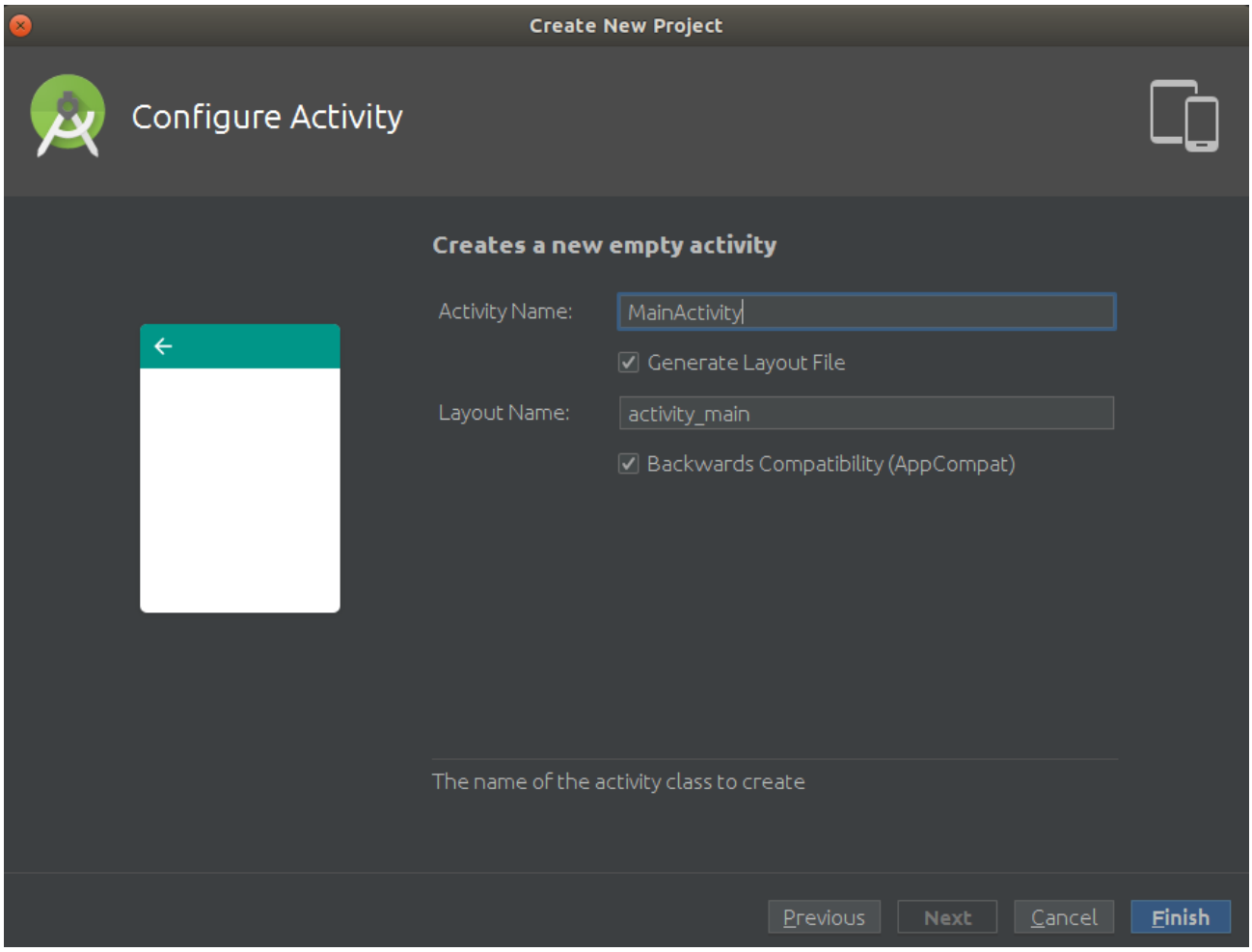
The next screen allows you to choose which type of Hello World to auto-generate.



We will choose the `Empty Activity`. Click on it then click on `Next`.

Configure Activity

On the final step of the wizard, you will be asked to configure you activity.



It is important to know that the options presented on this screen will be different depending on the type of project you asked it to generate.

Activity Name

This will be the name of the activity class that is used to launch your application. It defaults to `MainActivity` (without the `.kt` or `.java` extension).

Many people don't bother changing it because it is the launching point of their app. Unfortunately, after a few major revisions, they introduce things like splash screens or seasonally change what the starting point should be. For that reason, I suggest that we name it for what it does rather than for where it is currently being called.

Let's call it `BookListActivity` to reflect that it will display a list of books. This will auto-generate `BookListActivity.kt` for us.

You will notice that when you change it, the Layout Name changed automatically to match.

Generate Layout File

We will go over the auto-generated files here shortly, but this checkmark will allow the IDE to auto-generate a starting point for you. Make sure it is checked.

Layout File

This is the name of the xml file that will be generated. By default, the IDE will call it `activity_` followed by the name of your activity. Camel cases are replaced with underscores as the names are all lowercase.

It should be defaulting to `activity_book_list`, which in turn will auto-generate `activity_book_list.xml`.

Backwards Compatibility (AppCompat)

This option specifies whether to use `android.app.Activity` or `android.support.v7.app.AppCompatActivity` (which itself will get replaced with `androidx.appcompat.app.AppCompatActivity`). This option is due to a breaking change that was made long ago in the Android ecosystem, and yet another that occurred recently. While you could technically use either one when

only supporting newer APIs, you are going to find that many of the libraries assume that you are using the AppCompat version. Make sure it is selected.

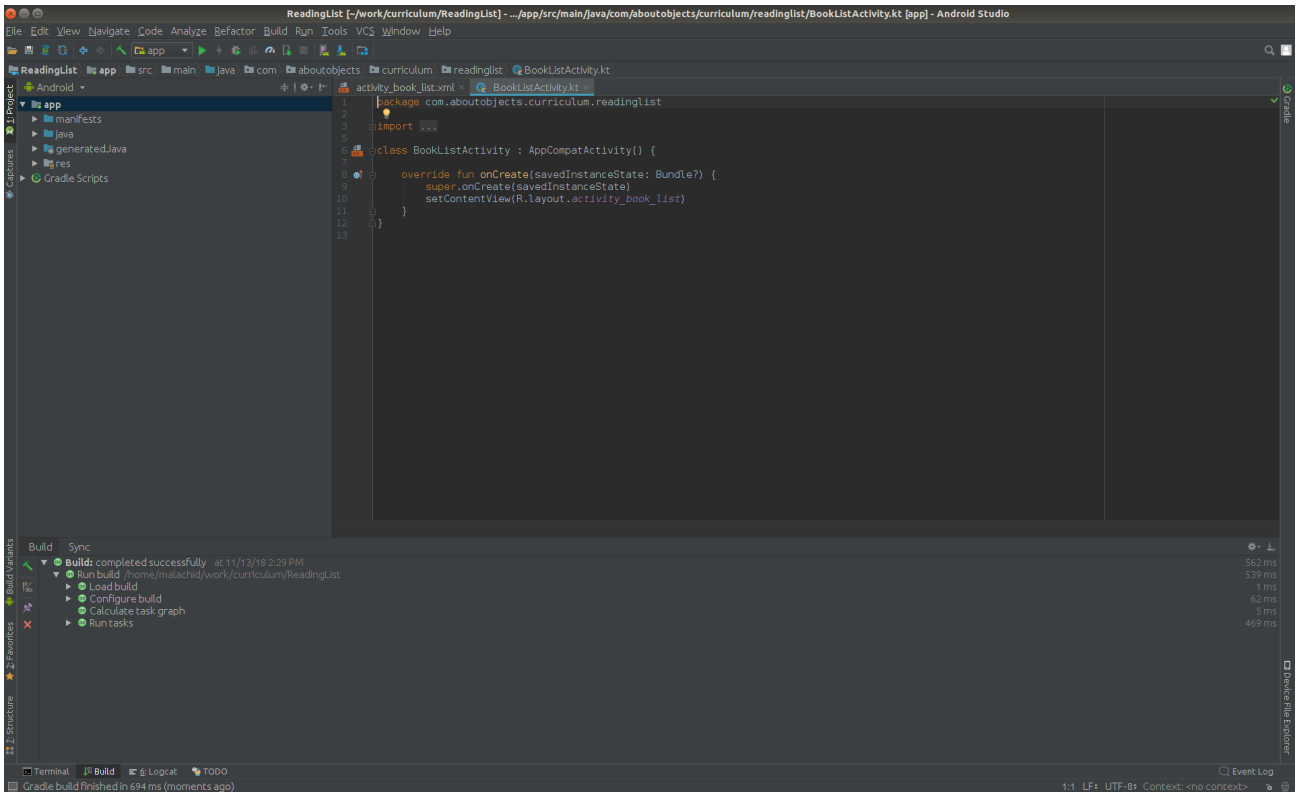
Click `Finish`.

> NOTE: If you are checking this project into source control, as you should never check in any files that are specific to your machine; consider making sure the following are in your `.gitignore`:

```
.idea/
*.iml
*.iws
local.properties
.gradle/
```

The Interface

Before we dive in to what was generated, let's take a brief look at the interface itself.



The largest portion of the screen is a tabbed editor. Through the settings, you can control how many tabs are visible, whether there are multiple lines of tabs, what color is used for text highlighting, et cetera.

Left of the editor is the project window. It will show the files in your project. The are some settings available on a tiny gear icon directly above it to control whether empty packages are hidden, among other useful features.

On the far left of the screen, left of the project window, you will see some sideways tabs. Those can be used to open additional tool windows onto that side view. You won't use those right now, but the Build Variants one is particularly useful when you start publishing your application.

Similar to those tabs, there are additional ones on the far right and far bottom of the screen. We will call them out as needed.

On the bottom of the screen, when you first opened your project, you will have noticed that the IDE automatically built your project. For now, you can click the red `X` to dismiss that bottom window. You might have to do it more than once.

Above the project and project window and the editor is the breadcrumb path. It will automatically update to show the path to the file currently visible in the editor.

At the very top of the page is the menu and toolbar. There are many options and settings in those. We will call those out as appropriate.

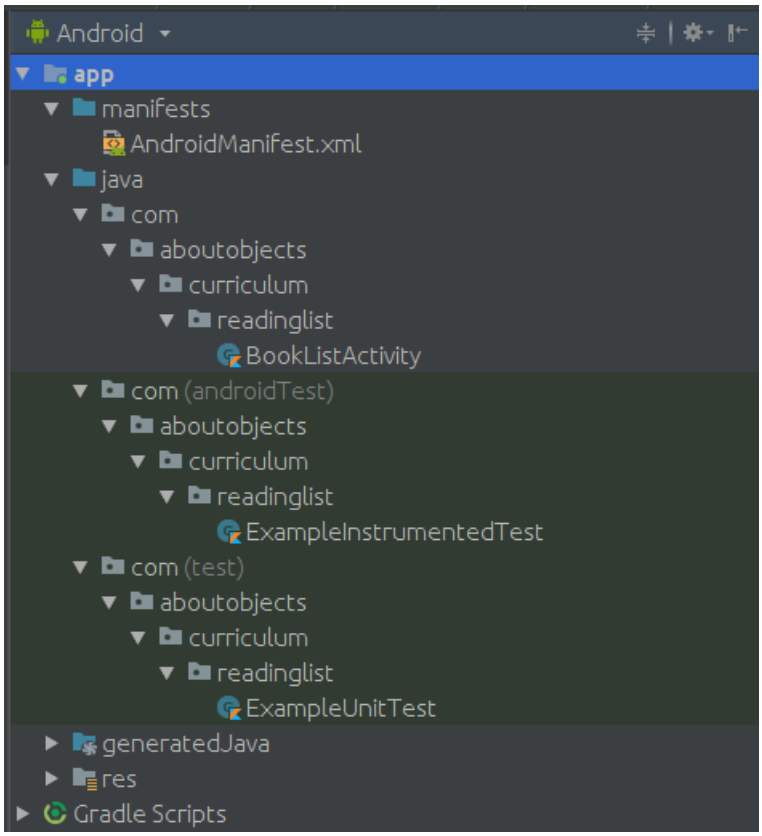
What was Generated?

If you expand the files in your project window on the top left, you will see three main categories of files.

When you open each file, you will notice that the breadcrumb above the editor shows the file path; while the project window has them arranged by functionality. For example, the `AndroidManifest.xml` is shown under `app/manifests` in the project window, but shows `ReadingList/app/src/main/AndroidManifest.xml` in the breadcrumb.

Let's take a look at each one.

Source Files



AndroidManifest.xml

As mentioned above, the `AndroidManifest.xml` is found in `app/src/main`. If you end up having different variants of your application (like free vs pro, or debug vs release) you will likely have more than one. Until that time though, the `main` directory will be used as the default.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.aboutobjects.curriculum.readinglist">

    <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:roundIcon="@mipmap/ic_launcher_round"
            android:supportsRtl="true"
            android:theme="@style/AppTheme">
        <activity android:name=".BookListActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The `manifest`/`package` is the unique identifier for your application. It will show up in Google Play and in the Settings app for the end user. It is also the key you would use to uninstall it from the command line.

Inside the `manifest` we have a single `application` tag. It specifies some global defaults for our application and acts as the container for everything else.

Inside the `application`, we have our `activity` tag for the `BookListActivity` we created. Note that the name starts with a period. That indicates that you want it to inherit the package name above. Alternatively, you could specify the fully qualified package name.

Inside the `activity` we have our `intent-filter`. This tells the system how to launch our activity. In this case, we are specifying that this particular activity is our MAIN LAUNCHER... ie: when someone runs our application, we expect this activity to be the one that starts.

Pay special attention to everything that starts with an at-symbol `@`. Those indicate references to other files that we will discuss in a moment.

More information on this file can be found on the Android Developer site.

BookListActivity.kt

If you had not selected the `Kotlin` checkmark during the `Create Android Project` wizard, this would have instead been `BookListActivity.java`. There would have been more lines of code, but the functionality would have been the same.

Even though the auto-generated files are Kotlin, note that they are included under `app/src/main/java`. If you are inheriting an older codebase, you might see `app/src/main/kotlin` - but that is no longer the preferred convention.

```
package com.aboutobjects.curriculum.readinglist

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class BookListActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_book_list)
    }
}
```

In the auto-generated Hello World, this class does not have much to it. It is using the name we specified in the wizard, extending the AppCompat version of the Activity (as we specified in the wizard) and displaying the auto-generated UI layout that we also specified in the wizard.

It is worth pointing out that while the XML file above used the `@`-style convention to reference other files (like `@string/app_name`), the Java or Kotlin code uses the `R.` programmatic convention to reference them (like `R.layout.activity_book_list`). We will see more of this as we move on.

ExampleInstrumentedTest.kt

We will discuss Instrumentation Testing in another chapter. For now, just note that the are included in the `app/src/androidTest` subdirectory.

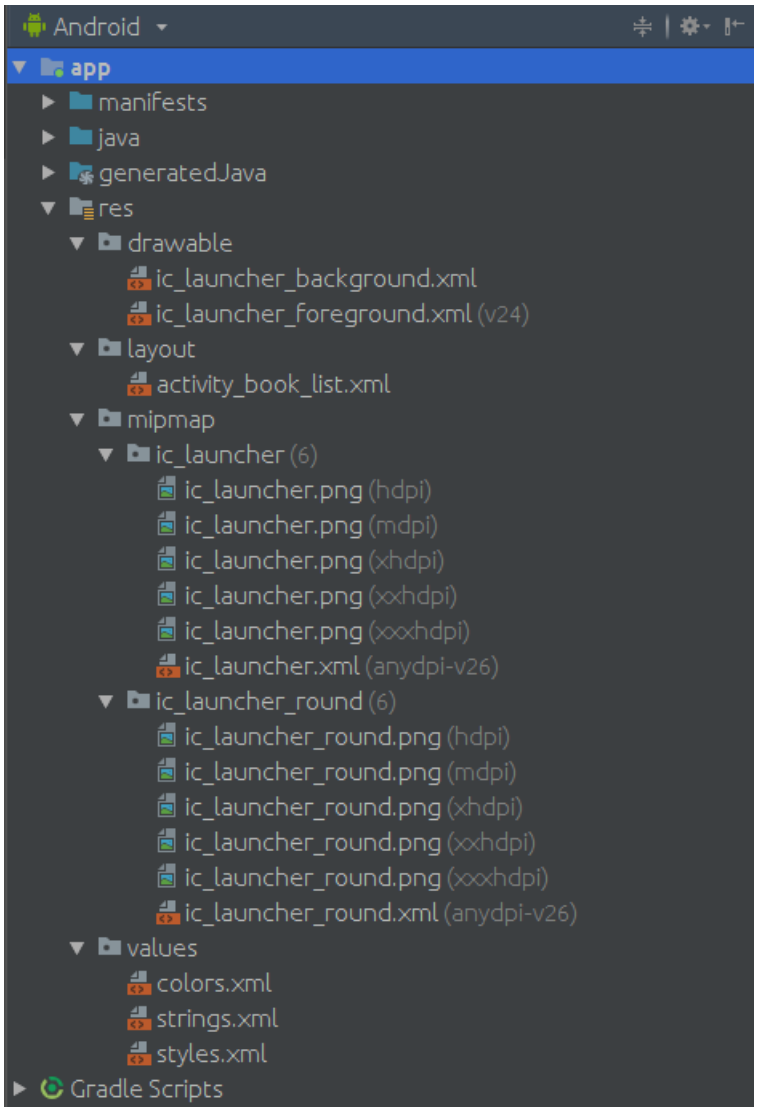
```
package com.aboutobjects.curriculum.readinglist

import android.support.test.InstrumentationRegistry
import android.support.test.runner.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getTargetContext()
        assertEquals("com.aboutobjects.curriculum.readinglist", appContext.packageName)
    }
}
```

ExampleUnitTest.kt

Similarly, we will discuss Unit Testing in that same chapter. For now, just note that the are included in the `app/src/test` subdirectory.

```
package com.aboutobjects.curriculum.readinglist

import org.junit.Test

import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

Resource Files



ic_launcher_background.xml, ic_launcher_foreground.xml

Android 8 (Oreo) introduced the concept of Adaptive Launcher Icons.

```
<?xml version="1.0" encoding="utf-8"?>
<vector
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:height="108dp"
        android:width="108dp"
        android:viewportHeight="108"
        android:viewportWidth="108">
    <path android:fillColor="#008577"
          android:pathData="M0,0h108v108h-108z"/>
    <path android:fillColor="#00000000" android:pathData="M9,0L9,108"
          android:strokeColor="#33FFFFFF" android:strokeWidth="0.8"/>
...clip...
    <path android:fillColor="#00000000" android:pathData="M69,19L69,89"
          android:strokeColor="#33FFFFFF" android:strokeWidth="0.8"/>
    <path android:fillColor="#00000000" android:pathData="M79,19L79,89"
          android:strokeColor="#33FFFFFF" android:strokeWidth="0.8"/>
</vector>
```

Creating these icons is beyond the scope of this tutorial, but there is a guide here on how to auto-generate them with Android Studio.

activity_book_list.xml

You will notice that this file is located in `src/main/res/layout`. Barring variants, this is the directory where you will put most of your layouts.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".BookListActivity">

    <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello World!"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

</android.support.constraint.ConstraintLayout>
```

A few things to point out here.

The xml namespaces at the top-level will almost always be included. If you forget them, the IDE will prompt you and auto-add them.

There can only be one top-level element, in this case a ConstraintLayout. Each type of layout has their own attributes and quirks. For example, a LinearLayout must specify an orientation, and a ScrollView can only have one child.

Here, we are specifying that the ConstraintLayout should match the parent height and width. If you start nesting layouts, you may need to know what kind of parent you are nesting into.

The TextView is set to take up the minimal amount of required space to actually show itself (`wrap_content`) and to be centered (equidistant to each of the four sides).

For more information on ConstraintLayout, you can look at the API documentation.

ic_launcher.png, ic_launcher_round.png (hdpi, mdpi, xhdpi, xxhdpi, xxxhdpi)

These are the launcher icons for different screen sizes.

On the Android Developer site you can see a table that explains what each of these dpis are intended to represent, how calculations are performed and an example of the differences between the sizes.

For our purposes today, your icons were auto-generated by the IDE. When you are ready to make your own images it is recommended that you start with an SVG source and create adaptive icons as mentioned above.

ic_launcher.xml, ic_launcher_round.xml (anydpi-v26)

While these are bundled with the PNGs, they just tell the system to use the adaptive icon files above.

```
<?xml version="1.0" encoding="utf-8"?>
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
    <background android:drawable="@drawable/ic_launcher_background"/>
    <foreground android:drawable="@drawable/ic_launcher_foreground"/>
</adaptive-icon>
```

Note that the `.xml` extension is not included in the references.

colors.xml

Rather than hard-coding colors throughout your code, you can specify them in a single file and refer to them by name. That makes it easier to make global changes to your app (like, say, applying a holiday theme).

While this file is located in `src/main/res/values` it is possible to have different variants of your application use different color schemes just by having two different versions of this file.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#008577</color>
    <color name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>
</resources>
```

These colors are represented by ARGB hex values. The `colorPrimary` is used for the bar at the top of your application as well as some other UI. The `colorPrimaryDark` is used for the status bar, above your application bar as well as some contextual UI. The `colorAccent` is used for controls like text fields and checkboxes.

If you are unfamiliar with ARGB hex values, you can head over to W3C to help you pick colors.

strings.xml

In the same directory, you will find your `strings.xml`. While it currently only contains one string, it will become a very important file for you. Every time you consider hard-coding any text in your app, it probably should go in this file instead.

```
<resources>
    <string name="app_name">ReadingList</string>
</resources>
```

Currently, it only contains the name of your application. You might remember that we encountered it before when we discussed the AndroidManifest:

```
android:label="@string/app_name"
```

In this reference, the manifest specified that the label was the `app_name` entry in this `strings.xml` file.

If you go back and look at the manifest, you will see the icons were referenced in the same way.

styles.xml

The initial version of our `styles.xml` might seem a little confusing.

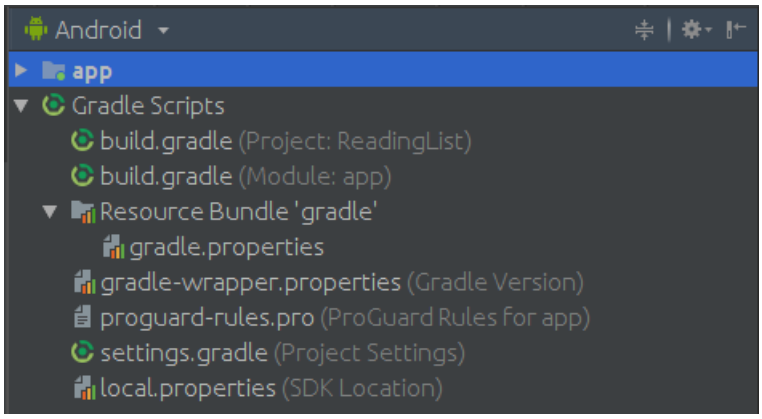
```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

Why are we setting `colorPrimary` to itself, you may ask?

Actually, the `Theme.AppCompat.Light.DarkActionBar` theme has a `colorPrimary` property. By default, it doesn't know what color you set in the `colors.xml`. You could have called it anything, maybe `fred`. Here, you are assigning your chosen color to the property in the theme.

As we continue our lessons, we will re-visit this configuration file to control other parts of the UI styling. For now, it's enough to know that this is where it is done.

Build Files



The Android project that was auto-generated is built using Gradle. It is not the only way to build Android projects, but it is the one we will focus on.

build.gradle (Project: ReadingList)

You will notice that there are two `build.gradle` files. One says `Project` and one says `Module`. You will have one for the top-level all-encompassing project, and one for every module in it. Currently, we have one module.

The project-level `build.gradle` specifies anything that is common between all modules. Since we only have one module, the most important thing is that it sets us to to do a build.

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.10'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.2.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

The first thing you will notice is that the file specifies two repositories, not once but twice. These repositories are generally Maven repositories that contain third party libraries. For example, take a look at jcenter mentioned above.

But, why is it listed twice?

The first one, inside the `buildscript` closure, allows the build tools to resolve their own dependencies.

The second one, inside the `allprojects` closure, allows your modules to resolve their dependencies.

They do not have to have the same list.

The order in each list does, however, matter. Sometimes a dependency will show up in more than one repository. The first match will be the one that is used. The recommended approach is to list your internal repository first, if you have one, so that is can cache and proxy results reducing bandwidth, costs and time. In addition, you might want faster or more reliable repositories to be higher on the list.

The other thing you might notice about this file is that it specifies the version of the `com.android.tools.build:gradle` and `org.jetbrains.kotlin:kotlin-gradle-plugin` dependencies. In this case, we have specified that we will use the Android Gradle plugin 3.2.1 and the Kotlin Gradle plugin 1.3.10.

> While you can specify Gradle dependencies as Group, Artifact and Version; they are almost always specified as a single string in the form "Group:Artifact:Version"

build.gradle (Module: app)

The module-level build.gradle controls the building for the specific module, in this case named 'app'. While not very creative, it is the default name given to application projects created by the IDE.

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.aboutobjects.curriculum.readinglist"
        minSdkVersion 22
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
}
```

Let's walk through this one.

The first line does `apply plugin: 'com.android.application'` which tells the build system that we are building an Android application. Not a middleware library, not a Java webservice, et cetera. This in turn adds additional functionality and steps to the build.

The next two plugins, `kotlin-android` and `kotlin-android-extensions` enable the use of Kotlin in the build.

After that we have two closures, `android` and `dependencies`.

In the `android` closure, we specify the `compileSdkVersion`. This is the version of the Android API level we are using to compile and *MUST* be installed in the SDK Manager. You should strive to make this the latest version at all times as it will help you catch problems early.

Inside the `defaultConfig` closure, we specify our `applicationId` which matches our packageName in the AndroidManifest.

The `minSdkVersion` is the minimum API level that we support. During the Create Project Wizard, we changed this from 16 to 22.

The `targetSdkVersion` is the highest API level your application will support. Specifically, code blocks in the framework will check this to determine whether to change behavior.

Having compileSdkVersion higher than targetSdkVersion will allow you to start working towards support for new features before

enabling them.

For a larger discussion on compileSdkVersion/minSdkVersion/targetSdkVersion, please see this article by Ian Lake from Google.

`versionCode` is an ever-increasing number whereas `versionName` is a human-readable version. The `versionName` is what shows up in Google Play to the end user. You can think of `versionCode` as more of a uniqueKey, where the newest one always has to be higher than the last.

You might be asking yourself what to use for the `versionName`. Should it be '1.0' or '1.2.3' or something else entirely. There are some unique version schemes out there. TeX, for example, uses ever increasing digits of pi for their versioning. By far, the most common and recommended version scheme is semantic versioning.

The `testInstrumentationRunner` just specifies how we will run unit tests and can be ignored for now.

Under the `buildTypes` closure, you can see there is a `release` closure. It specifies that we will use the Proguard file mentioned below. In general, we never obfuscate when doing debug builds.

In the `dependencies` closure at the bottom, we see a list of `implementation` lines, a `testImplementation` line and two `androidTestImplementation` lines. The `testImplementation` is a dependency for the unit test we saw earlier in the `src/main/test` directory. The `androidTestImplementation` are dependencies for the instrumentation tests we saw earlier in the `src/main/androidTest` directory.

Of the other `implementation` dependencies, it is important to call out the first one.

```
implementation fileTree(dir: 'libs', include: ['*.jar'])
```

This line allows `.jar` files inside the `libs` directory to be treated as dependencies. Note that it does not work for `.aar` and does not handle transitive dependencies.

> This is a *really* bad idea. You should *never* check these files into source control. They belong in an internal repository instead. You can get a free open-source version of Nexus here written by the company that wrote the O'Reilly book. You can run it locally or in the cloud - but I would recommend having one for your organization, as discussed in the project-level `build.gradle` section.

> Whenever I create a new project, I usually delete that line as soon as possible to discourage it's use.

gradle.properties

This file is checked into source control and is used to further configure the build.

```
# Project-wide Gradle settings.
# IDE (e.g. Android Studio) users:
# Gradle settings configured through the IDE *will override*
# any settings specified in this file.
# For more details on how to configure your build environment visit
# http://www.gradle.org/docs/current/userguide/build_environment.html
# Specifies the JVM arguments used for the daemon process.
# The setting is particularly useful for tweaking memory settings.
org.gradle.jvmargs=-Xmx1536m
# When configured, Gradle will run in incubating parallel mode.
# This option should only be used with decoupled projects. More details, visit
# http://www.gradle.org/docs/current/userguide/multi_project_builds.html#sec:decoupled_projects
# org.gradle.parallel=true
# Kotlin code style for this project: "official" or "obsolete":
kotlin.code.style=official
```

By default, it doesn't have much in it. You might add things to it like specifying where to find some config file or keystore.

gradle-wrapper.properties

This file tells Gradle what version to install and run.

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.6-all.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

Most people just change the `4.6` above to whatever version they want to use and rebuild, but it is possible to configure it via a closure inside the project-level `build.gradle`.

proguard-rules.pro

When you go to release your application, you will need to tell Proguard which files/methods/variables to *NOT* obfuscate.

```
# Add project specific ProGuard rules here.
# You can control the set of applied configuration files using the
# proguardFiles setting in build.gradle.
#
# For more details, see
#   http://developer.android.com/guide/developing/tools/proguard.html

# If your project uses WebView with JS, uncomment the following
# and specify the fully qualified class name to the JavaScript interface
# class:
#-keepclassmembers class fqcn.of.javascript.interface.for.webview {
#   public *;
#}

# Uncomment this to preserve the line number information for
# debugging stack traces.
#-keepattributes SourceFile,LineNumberTable

# If you keep the line number information, uncomment this to
# hide the original source file name.
#-renamesourcefileattribute SourceFile
```

While outside the scope of this course, Google has posted a YouTube video from Google I/O 2018.

settings.gradle

This file will seem a little silly at first.

```
include ':app'
```

If you have multiple modules that you want to build, you would list them all here on a comma-separated line.

For more extensive use of this file, see the Gradle documentation.

local.properties

Unlike the gradle.properties, this file should ***NOT*** be checked into source control. This file is specific to the machine it is running on. How your machine is setup may not be the same as mine. Not only might we have different applications, directories or memory requirements - we may not even be on the same operating system. This file allows us to do machine-specific tweaking.

For example, on my machine, it auto-generated this:

```
## This file must *NOT* be checked into Version Control Systems,
# as it contains information specific to your local configuration.
#
# Location of the SDK. This is only used by Gradle.
# For customization when using a Version Control System, please read the
# header note.
#Tue Nov 13 14:29:55 PST 2018
ndk.dir=/usr/local/android-sdk/ndk-bundle
sdk.dir=/usr/local/android-sdk
```

What did yours generate?

Emulator

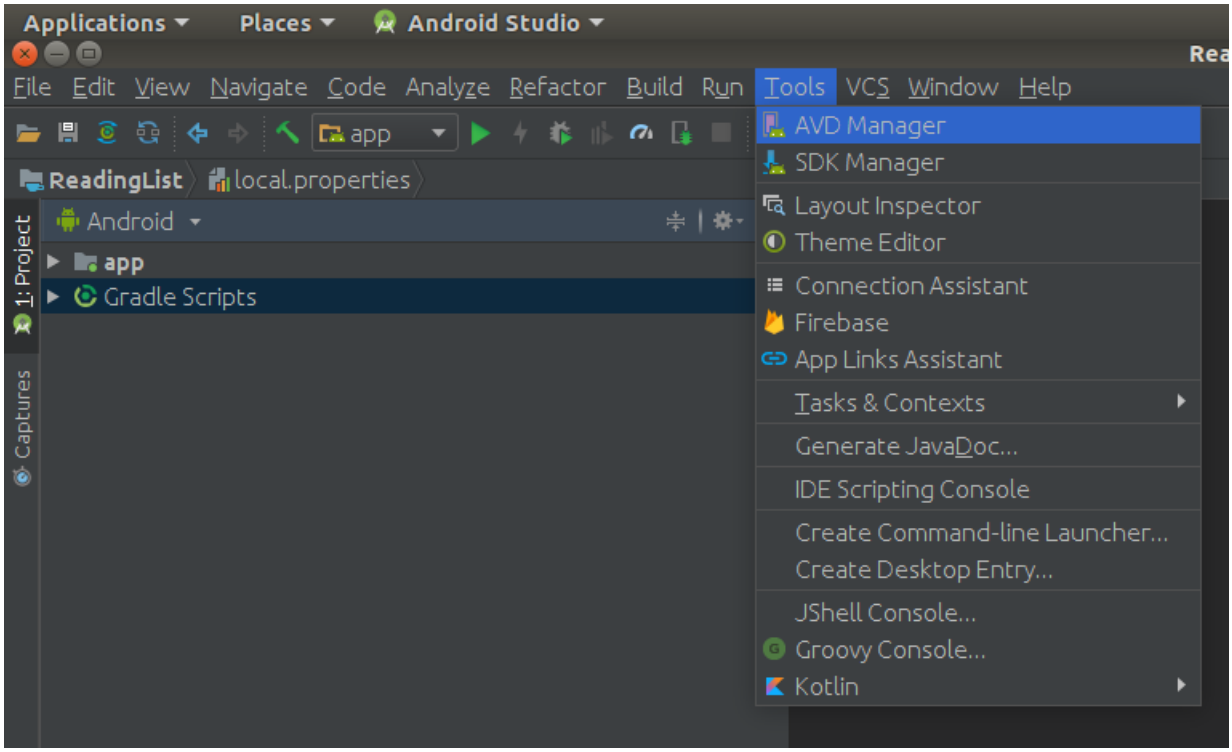
Android developers will often deploy the application they are developing to their own device. Sometimes that requires extra device drivers, extra udev rules, OEM software, et cetera.

The other common option is to deploy to an Android Emulator. While these tend to take more computing resources than attaching your own device, they are often simpler to get working. We will use an emulator for the rest of this course.

If you later wish to try with your own devices, please see this page to get information on how to do so for your platform.

Setup

We will start by launching the `AVD Manager` in the toolbar under `Tools`.

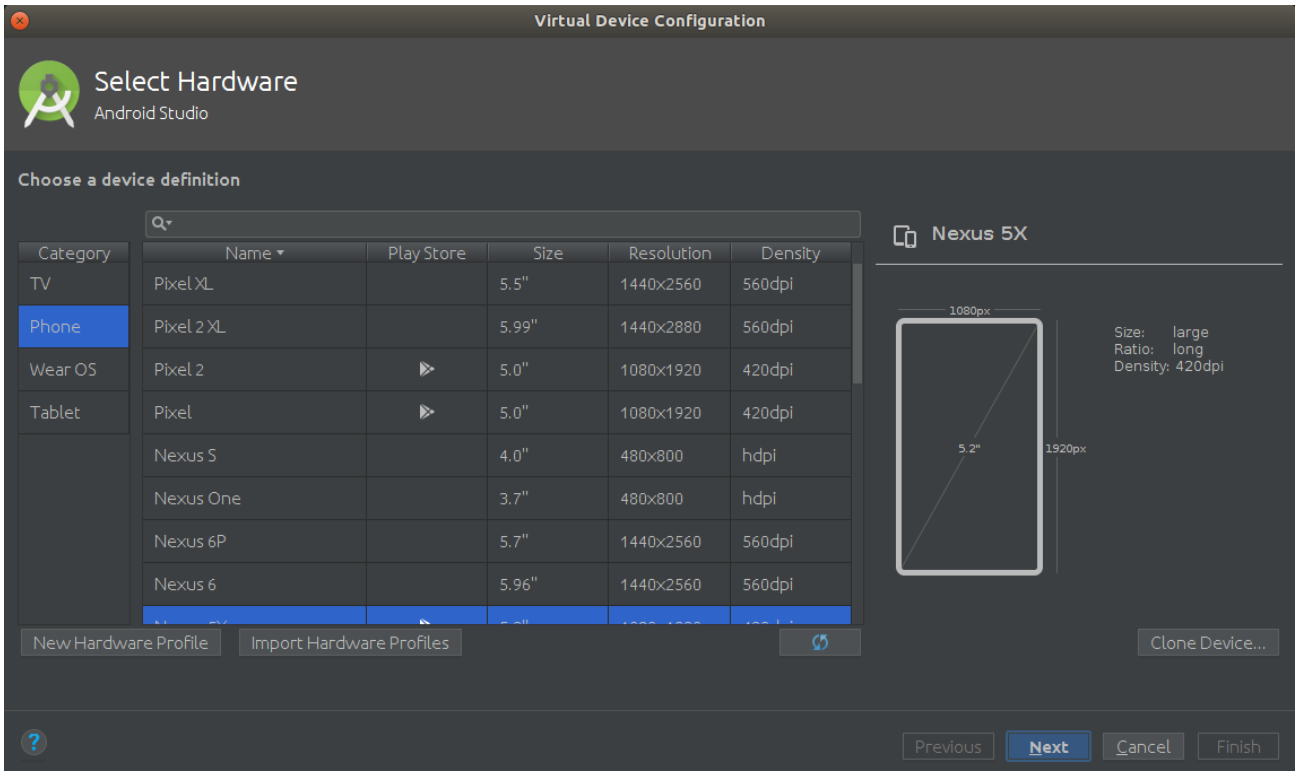


Android Virtual Device Manager

On the `Android Virtual Device Manager` dialog, you will see a list of any virtual devices you have already configured.

On the bottom left, click on `Create Virtual Device...`

Select Hardware



Category

This specifies the type of device you are going to create. They currently do not show Android Auto or Android Things on that list. Select `Phone`.

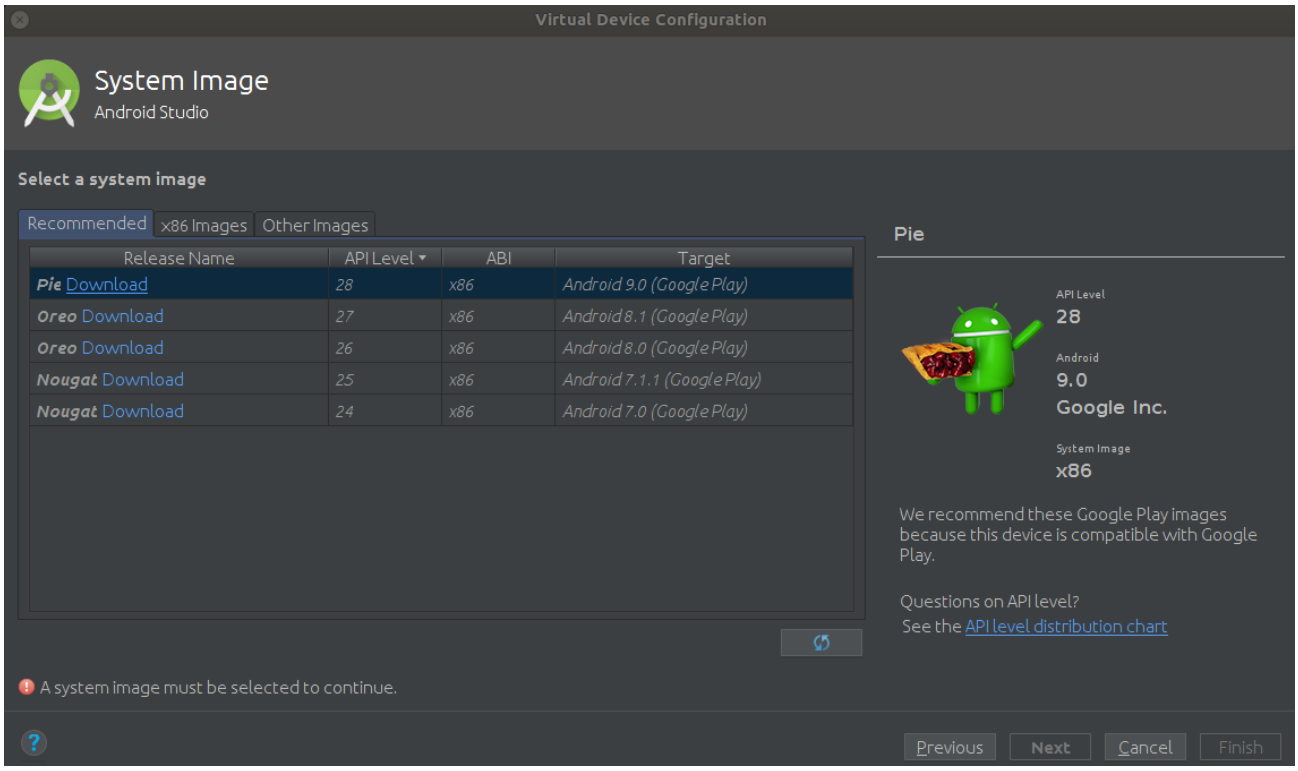
Device

The center of the screen shows a list of device specifications. You can search the list, add new ones, etc.

For now, let's select the 420dpi `Pixel 2` that shows it has Play Store support, and click `Next`.

System Image

On the next screen, you are asked to select a system image.



Google shows you the recommended system images - and they may be different on your platform than they are on mine.

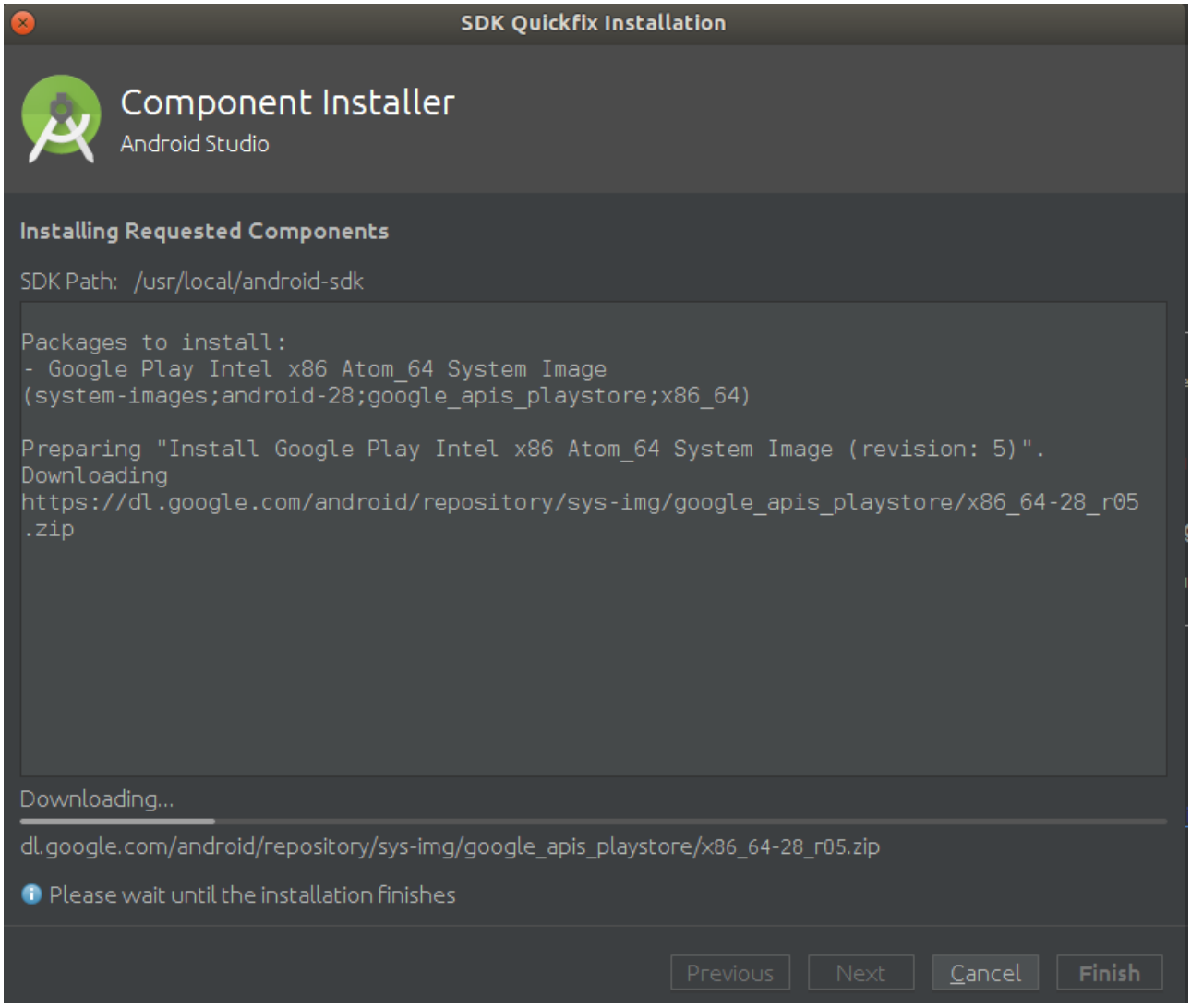
> One thing I have noticed is that even though x86_64 (64-bit) may be supported and shown under the x86 tab, it doesn't necessarily show under the Recommended tab. This is unfortunate since the x86_64 images will perform better on platforms that support it.
>
> If your system is not an x86 platform, the x86 images do not add any performance benefit to you.

Select a `Pie` / `API Level 28` / `Android 9.0 (Google Play)` option available to you. If you wish to select the `x86 Images` tab to find one that is `x86_64` you may do so - just make sure the rest of the options match.

> Note that there is a difference between `(Google Play)` and `(Google APIs)`

When you have chosen the one you want, click the `Download` link, if available. This step takes awhile.

> If you have previously downloaded this system image, you will skip this step.

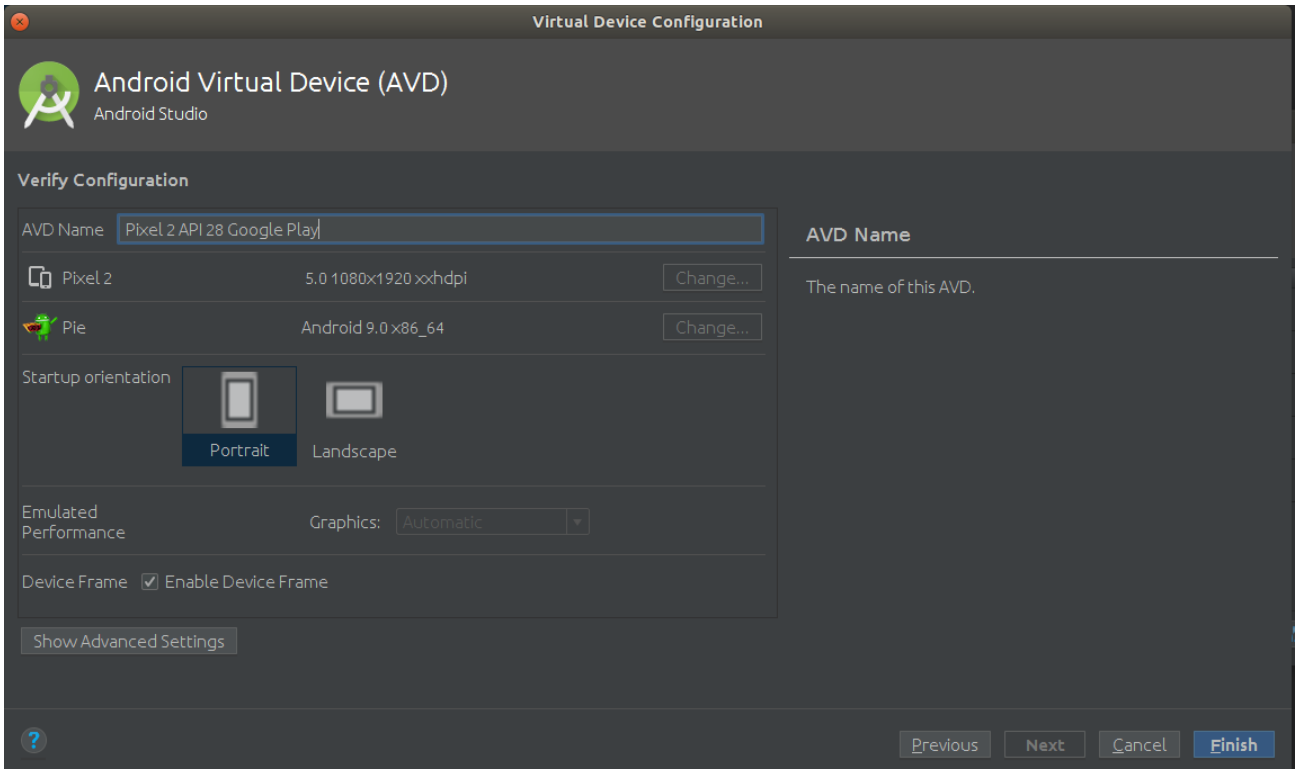


When it is done, click `Finish` and re-select the newly installed system image.

Click `Next`.

Android Virtual Device (AVD)

Now you can configure your new emulator.



AVD Name

We want something descriptive. If you end up making multiple emulators to test customer experience, you don't want to have to guess which is which.

Name it `Pixel 2 API 28 Google Play`

Density

Take note that just below the name it tells you the screen dimensions. For us, we can see that this emulator will be a 5" display, 1080 pixels wide, 1920 pixels high and have a screen density in the `xxhdpi` bucket that we discussed earlier.

Startup Orientation

This really matters more to tablets. The native orientation for some tablets is landscape and for others portrait. The reason this matters is that a lot of routines *assume* 90° rotation is landscape - but that is not always the case.

Leave it at `Portrait`.

Device Frame

This option doesn't matter much, but will make your experience with the emulator better. Leave it enabled.

Advanced Settings

Click the `Show Advanced Settings`.

We are not going to change the Camera or Network settings here, but feel free to look at them.

Emulated Performance

This is a tricky one. The default setting is to `Quick Boot`, which, in theory, makes the emulator boot up faster.

Unfortunately, it often prevents the emulator from booting at all. For the duration of this class, set it to `Cold Boot`. We would rather it start up correctly than take a few seconds less.

Memory and Storage

It is very possible you will change these options when working on your own application. You may create custom emulators with low memory or really large storage to test different scenarios.

For now, leave them at the defaults.

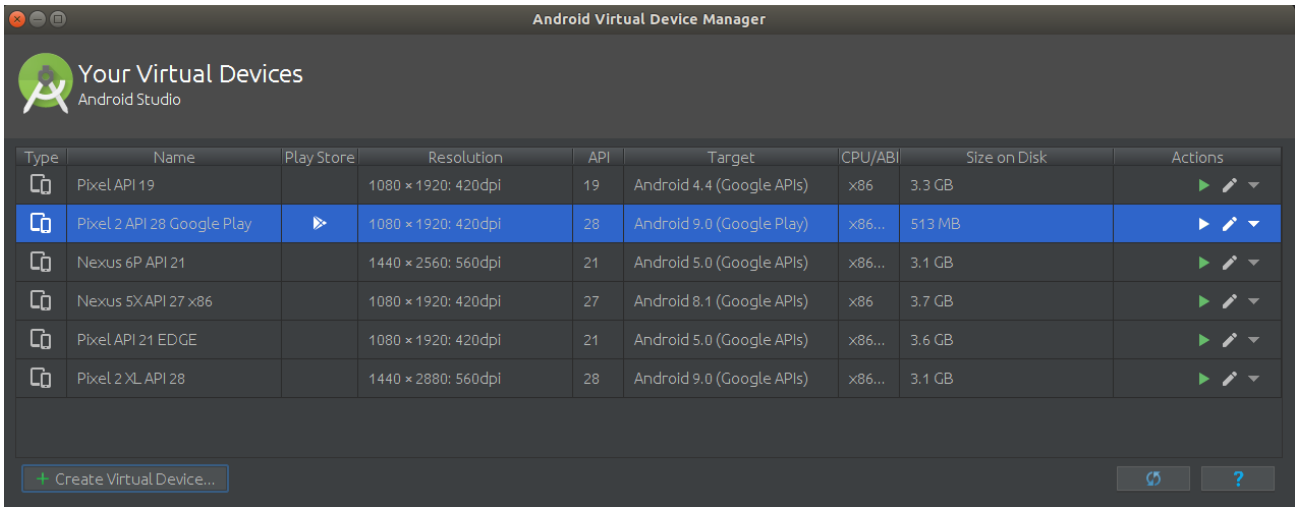
Keyboard

When it works, the keyboard input can make your life much easier. It allows you to use your computer keyboard to enter text on the emulator rather than using the mouse with the virtual keyboard. Sometimes it doesn't work -- but leave it enabled and hope for the best.

Once you are satisfied with everything, click `Finish`.

Starting the new Emulator

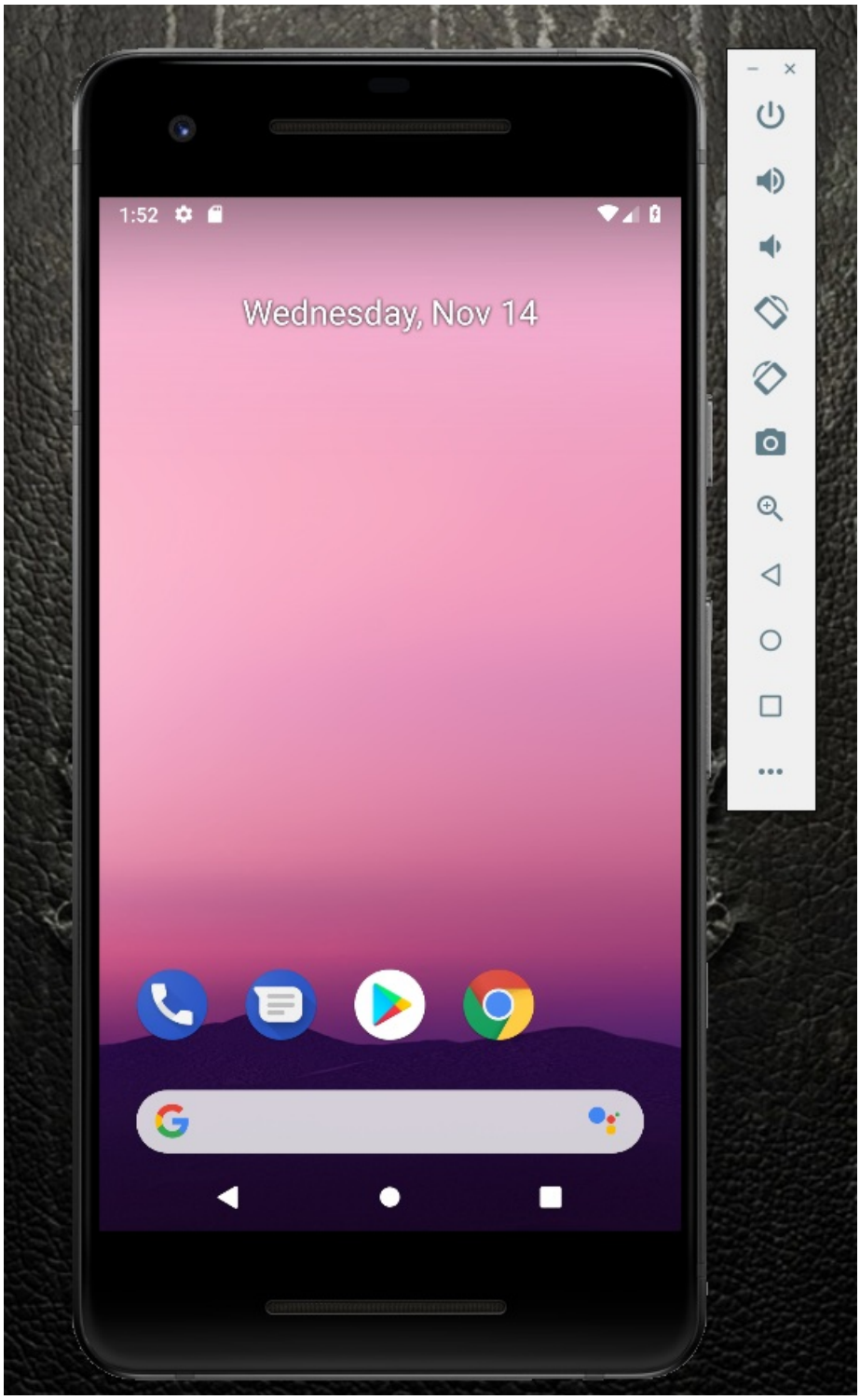
You should now see your new `Pixel 2 API 28 Google Play` AVD listed.



By default, if the emulator is not running, the build can start the emulator when it deploys your application. Once you are done testing, it can then shut your emulator down again.

Rather than waste a lot of time doing that, it's much convenient to start the emulator once and leave it running. If your machine has low memory, that may not be an option - but we will try it now so that you know how to do it.

On the line containing your new emulator, click the play button on the far right next to the edit pencil.

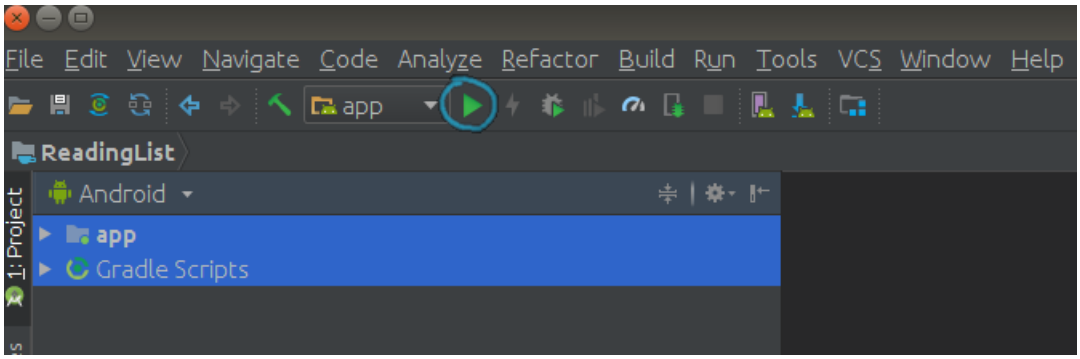


With your emulator now running, you can close the AVD manager and continue to use the emulator and IDE at the same time.

Deploy

So what's the first thing we want to do with our new emulator? How about deploy our auto-generated project?

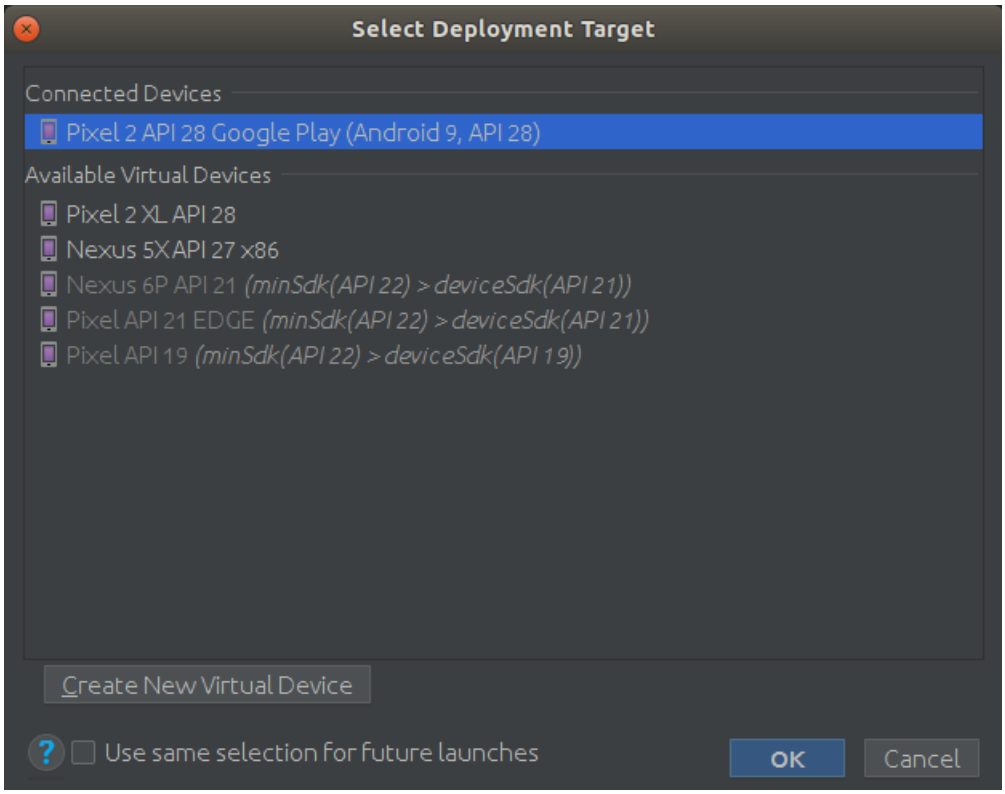
Back in the IDE, look at the toolbar.



Next to the selector that says `app` you will see a green play button. Click it.

Select Deployment Target

You will be prompted to select which device you would like to deploy your application to.

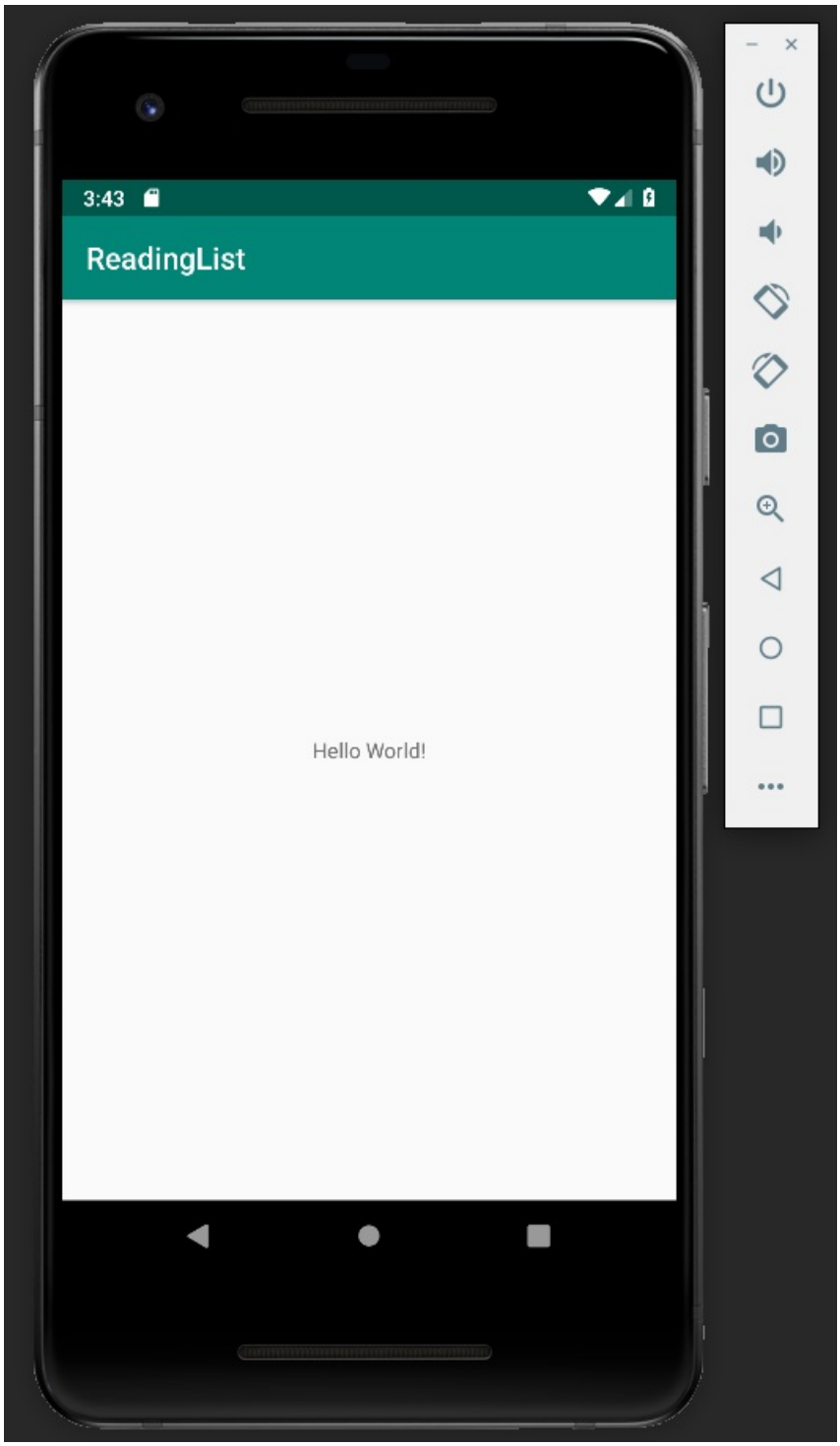


If the emulator is still running, it will show up as a `Connected Device`.

If you have a connected Android device (with proper udev and/or drivers), it will also show up as a `Connected Device`.

If neither of those are the case, your emulator will show up under `Available Virtual Devices`. In this case, it will start the emulator in order to deploy to it.

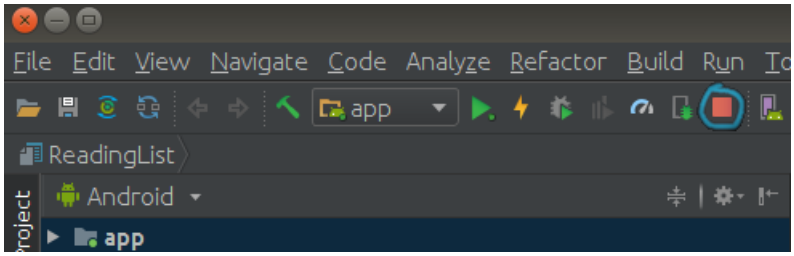
Select `Pixel 2 API 28 Google Play` and click `OK`.



Stop the Application

Now that we have seen our Hello World, let's go ahead and stop it.

Back in the IDE, click on the red stop button in the toolbar.



You might notice the lightning bolt that wasn't there before. That allows us to update the running application without stopping it first. We're not ready for that yet.

Tweak the UI

Next, we will tweak a few of the files we looked at earlier to understand how they impact your application.

Layouts

We're going to start with the Layout. Open your `app/res/layout/activity_book_list.xml` in the editor.

As mentioned above, it currently looks like

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".BookListActivity">

    <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello World!"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

</android.support.constraint.ConstraintLayout>
```

Let's make a couple small changes to it.

Modify the ConstraintLayout

The ConstraintLayout represents the background container of this page of our application. Right now, that is a stark white background. Let's soften it a little.

In between the `android:layout_height` and `tools:context` parameters, let's add a new background color.

```
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#fff8dc"
        tools:context=".BookListActivity">
```

The hex value `fff8dc` represents the HTML color `cornsilk`.

Once you have made that change, redeploy your application and check the results.

Modify the TextView

The background isn't as bright now, but the text is a bit harder to see.

textStyle

Inside the `TextView` tag, after the `android:text="Hello World!"` line, hit enter to create a new line. Type in `android:` and notice that you are prompted with the avaialble properties. Choose the `textStyle` one and notice that you are now prompted with the possible values. Choose `bold`.

It should now look like

```
<TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>
```

Deploy your application and see the results.

textSize

It's a little better, but can we make the text larger?

In the same way that you did textStyle, add a textSize.

Note that this time it only prompted you with an `@android` value. Here we will have to improvise. Use `24sp` as the value.

> You might recall from earlier that we discussed different screen densities?
>
> When we are specifying sizes in Android, we never use pixel values; which can seem counter-intuitive. Instead, Android uses something call `dp` or Device-Independent Pixels to represent a similar UI size, regardless of the phone screen size. Similarly, Android also uses `sp`, or Scalable Pixels for text specifically so that the text size also changes if the user modifies their accessibility settings.
>
> Long story short, use `dp` for layouts and `sp` for text.
>
> For more information on screen densities, see the Android Developer site.

You should have added:

```
android:textSize="24sp"
```

Deploy your application and see the results.

Add a UI element

Let's start by grabbing a public domain image and saving it locally. Download this image and save it into your `app/src/main/res/drawable/` folder as `earth.png`. Remember that Android is case-sensitive.

Below the TextView, but still inside the ConstraintLayout (remember that we can only have one top-level container), start typing `<Ima` and choose `ImageView`. When prompted for the sizing, choose `wrap_content` for both the width and the height.

Here's what we have so far.

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

Add another attribute to the ImageView. As you type `android:` you will see `src` as an option. Select that and you will see `@drawable/earth` as an option. That is the image you just saved. Select it.

If you deploy your application now, what do you see?

Adjusting the layout

The Earth shows up, but it takes the whole screen. Where is your text?

If you look at your TextView, you will see that it is constrained to all edges of the parent. What about the ImageView? It has no constraints, so it is overlapping it and taking over the whole screen. You can see this by clicking on the `Design` tab at the bottom of the screen in the IDE.

Go back to the `Text` view and let's fix it.

Let's start by copying the 4 constraints from the TextView into the ImageView.

Next, we'll want to tell the TextView to be above the ImageView; and tell the ImageView to be below the TextView. To do this, we need to give them each an identifier.

In the TextView, add an attribute `android:id="@+id/hello_text"`. In the ImageView, add `android:id="@+id/earth_image"`.

Now we can specify relationships between them.

In the TextView, replace `app:layout_constraintBottom_toBottomOf="parent"` with `app:layout_constraintBottom_toTopOf="@+id/earth_image"`.

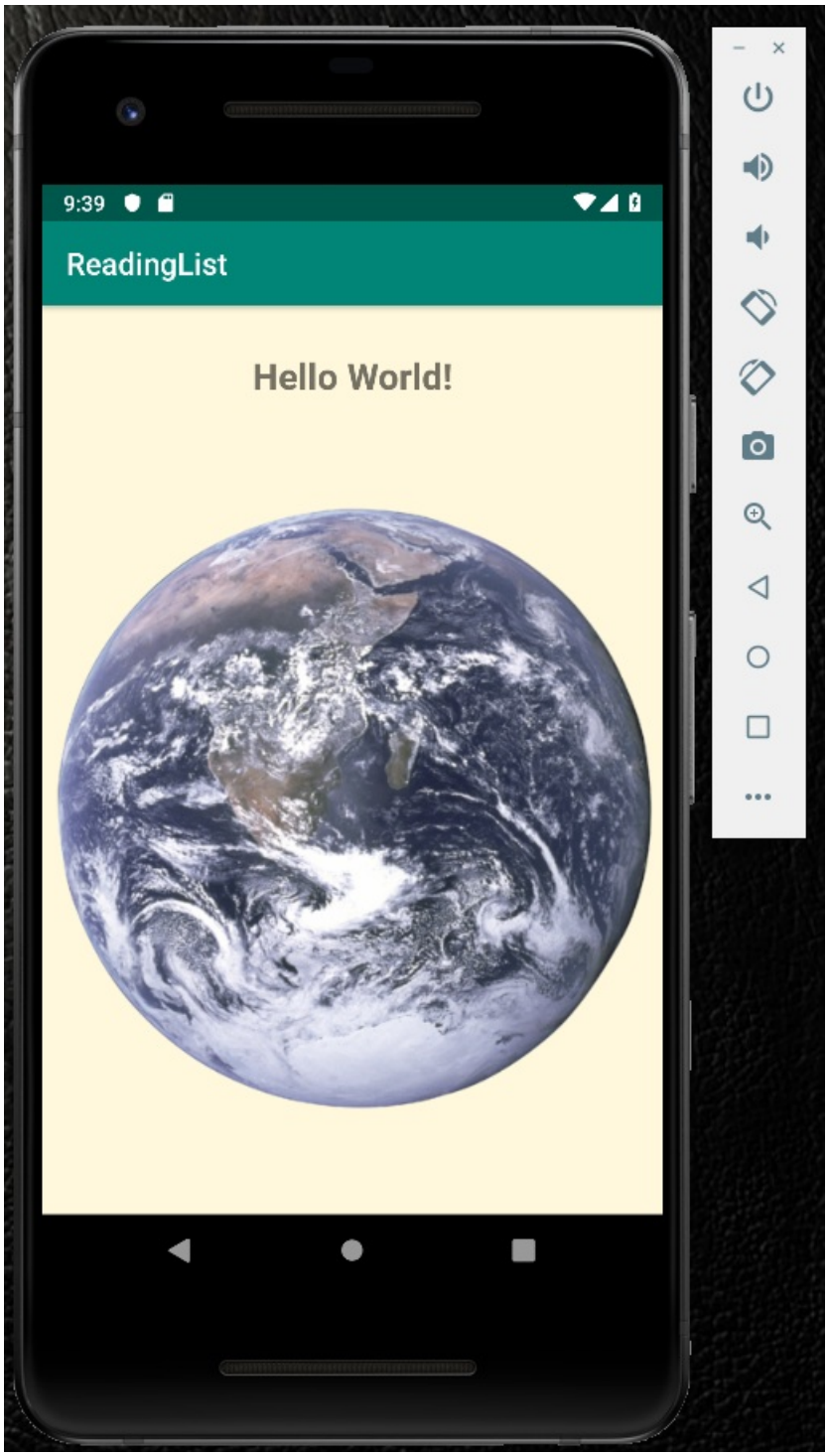
In the ImageView, replace `app:layout_constraintTop_toTopOf="parent"` with `app:layout_constraintTop_toBottomOf="@+id/hello_text"`

End result is something like this

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#fff8dc"
        tools:context=".BookListActivity">

    <TextView
        android:id="@+id/hello_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:textStyle="bold"
        android:textSize="24sp"
        app:layout_constraintBottom_toTopOf="@+id/earth_image"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

    <ImageView
        android:id="@+id/earth_image"
        android:src="@drawable/earth"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/hello_text" />

</android.support.constraint.ConstraintLayout>
```

Take a look at the Design tab and you should see that "Hello World" is now above the image. Switch back to the Text tab.

Deploy your application.



Strings

Keep your `activity_book_list.xml` and open `app/res/values/strings.xml` next to it.

You might remember that we discussed earlier that we should avoid hardcoding values where possible? In our layout, we are currently hardcoding the text `"Hello World!"`. Let's move that to our `strings.xml`.

In your `strings.xml` file, in between the two `resources` tags (remember, one top-level container), let's add another `<string` tag like the one used for your `app_name`. Let's call this one `hello`. When you type `<st` it will prompt you to complete it as a `string`. If you do so, it will put the caret inside the quotes for you to type `hello`. You will notice that it is surrounded by a red box. Hit the tab character on your keyboard to tell it you are done, then type `>` for it to auto-complete the end-tag. It's now expecting you to enter the value. You can type it or copy it from the layout.

```
<resources>
    <string name="app_name">ReadingList</string>
    <string name="hello">Hello World!</string>
</resources>
```

To tell the layout to use this new string resource, replace `android:text="Hello World!"` with `android:text="@string/hello"`. You will notice that auto-completion is there as well.

> You might be asking - aren't we still hardcoding it? We are, but you can provide variations of this file for different versions of your application, different languages, different countries, et cetera -- without needing to change any code or layouts.

Deploy your application to make sure it is still working.

Still looks OK, but how can we be sure?

Let's change the text in the `strings.xml`.

```
<resources>
    <string name="app_name">ReadingList</string>
    <string name="hello">Hello, Earth!</string>
</resources>
```

Re-deploy and verify.

> Any time you writing hard-coded text in an layout or source file, consider putting it inside the `strings.xml` instead.

Colors

Similarly, we should not be hardcoding hex color values anywhere. Let's fix that.

Open your layout file and your `colors.xml` file.

We are going to replace the background color in the ConstraintLayout with an `@color` reference, the same way we did with the strings.

Before continuing, see if you can figure it out. Remember that the hex value we used was for the color `cornsilk`.

As a reminder, you can always use the W3C to match up hex color values with common names.

Were you able to figure it out?

Inside the `colors.xml` file, we add a new `color` tag with our hex value and name.

```
<color name="cornsilk">#fff8dc</color>
```

Inside our layout, we replace our hardcoded value `android:background="#fff8dc"` with the new color reference `android:background="@color/cornsilk"`.

This same pattern is used through most of Androids' resources.

Deploy the application and verify that everything still looks the same.

Built-in colors

Of course, we want to make sure there was a change...

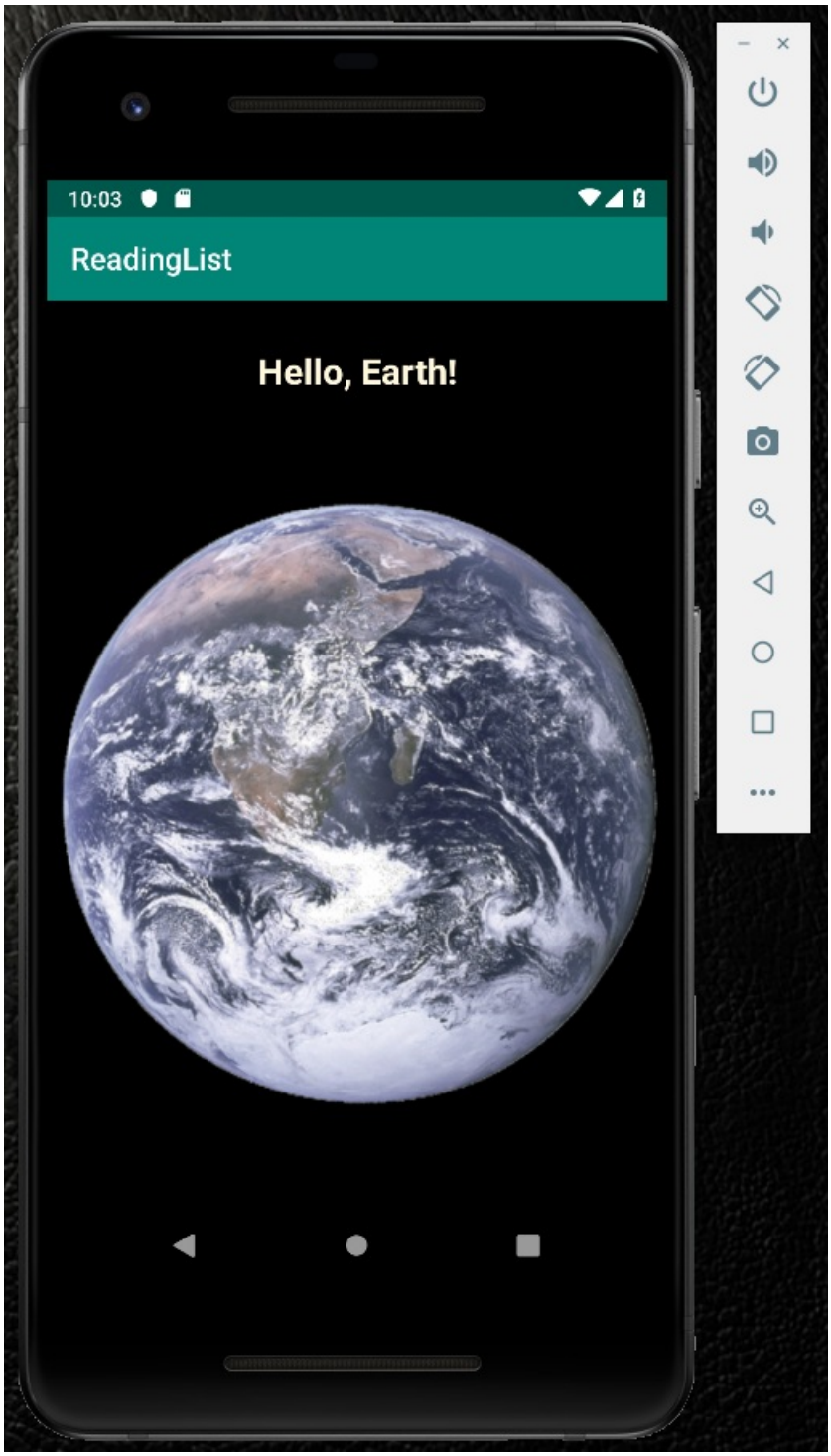
Android has some built-in colors you can reference. You can reference them with `@android:color/` instead of just `@color/`. Let's try that now. Change the ConstraintLayout background to `@android:color/black`.

Redeploy.

The background changed, but now you can't see your text.

In the TextView, add a new attribute. Let's use your newly defined cornsilk for the text. `android:textColor="@color/cornsilk"`.

Redeploy.



Styles

When you start having multiple layouts doing the same thing over and over, you will start realizing that even the styling of those layouts can feel like hardcoding. That's especially true if you are always setting the same text size, color, styling, font, etc... on every single text view.

To combat that, we follow the same pattern that we just learned about and move settings into styles.

Let's look at your layout file and your `styles.xml`.

Styles are a little different in that they often specify a parent that they are inheriting from. We won't worry about that for now, and will address it as it comes up.

For now, let's move the styling for our TextView. You will want a unique name that specifies what it is and would make sense across the entire application. For now, we will call it `LightTitle`.

Inside the `styles.xml` file, inside the main `resources` tag, we will add another `style` tag named `LightTitle`, with no parent.

So far, you should have this:

```
<style name="LightTitle">

</style>
```

> It is somewhat arbitrary what should be in the layout and what should be in the style. Each team might have their own policies. To keep it simple, we will assume *styling* goes in styles and *laying out* goes in the layout.

Looking at our current TextView, we have

```
<TextView
    android:id="@+id/hello_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    android:textStyle="bold"
    android:textSize="24sp"
    android:textColor="@color/cornsilk"
    app:layout_constraintBottom_toTopOf="@+id/earth_image"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>
```

- The `id` must stay at the layout.
- The `text` is specific to this instance, so should stay.
- The `layout_constraint*` are specific to how it is positioned on the screen, so that should stay.
- The `textStyle` is a generic style for our LightTitle, so should be moved.
- The `textSize` is should also move.
- If our style was just `Title`, we could justify keeping the `textColor` in the layout. Since we are calling it `LightStyle`, then the intention is to include the color as well. We will move it.
- What about the `layout_width` and `layout_height`. This one is an interesting one. On one hand you could argue that LightTitle should (or should not) take up the entire width of the screen, as a matter of styling. On the other hand, it has the word layout right in the attribute. This is one that developers often disagree about. We often tend to put it into the styles because we copy/paste it everywhere; but it is specific to the type of layout that holds it. We'll leave them for now.

Ok, so let's move `textStyle`, `textSize` and `textColor`.

In your new style, add the following attributes to match what we already have in the layout:

```
<style name="LightTitle">
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">24sp</item>
    <item name="android:textColor">@color/cornsilk</item>
</style>
```

The name of each item matches the parameter name in the layout. The values are the same, but without the wrapping quotes.

In your layout, add `style="@style/LightTitle"` to your TextView then remove the `textStyle`, `textSize` and `textColor` attributes.

Save and redeploy.

Ok, everything should still be working. Anytime you want to make a title with those three parameters, you can just specify the style instead.

Style Parents

Let's talk about the style parents.

Remember how we said that the `textColor` was the only reason our style was a `LightTitle`?

Add a new style above it called `MyTitle` (creative, right?).

Move the `textSize` and `textStyle` from `LightTitle` to `MyTitle`.

Now, add `parent="MyTitle"` to `LightTitle`.

End result should look something like

```
<style name="MyTitle">
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">24sp</item>
</style>

<style name="LightTitle" parent="MyTitle">
    <item name="android:textColor">@color/cornsilk</item>
</style>
```

You now have two styles. One generic title style, and one specifically light colored. The `LightTitle` inherits the attributes from `MyTitle`. In this way you can organize your styles so that a simple change has application-wide impact.

Redeploy your application and confirm everything is still working.

#

- Current Repo: v1-Basic-UI
- Continue to Data -->

Data.md

Goal

Learn how to load, save and convert data.

Table of Contents

- buildConfigField and BuildConfig.java
    - int
    - boolean
    - String
    - rootProject.ext
- SharedPreferences
    - Migrate to AndroidX
    - Add Android KTX
    - Editing with KTX
    - Reading from SharedPrefs
- Kotlin data classes and Gson
    - Asset Directory
    - Download Sample Data
    - Creating a Model
    - Setup Gson

    - Deserializing
    - Serializing
- Note about Databases

Data is a necessary part of any application. We will show you a few ways you can store and retrieve data on Android.

buildConfigField and BuildConfig.java

In your IDE menu bar, click on `Navigate` and notice the shortcut for `Class...`. For me it is currently `Ctrl+N`. For a Mac, it is `Command+O`. It will be different for various platforms, and can also be changed by you in your `File | Settings | Keymap`. While you can use the menu to open this dialog, it is worth learning as you will use it a lot.

> You can also find a list of the default shortcuts here.

Open that dialog now.

Search for `BuildConfig` and open the found class.

> It's important to click inside the search field. If it is not in focus, you could accidentally edit your source code.

```
/**
 * Automatically generated file. DO NOT MODIFY
 */
package com.aboutobjects.curriculum.readinglist;

public final class BuildConfig {
  public static final boolean DEBUG = Boolean.parseBoolean("true");
  public static final String APPLICATION_ID = "com.aboutobjects.curriculum.readinglist";
  public static final String BUILD_TYPE = "debug";
  public static final String FLAVOR = "";
  public static final int VERSION_CODE = 1;
  public static final String VERSION_NAME = "1.0";
}
```

> You will note that even though we selected a Kotlin project, this file was still created as a Java class.

This class has some constants for things like your application package, version and whether it is the debug build.

You will also notice that it tells you not to modify it. So how do you add new values to it?

Open your module-level `build.gradle`. That's the one that says `build.gradle (Module: app)`.

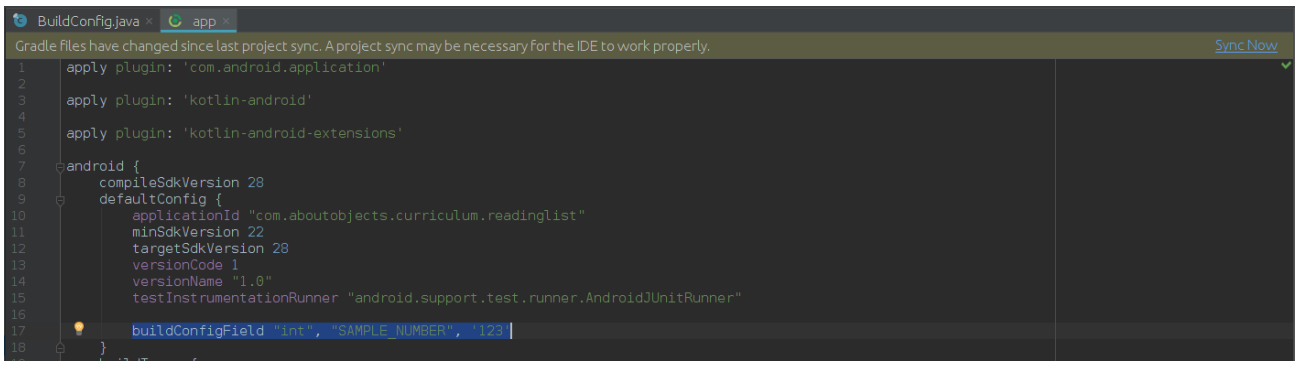
int

First, let's look at how to store a number. Inside the `android` / `defaultConfig` closure, add this line:

```
buildConfigField "int", "SAMPLE_NUMBER", '123'
```

This is specifying that we want an int named `SAMPLE_NUMBER` with a value of 123.

Once you make this change, the IDE will tell you that a sync is necessary.



This is because you have changed the rules of the build and it needs to regenerate files. The reason this wasn't necessary earlier is because you were only modifying resources and not the build script itself.

Click the `Sync Now` button.

> It is possible to automatically sync whenever a change is made. Unfortunately, this has the side effect on constantly syncing and slowing you down while you are making a lot of changes to those files.

Once the sync has finished, go back to your `BuildConfig.java` and notice the new entries.

```
// Fields from default config.
public static final int SAMPLE_NUMBER = 123;
```

> You might be wondering why the number has to be wrapped in single-quotes. Try removing them in your `build.gradle` and re-syncing.
>
> Once you have completed that test, change the code back and re-sync.

boolean

What about true/false instead of a number?

In the same way, add another entry to your `build.gradle`:

```
buildConfigField "boolean", "SAMPLE_BOOLEAN", 'true'
```

Once you sync, the new value will show up in your `BuildConfig`.

> You might be asking why the value `'true'` instead of just `true`. Try it.

String

That's all well and good, but you would like something a bit more complex right? Maybe a string? We've got you covered.

```
buildConfigField "String", "SAMPLE_STRING", '"This is a string"'
```

Sync and verify.

> Now wait a sec, why the double-quotes inside the single-quotes? Try with just double quotes and look at your BuildConfig results.
>
> Unlike the other tests, the auto-generation completes; but the code it generates would not be viable.

rootProject.ext

Gradle has the ability to specify variables in the project-level `build.gradle`. How would you then use them inside the module-level `build.gradle` buildConfigField?

In your project-level `build.gradle`, let's add a global variable. That's the one that says `build.gradle (Project: ReadingList)`.

> If you checked out our git repository, it would read `build.gradle (Project: __name_of_git_directory__)`.

After `buildscript` but before `allprojects` add a new `project.ext` closure at the same level as those two, like so:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.10'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.2.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

project.ext {
    sampleVariable = "This is a sample variable"
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Now, we want to reference that `sampleVariable`. Back in your module-level `build.gradle`, add another buildConfigField:

```
buildConfigField "String", "SAMPLE_VARIABLE", "\"${rootProject.ext.sampleVariable}\""
```

> You might have noticed that the IDE wanted to sync when you modified the first file. Since we do not have it set to auto-sync, we were able to edit both files before syncing.

Sync and look at your `BuildConfig`.

> You are most likely wondering why we used double-quotes on the outside instead of single-quotes and then escaped the inside one. Try using `'"${rootProject.ext.sampleVariable}"'` instead and see what happens.
>
> The single-quotation marks prevent the variable from being processed, thus treating the `${` as a string literal instead of as instructions.

Before moving on, your `BuildConfig.java` should have these entries:

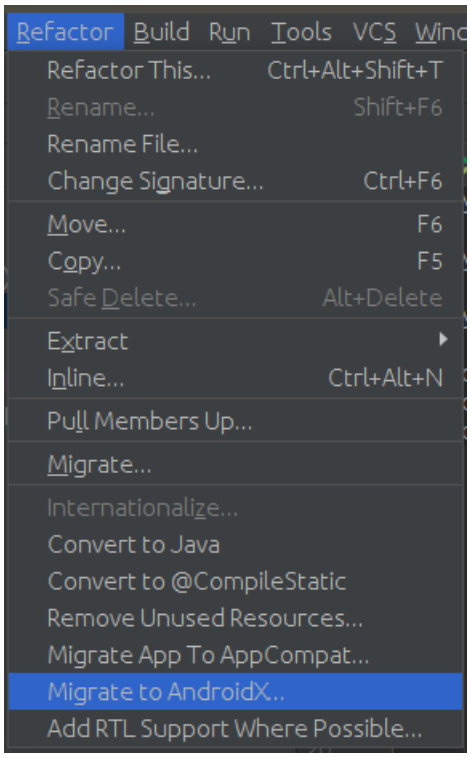
```
// Fields from default config.
public static final boolean SAMPLE_BOOLEAN = true;
public static final int SAMPLE_NUMBER = 123;
public static final String SAMPLE_STRING = "This is a string";
public static final String SAMPLE_VARIABLE = "This is a sample variable";
```

SharedPreferences

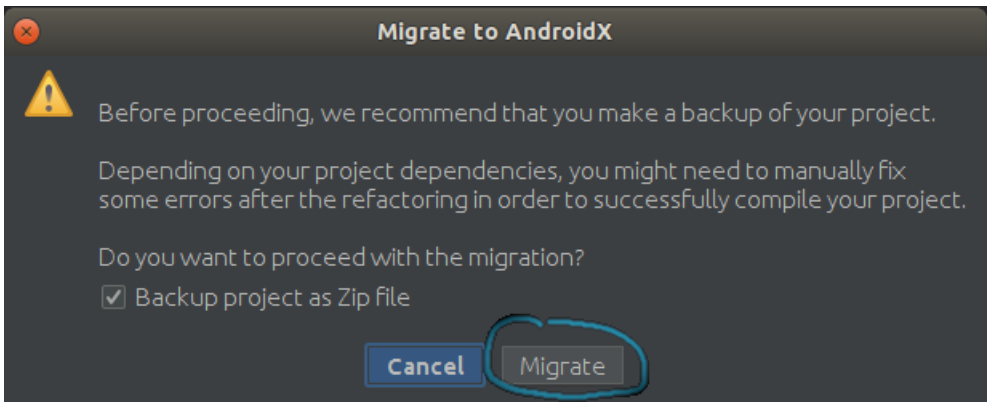
Google recently introduced the Android KTX (Kotlin Extensions) to reduce some of the boilerplate code. Let's use that for this example.

Migrate to AndroidX

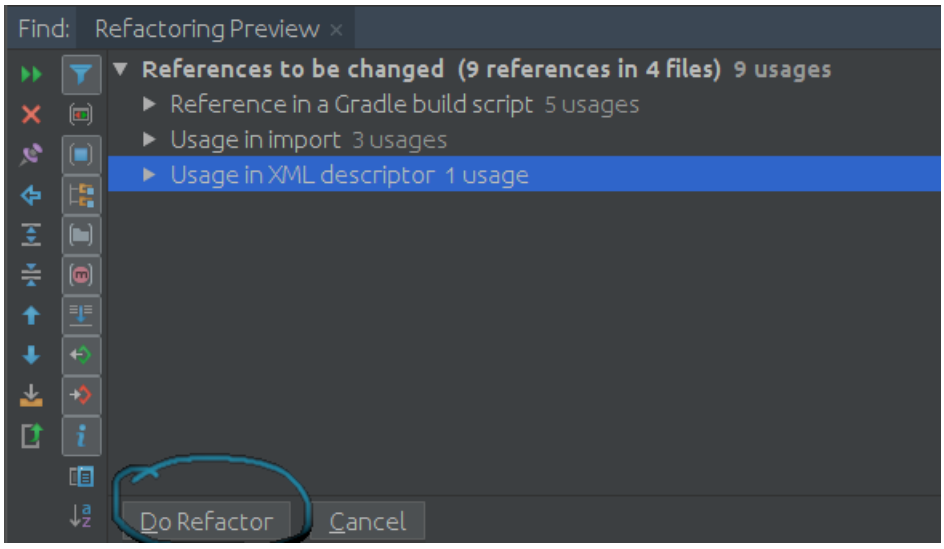
Google also recently started changing the package structure of their APIs. In order to use KTX, we need to first migrate our application to the new API structure, AndroidX. You can read more about the change here.



First, in the toolbar go to `Refactor | Migrate to AndroidX`.



It will give you a warning. Select `Migrate`



It shows you what it will do. Choose `Do Refactor`.

Add Android KTX

In the `dependencies` closure of your module-level `build.gradle`, add the KTX core dependency

```
implementation 'androidx.core:core-ktx:1.0.1'
```

You can find the latest version of the various KTX modules on the Android Developer site.

Sync and re-deploy your project.

At this point, your application should look exactly the same -- but we have added some new functionality for us to use.

Editing with KTX

SharedPreferences are often used to keep track of session or local data. It might be used to keep track of the timestamp the user last posted a picture or a webservice token for API access.

Our application doesn't really have any functionality yet, so for now let's do something simple.

Open your `BookListActivity`.

Inside the class, we'll add a `companion object`. If you are unfamiliar with Kotlin, think of this as a block of static variables and functions in Java.

```
package com.aboutobjects.curriculum.readinglist

import androidx.appcompat.app.AppCompatActivity

class BookListActivity : AppCompatActivity() {
    companion object {

    }
}
```

We'll add a couple static variables inside that block.

The `timestampPattern` comes directly from the Javadocs for `SimpleDateFormat`. Specifying `const` is like final in Java.

```
// from https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
const val timestampPattern = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
```

`KEY_TIMESTAMP` will be a unique key to be stored in our `SharedPreferences`. Often strings in Kotlin have variables embedded in them, as in this case. The `${BookListActivity::class.java.name}` portion translates to the equivalent of `BookListActivity.class.getName()` in Java. Unfortunately, Kotlin does not recognize this as something that can be a constant.

```
val KEY_TIMESTAMP = "${BookListActivity::class.java.name}::timestamp"
```

At this point, you should have something like:

```
package com.aboutobjects.curriculum.readinglist

import androidx.appcompat.app.AppCompatActivity

class BookListActivity : AppCompatActivity() {
    companion object {
        // from https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
        const val timestampPattern = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"

        val KEY_TIMESTAMP = "${BookListActivity::class.java.name}::timestamp"
    }
}
```

Now we will add some additional variables and functions inside `BookListActivity` (outside of the `companion object`).

The `timestampFormat` value is created the first time it is called because of the `by lazy` keywords.

```
private val timestampFormat: SimpleDateFormat by lazy {
    // Don't set Locale.getDefault() in companion because user may change it at runtime
    SimpleDateFormat(timestampPattern, Locale.getDefault())
}
```

> When the class does not yet have an `import` for the class you are referencing, the IDE will try to suggest possible solutions. If you click on the red highlighted word, the IDE will prompt you to select one. I tend to use `Alt+Enter` to bring up the suggestions, but it will also add a clickable lightbulb in the left gutter near the line numbers.
>
> Make sure to use the `import java.text.SimpleDateFormat` not the `icu` one.

> If you put this in the `companion object`, the IDE will complain that `Locale.getDefault()` should not be called statically.

Next, we have a function called `timestamp` that just uses the above format to convert the current time to a String and return it.

```
private fun timestamp(): String {
    return timestampFormat.format(Calendar.getInstance().time)
}
```

When we specify the `prefs` value, we are giving it the name of a file to use (`samplePrefs`.xml), and telling it what mode to use (MODE_PRIVATE). The `getSharedPreferences` method exists on the parent `AppCompatActivity`.

```
private val prefs: SharedPreferences by lazy {
    getSharedPreferences("samplePrefs", Context.MODE_PRIVATE)
}
```

Next, we are going to override a method in the parent class. Press `Ctrl+O` and select the `onCreate(savedInstanceState: Bundle?): Unit` method. Click `OK`. There are multiple `onCreate` methods; make sure you select the correct one.

With all the boilerplate out of the way, we now use the Android KTX inside the new `onCreate` to just store (with the `putString`) when the user last launched this activity.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_book_list)
    prefs.edit {
        putString(KEY_TIMESTAMP, timestamp())
    }
}
```

> The IDE may try to get you to auto-import the `android.provider.Settings.*` from the `putString` call. That is the wrong import. Click on the word `edit` in `prefs.edit` and it will prompt you to import the `androidx.core.content.edit`. Once that is completed, the `putString` method is no longer red. In general, it's a good idea to resolve imports outside-in and top-down.

The final class should look something like this:

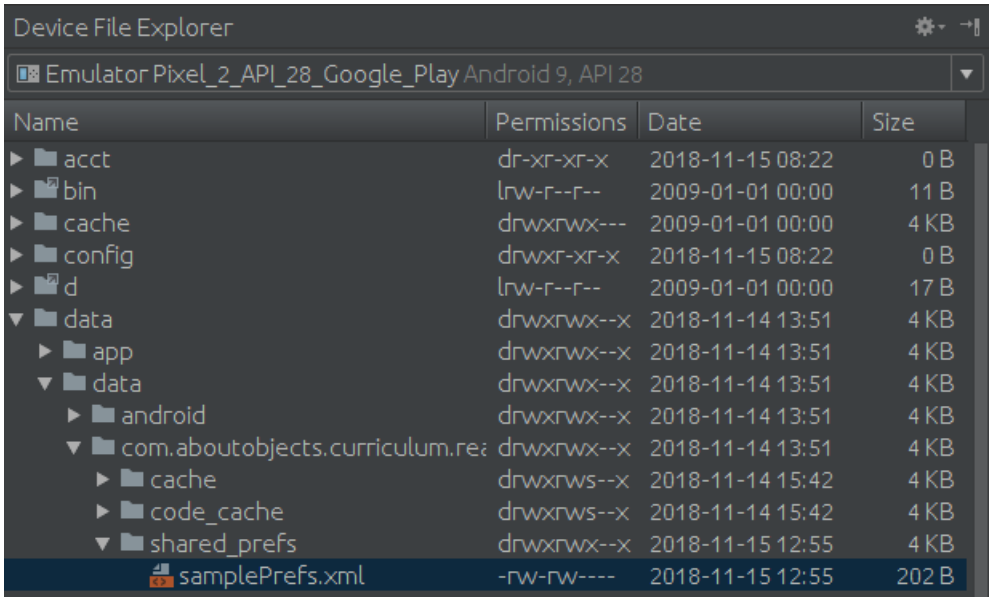
```
package com.aboutobjects.curriculum.readinglist

import android.content.Context
import android.content.SharedPreferences
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.core.content.edit
import java.text.SimpleDateFormat
import java.util.*

class BookListActivity : AppCompatActivity() {
    companion object {
        // from https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
        const val timestampPattern = "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"

        val KEY_TIMESTAMP = "${BookListActivity::class.java.name}::timestamp"
    }

    private val timestampFormat: SimpleDateFormat by lazy {
        // Don't set Locale.getDefault() in companion because user may change it at runtime
        SimpleDateFormat(timestampPattern, Locale.getDefault())
    }

    private fun timestamp(): String {
        return timestampFormat.format(Calendar.getInstance().time)
    }

    private val prefs: SharedPreferences by lazy {
        getSharedPreferences("samplePrefs", Context.MODE_PRIVATE)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_book_list)
        prefs.edit {
            putString(KEY_TIMESTAMP, timestamp())
        }
    }
}
```

Save and run your application.

So far, nothing has changed on the UI. We can still take a look at the file that was created.

On the bottom right corner of the editor, click on the tab that says `Device File Explorer`.

You are going to navigate to `data/data/com.aboutobjects.curriculum.readinglist/shared_prefs`.



Right-click on `samplePrefs.xml` and choose `Open`.

> If it fails the first time, try to Open it a second time.

It should look something like this:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="com.aboutobjects.curriculum.readinglist.BookListActivity::timestamp">2018-11-15T13
:14:47.515-08:00</string>
</map>
```

Reading from SharedPrefs

Now that we are storing some data in our SharedPreferences, maybe we can retrieve data as well?

Update the UI

Let's start by providing ourselves something to update.

strings.xml

Open your `strings.xml`. If you don't remember where it is, you can also navigate to it on the menu from `Navigate | File`.

Add a new entry:

```
<string name="last_login">Your last login was: %1$s</string>
```

> While you could just use `%s`, the build system will throw warnings if you don't include the format position.

styles.xml

Open your `styles.xml` and add a new style for our information.

```
<style name="LightInfo">
    <item name="android:textSize">12sp</item>
    <item name="android:textColor">@color/cornsilk</item>
</style>
```

activity_book_list.xml

Open your `activity_book_list.xml` and add another TextView between the title and the image.

- Give it an id of `login_text`
- Use the style `LightInfo`
- Adjust the constraints so that it is between the title and image.
- Use `tools:text` instead of `android:text` to point to our new `@string/last_login`. We aren't actually setting the text value here, just setting a temporary value to be displayed in the `Design` window.

End result should look something like

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/black"
        tools:context=".BookListActivity">

    <TextView
        android:id="@+id/hello_text"
        style="@style/LightTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        app:layout_constraintBottom_toTopOf="@+id/login_text"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

    <TextView
        android:id="@+id/login_text"
        style="@style/LightInfo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        tools:text="@string/last_login"
        app:layout_constraintBottom_toTopOf="@+id/earth_image"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/hello_text"/>

    <ImageView
        android:id="@+id/earth_image"
        android:src="@drawable/earth"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/login_text" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

BookListActivity.kt

Now we need to update the activity to set the text. Open `BookListActivity`.

Inside our `companion object`, add

```
const val displayPattern = "yyyy.MM.dd 'at' HH:mm:ss z"
```

Add a format value (not inside the `companion object`)

```
private val displayFormat: SimpleDateFormat by lazy {
    SimpleDateFormat(displayPattern, Locale.getDefault())
}
```

Then, inside the `onCreate`, before we change the preference value, let's update the UI.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_book_list)

    prefs.getString(KEY_TIMESTAMP, null)?.let {
        val time = timestampFormat.parse(it)
        findViewById<TextView>(R.id.login_text).text = resources.getString(R.string.last_login, disp
layFormat.format(time))
    }

    prefs.edit {
        putString(KEY_TIMESTAMP, timestamp())
    }
}
```

Let's walk through that change.

We ask the SharedPreferences to give us the String value associated with `KEY_TIMESTAMP`. If it has not been saved before (ie: first time the application has been launched), we want it to return `null`.

We then have the `?.let` which means that we only want to run that closure if the value returned is NOT null. So, if it successfully loaded a previously saved timestamp, it will execute that closure.

We take that returned value (`it`) and convert it to a time value using the same formatter that was used to save it -- because we know that is the format it was saved as.

We then convert that new time value to the new display format, pass that as a parameter (remember the `%1$s`) to our String resource and save it to the view associated with our layout id.

> Note: We are using findViewById here as a stepping stone. In later chapters we will be using a more efficient approach.

Save and deploy the application.

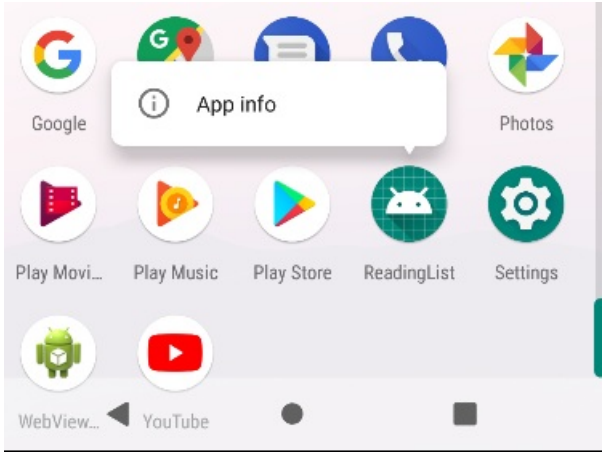
Wait a moment, then deploy again. Did the timestamp change?

> If not, the IDE may have been updating the information on the screen live. Click the stop icon, then the play icon. Did it change that time?

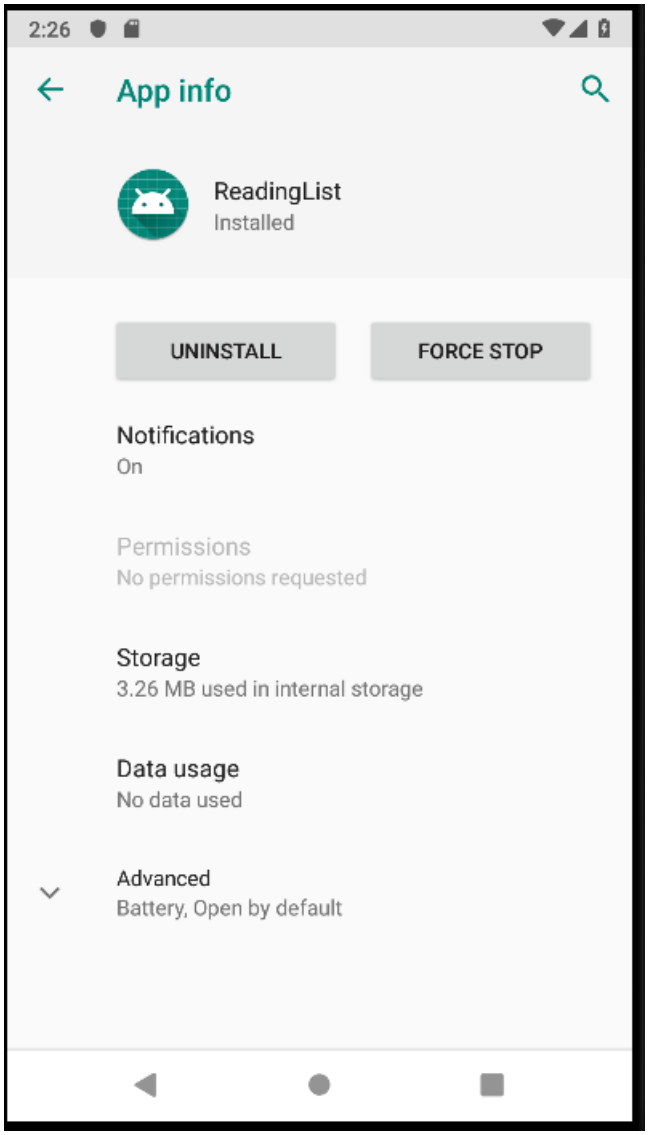
Re-validating first time use

So it is updating the last login, but how can you make sure that it works properly the first time?

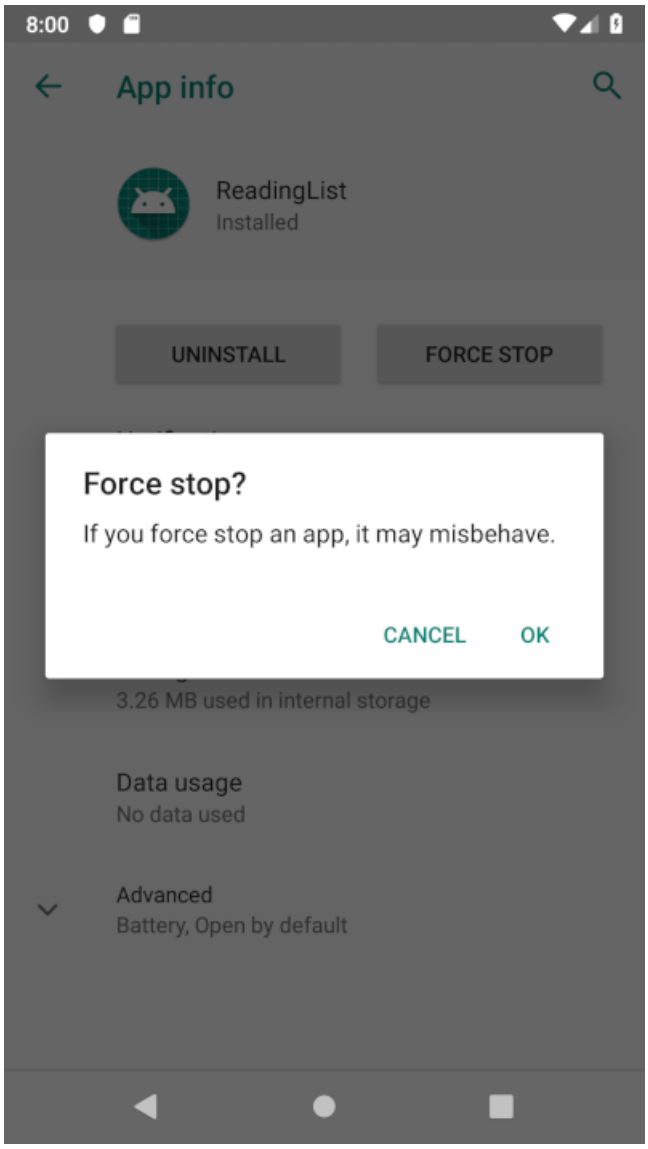
In your emulator, open the application drawer and long press on the ReadingList icon.



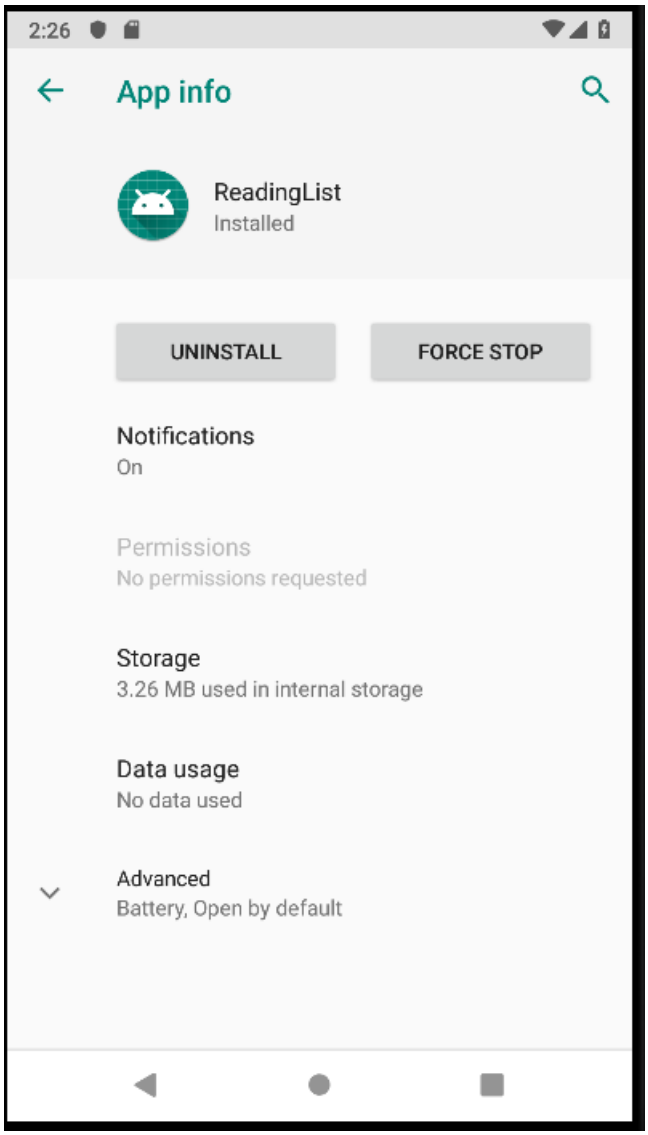
Click on `App info`.



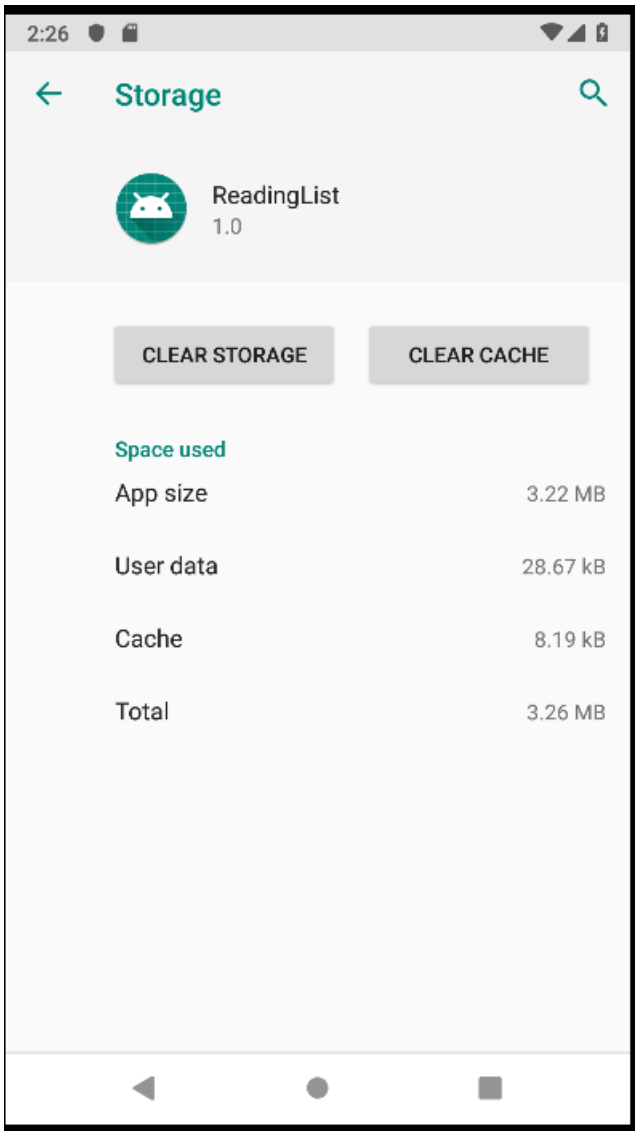
Click on `FORCE STOP`.


8:00
App info
ReadingList
Installed
UNINSTALL
FORCE STOP
Force stop?
If you force stop an app, it may misbehave.
CANCEL
OK
3.26 MB used in internal storage
Data usage
No data used
Advanced
Battery, Open by default

Click on `Ok`



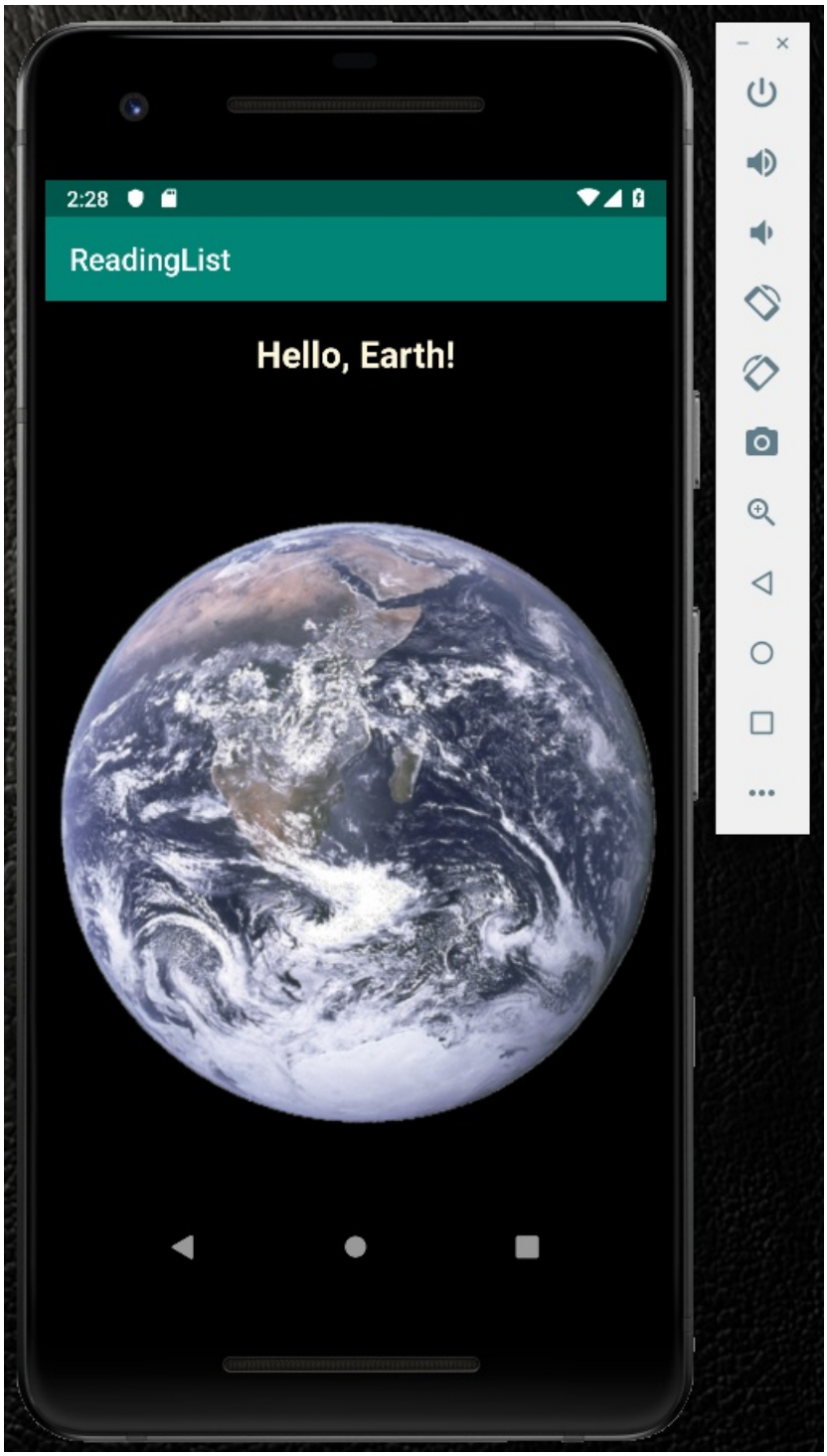
Click on `Storage`.



Click on `Clear Storage` then `OK` if prompted.

> Note: These instructions could be completely different for other phones, but the general idea should always be available somewhere in the Settings application.

Relaunch your application.



Kotlin data classes and Gson

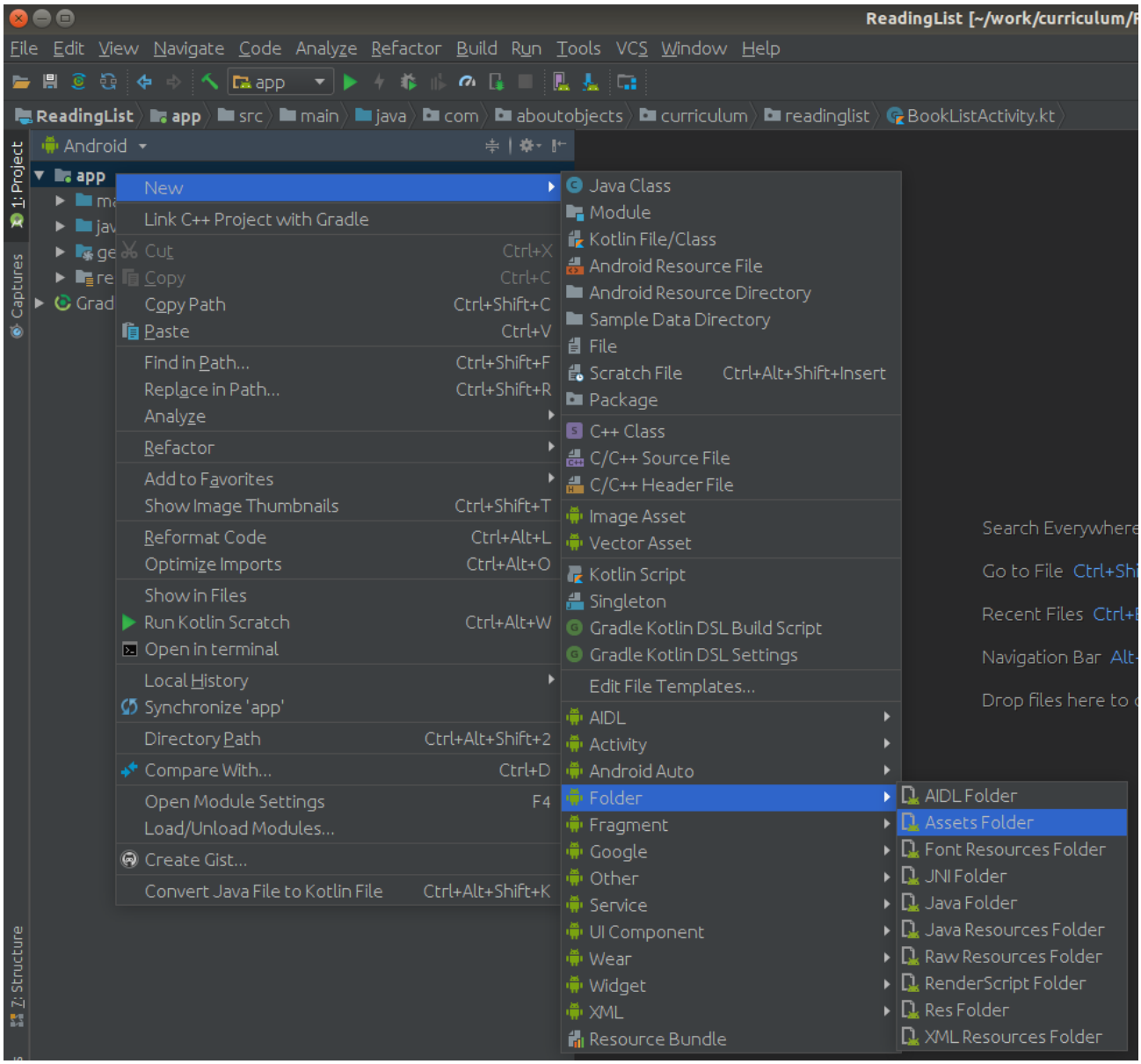
Kotlin data classes are very similar to a standard Java POJO, except that all the getters and setters are auto-generated (usually).

Gson is a popular method for reading and writing Json data.

We will generate our models using Kotlin data classes; then use Gson to serialize them to Json and de-serialize them back into models.

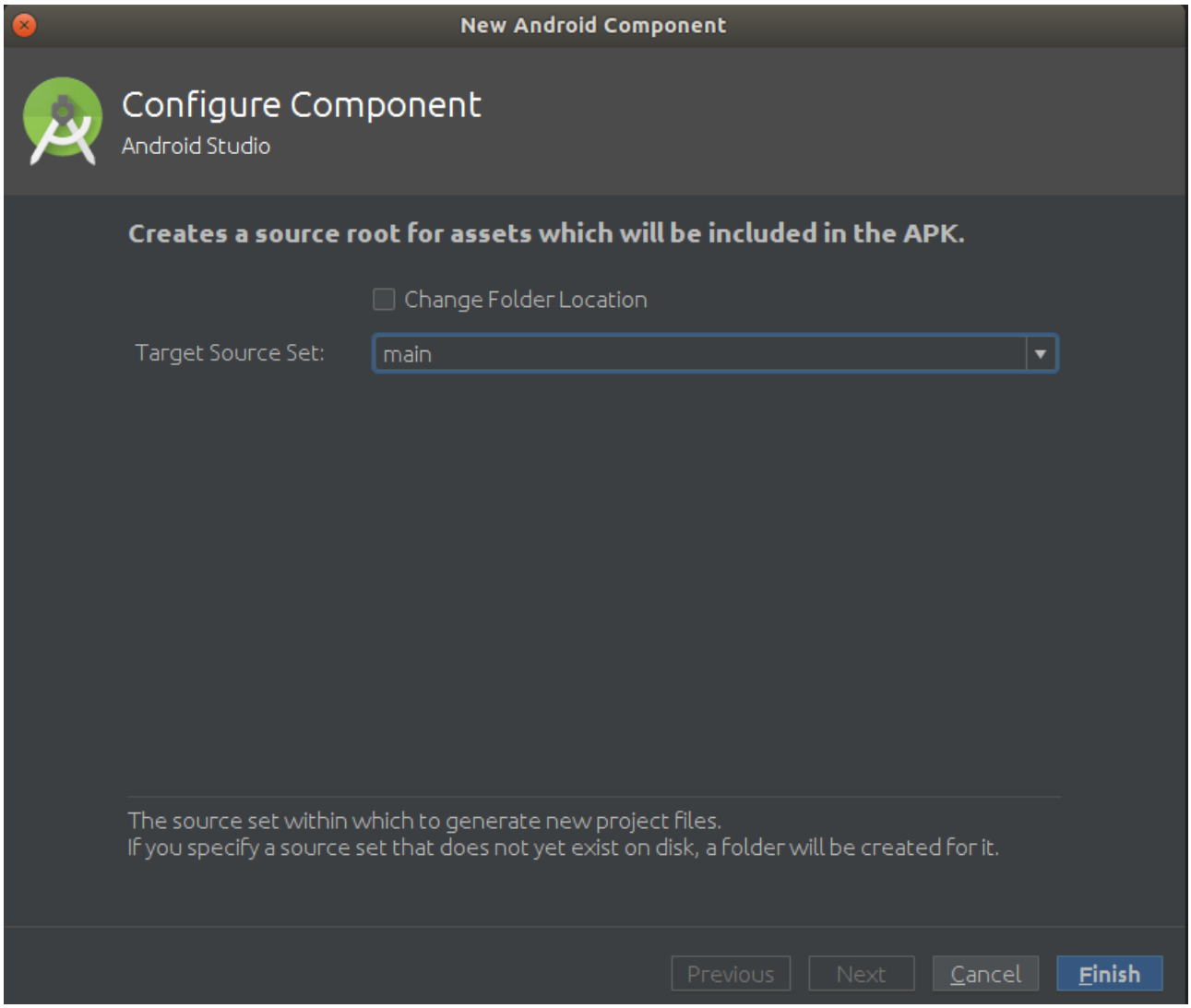
Asset Directory

First, let's create a directory to store some sample data.



Right-click on the `app` folder and select `New | Folder | Assets Folder`.

Configure Component



The next screen will give you the option of specifying that the folder is only for a particular variant/flavor (in this case your `debug` or `release` build). We are going to leave `main` selected, which means it applies to all variants/flavors.

Click `Finish`.

Download Sample Data

Now that we have a directory to save a file into, let's grab some data.

Save this file into `app/src/main/assets`.

Creating a Model

Open the new file in the IDE. You'll find it inside a new `app/assets` category. This will provide the basis for our model.

We can see that the Json has a "title" and an array called "books".

Each book in the array has a "title", "author" and "year".

The author further has a "firstName" and a "lastName".

Creating a new package

Let's start by creating a new package to hold our models and keep everything organized.

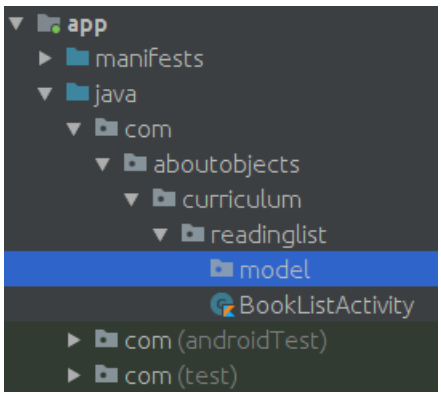
Open `app/java/com/aboutobjects/curriculum/readinglist` in the project window.

> Note that `java` is really a shortcut to `src/main/java` or `src/debug/java` or `src/release/java`. The IDE is simplifying it for us.

> Note also that we are choosing `com` not `com (androidTest)` or `com (test)`

This is the main directory for our application. In general, any new packages we create should be under it.

Right-click on the `readinglist` package and choose `New | Package`. Enter the name `model` and click `OK`.



The IDE should now show the model package and the BookListActivity under the `readinglist` package.

Creating Kotlin Data Classes

Most of us will naturally start at the top of the Json and work our way down. If you do that, you will create the "title" variable, then realize you can't create the `ReadingList` model until you create the `Book` model. While creating that model, you'll realize that you can't finish it until you create the `Author` model.

It is in your best interest to fully understand the model you are trying to represent before you start coding it. This is especially true when different sub-models tend to have the same parameters and could be represented by the same type of object (like say, a "thumbnail"). Based on that, and our analysis above - let's work our way inside out and start with models that have no nested dependencies.

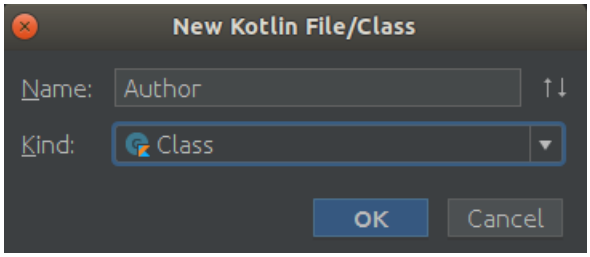
The Author model

From the Json

```
"author" : {
  "firstName" : "Ernest",
  "lastName" : "Hemingway"
},
```

While the "author" tag itself belongs to the containing model, we can use it to name our model. We'll name our model `Author`, and it will have two fields, `firstName` and `lastName`.

Right-click on our new `model` package and choose `New | Kotlin File/Class`.



Name it `Author` and choose a type of `Class`. Click `OK`.

By default, it only gives us

```
package com.aboutobjects.curriculum.readinglist.model

class Author {
}
```

What we are going to do is change it from `class` to `data class` and then replace `{...}` with `(list of parameters)`, like so:

```
package com.aboutobjects.curriculum.readinglist.model

data class Author(
    val firstName: String? = null,
    val lastName: String? = null
)
```

What we have done here is specified that the firstName and lastName parameters should be supplied to the constructor; and said that they *can* be null (generally safer bet if you ever talk to a server).

That's it. The Author model is done.

If you wanted to do some more complicated logic, like say picking a display name based on the values of first and last names; we could do that, but let's hold off.

The Book model

Now that the Author model is complete, we have everything we need to complete the Book model.

```
{
  "title" : "For Whom the Bell Tolls",
  "author" : {
    "firstName" : "Ernest",
    "lastName" : "Hemingway"
  },
  "year" : "1951"
},
```

Like last time, we will add a new `Book` model, make it a `data class` and specify the parameters.

Unlike last time, the `author` parameter will be of type `Author?` instead of `String?`, so that it uses our newly created model.

```
package com.aboutobjects.curriculum.readinglist.model

data class Book(
    val title: String? = null,
    val author: Author? = null,
    val year: String? = null
)
```

Pretty straight forward, right?

Let's do one more.

The ReadingList model

The ReadingList model is a *little* different than the others. It has a title and an array of books. We can specify that we want it to be an array by using a List of a specific type. It will look like this.

```
package com.aboutobjects.curriculum.readinglist.model

data class ReadingList(
    val title: String? = null,
    val books: List<Book> = emptyList()
)
```

You will notice that the `books` parameter defaults to `emptyList()` instead of `null`. That allows you to do things like `readingList.books.size` without checking if it is null or not.

We now have a `ReadingList` model that contains a list of `Book` models that each contain an `Author` model.

Setup Gson

Open your module-level `build.gradle` and add a new dependency.

```
implementation 'com.google.code.gson:gson:2.8.5'
```

Then Sync.

Open your `BookListActivity` and add a new val.

```
private val gson: Gson by lazy {
    Gson()
}
```

There are many ways you can customize that instance (pretty printing for example), documented in the Gson User Guide.

For now, we just want you to get in the habit of re-using the same instance. Down the road, you will move it out of the activity entirely and into somewhere common.

Deserializing

Let's see whether we can use our new models to read the Json that we downloaded earlier.

In your `BookListActivity`, add a new const to the `companion object`

```
const val JSON_FILE = "BooksAndAuthors.json"
```

Add a new function

```
private fun loadJson(): ReadingList? {
    return try{
        val reader = InputStreamReader(assets.open(JSON_FILE))
        gson.fromJson(reader, ReadingList::class.java)
    } catch (e: Exception) {
        // @TODO introduce logging
        null
    }
}
```

The `assets.open` can throw an IOException and the gson call can throw a couple Gson-specific errors. We will learn about debugging and logging shortly. For now, we will treat any error as a failure to load the data by returning `null`.

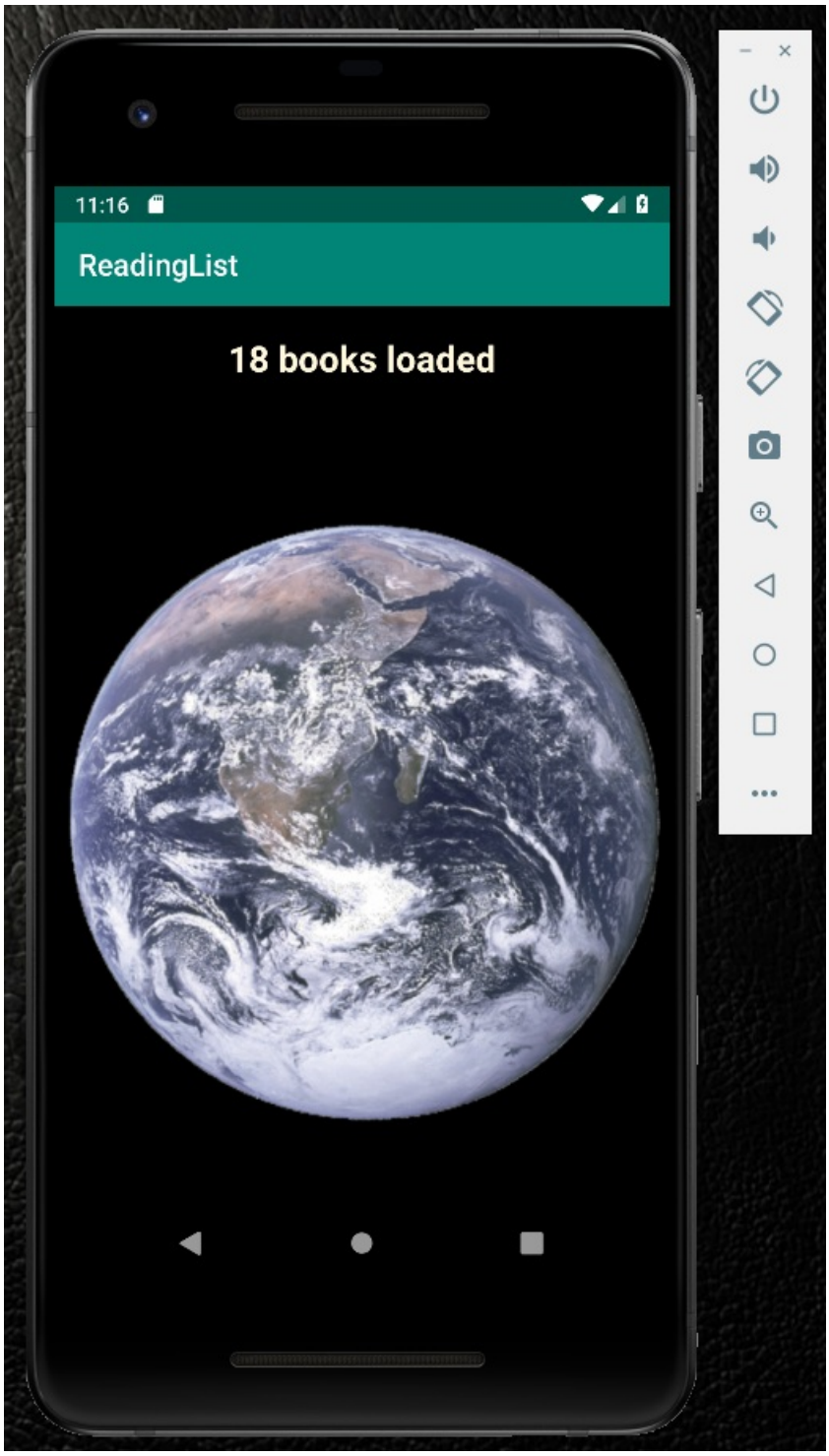
This function could be collapsed into one line, but was expanded to make it easier to read. In essence, it is reading the file you downloaded earlier, from the `assets` directory, and passing it into Gson which will try to convert it into a `ReadingList` type of model.

In your `onCreate`, after the `setContentView` let's add another line

```
loadJson()?.let {
    findViewById<TextView>(R.id.hello_text).text = "${it.books.size} books loaded"
}
```

Here, we will replace the "Hello, Earth" text with some *hardcoded* text *if* the Json file loaded properly.

Deploy your app.



You will notice that the IDE warns you that we hardcoded the text and suggests using a String resource instead (as we discussed earlier).

While this is temporary code (we are going to replace this UI with the actual book list shortly), see if you can figure out how to do what it asks. *Hint:* Look at the rest of your `onCreate` method.

> Note: Normally we would not recommend loading data in the onCreate method as it can be slow. This will be discussed in more depth later.

Serializing

Now that we can load data, how about saving it to Json as well?

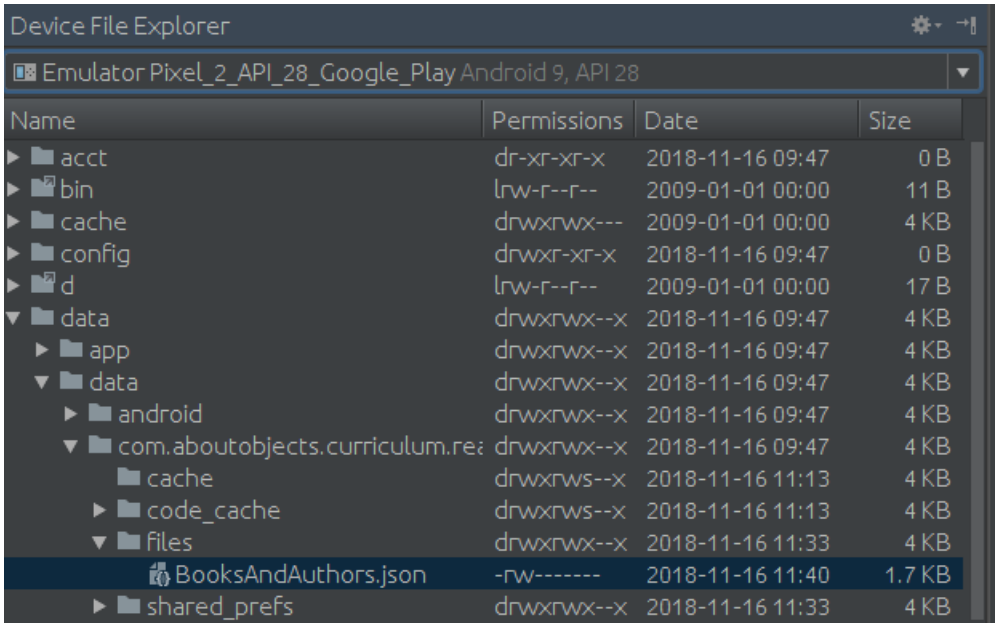
In the `loadJson()?.let` closure that we just created, let's add a bit more.

```
try{
    val writer = FileWriter(File(filesDir, JSON_FILE))
    gson.toJson(it, writer)
    writer.flush()
    writer.close()
}catch(e: Exception){
    // @TODO introduce logging
}
```

Here, we are asking Gson to save our Kotlin data class back out to a new file with the same name, but in the `filesDir`.

Examine Results

Run your application.



On the bottom right corner of your IDE, open the `Device File Explorer`. Similar to how we opened the `shared_prefs` earlier, we are going to open `data/data/com.aboutobjects.curriculum.readinglist/files/BooksAndAuthors.json`.

> If you don't see a `files` directory, right-click on the `com.aboutobjects.curriculum.readinglist` directory and select `Synchronize`.

Now, you'll notice that your Json is one really long unwieldy line. Let's see what we can do about that.

Close the file.

Back in your `BookListActivity`, edit your `gson` val:

```
private val gson: Gson by lazy {
    GsonBuilder()
        .setPrettyPrinting()
        .create()
}
```

Re-run your application and re-open the on-device file.

Much better, right?

Note about Databases

There are many other ways to save and restore data. A common way, locally, has been sqlite3. There are also graph database options like Neo4j that work quite well on Android. Google recently introduced Room, which is a persistence library and acts in much the same way. Those are all larger topics that could be discussed in other classes.

It's also quite common for the server to be the source of data. We'll be talking about that during the chapter on Networking.

#

- Current Repo: v1-Data
- Continue to Unit Testing -->

Debugging.md

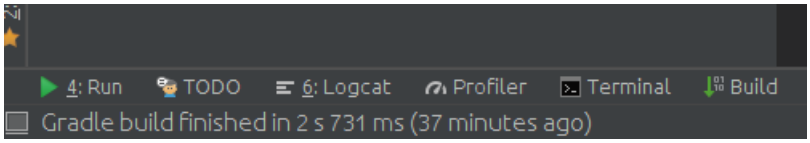
Goal

Learn about debugging and logging.

Table of Contents

- TODO
- Logging
- Debugger
- Profilers

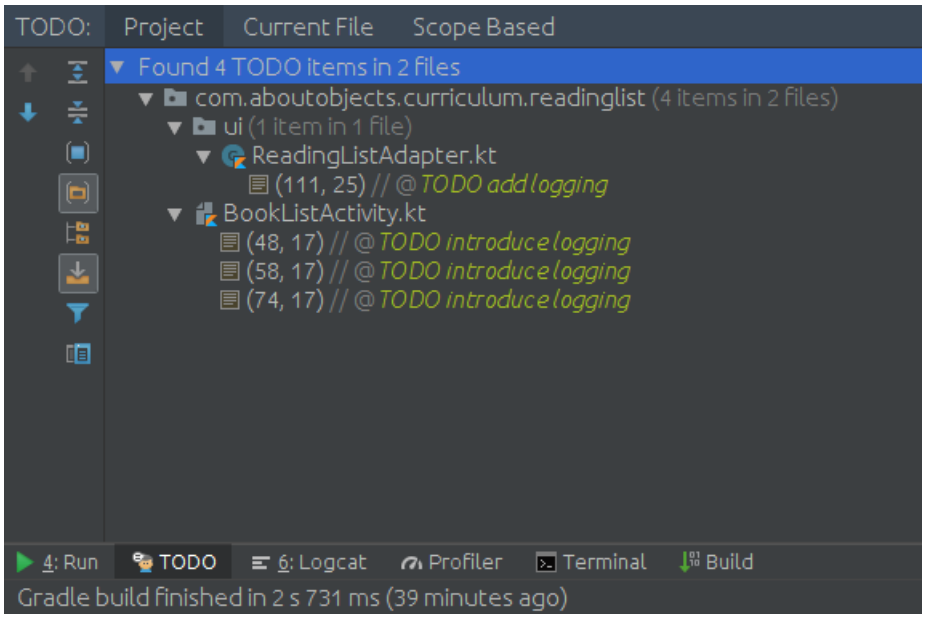
At the bottom of your IDE you will see some tabs that look like this



We're going to spend some time looking at a couple of these.

TODO

Let's first click on the `TODO` tab.



As we were coding, we left a couple of `// @TODO` comments in the code to remind us to come back and do something later.

This window allows us to quickly find all of them throughout the codebase.

> If you are using an issue tracker like JIRA, best practice is to include the ticket number at the beginning of the comment, like `// @TODO [MYAPP-1234] Remove this once server is ready`. That will greatly simplify cross-correlation.

It's important to note that only the *first* line of text will show up here, so on multi-line comments, make sure to make the first line count. The latest version of IntelliJ just changed that within the last month, so it will likely roll out to Android Studio fairly soon.

If you double-click on one of those entries, it will take you directly to that line of code in the editor. We'll be using that feature in the

next section.

> One word of caution: The TODO are only useful if you actually look at them. Adding hundreds of them might lead you to ignoring them. Recommendation would be to add them (with issue tracking identifiers) with the goal of removing them as soon as feasible.

Logging

There are multiple logging libraries available (like Timber, log4j, etc). We're going to focus on Androids' built-in logging facility. Devs sometimes refer to this in the vernacular as `Log.d`, though that is just one of the methods.

Double-click on the `BookListActivity` TODOs. Pick the `loadJsonFromAssets` one.

We're going to replace `// @TODO introduce logging` with a log statement.

There are multiple log methods available. For now, we will focus on "`d`ebug"-level logging. It takes the form of `Log.d(tag, message, throwable)`.

> Most of the documentation you find out there will encourage you to specify a tag based on the class name. As such every class in your application has a different tag. While that can be very helpful for you to differentiate different parts of your application, it is extremely annoying to any developer that is trying to filter out the noise. They have to filter out every single tag that they don't want to listen to, but there is a character limit.
>
> Generally, it is recommended that your application not expose any logs in release builds - but if you have ever used logcat on your device, you will know that almost every application from every vendor spams you.
>
> As such, we will take a different approach. We will use a single tag for our entire application. A single filter can be used to filter our logs, or everything but our logs.

Let's start by adding a new value to your `ReadingListApp`

```
companion object {
    const val TAG = "ReadingListApp"
}
```

Ok, back in our BookListActivity, we can construct our new log message.

```
private fun loadJsonFromAssets(): ReadingList? {
    return try{
        val reader = InputStreamReader(assets.open(JSON_FILE))
        app.gson.fromJson(reader, ReadingList::class.java)
    } catch (e: Exception) {
        Log.d(ReadingListApp.TAG, "Failed to load Json from assets/$JSON_FILE", e)
        null
    }
}
```

You will notice that the TODO no longer shows up on the bottom of the screen.

Let's go ahead and update the other two in this class as well.

```
Log.d(ReadingListApp.TAG, "Failed to load Json from files/$JSON_FILE", e)
```

and

```
Log.d(ReadingListApp.TAG, "Failed to save Json to files/$JSON_FILE", e)
```

While we are at it, let's add an "`i`nformational" log as well. The format is basically the same as the debug log. Add a log to your `onCreate`.

```
loadJson()?.let {
    Log.i(ReadingListApp.TAG, "${it.books.size} books loaded")
    viewAdapter.readingList = it
    saveJson(it)
}
```

To see these logs, click on the `Logcat` button on the bottom of the screen.

In the search bar, enter your TAG, ie `ReadingListApp`

Deploy your application.

You should see a log like this:

```
2018-11-22 11:34:32.202 10761-10761/com.aboutobjects.curriculum.readinglist I/ReadingListApp: 19 boo
ks loaded
```

This gives you your timestamp, process information, package name, log level ("`I`nfo" in this case), the tag (`ReadingListApp`) and your message (`19 books loaded`, because I have one that I added through our UI).

How would we validate the failure message?

We can temporarily force ourselves to mis-save it.

Update our save routine.

```
private fun saveJson(readingList: ReadingList) {
    try{
        val writer = FileWriter(File(filesDir, JSON_FILE))
        app.gson.toJson(readingList, writer)
        writer.append("XYZ") // @TODO temporary for testing - REMOVE
        writer.flush()
        writer.close()
    }catch(e: Exception){
        Log.d(app.TAG, "Failed to save Json to files/$JSON_FILE", e)
    }
}
```

This will output an additional invalid `XYZ` at the end of the json.

Right-click in the Logcat window and `Clear logcat`. That will make it easier to differentiate new logs from old.

Deploy the application with Logcat visible.

The first pass through, everything will seem OK because we are saving not reading the failure.

If you wish to look at the results that were saved, you can see those in the Device File Explorer in the bottom right corner, as before.

Now that we have corrupted our saved data, Deploy the application again with Logcat visible.

```
2018-11-22 11:43:41.182 11275-11275/com.aboutobjects.curriculum.readinglist D/ReadingListApp: Faile
d to load Json from files/BooksAndAuthors.json
    com.google.gson.JsonSyntaxException: com.google.gson.stream.MalformedJsonException: Use JsonRead
er.setLenient(true) to accept malformed JSON at line 157 column 3 path $
        at com.google.gson.Gson.assertFullConsumption(Gson.java:903)
        at com.google.gson.Gson.fromJson(Gson.java:866)
        at com.aboutobjects.curriculum.readinglist.BookListActivity.loadJsonFromFiles(BookListActivi
ty.kt:57)
        at com.aboutobjects.curriculum.readinglist.BookListActivity.loadJson(BookListActivity.kt:65)
        at com.aboutobjects.curriculum.readinglist.BookListActivity.onCreate(BookListActivity.kt:99)
        at android.app.Activity.performCreate(Activity.java:7136)
        at android.app.Activity.performCreate(Activity.java:7127)
        at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)
        at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2893)
        at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3048)
        at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:78)
        at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.ja
va:108)
        at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:68)
        at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1808)
        at android.os.Handler.dispatchMessage(Handler.java:106)
        at android.os.Looper.loop(Looper.java:193)
        at android.app.ActivityThread.main(ActivityThread.java:6669)
        at java.lang.reflect.Method.invoke(Native Method)
        at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)
     Caused by: com.google.gson.stream.MalformedJsonException: Use JsonReader.setLenient(true) to ac
cept malformed JSON at line 157 column 3 path $
        at com.google.gson.stream.JsonReader.syntaxError(JsonReader.java:1568)
        at com.google.gson.stream.JsonReader.checkLenient(JsonReader.java:1409)
        at com.google.gson.stream.JsonReader.doPeek(JsonReader.java:542)
        at com.google.gson.stream.JsonReader.peek(JsonReader.java:425)
        at com.google.gson.Gson.assertFullConsumption(Gson.java:899)
        at com.google.gson.Gson.fromJson(Gson.java:866)
        at com.aboutobjects.curriculum.readinglist.BookListActivity.loadJsonFromFiles(BookListActivi
ty.kt:57)
        at com.aboutobjects.curriculum.readinglist.BookListActivity.loadJson(BookListActivity.kt:65)

        at com.aboutobjects.curriculum.readinglist.BookListActivity.onCreate(BookListActivity.kt:99)

        at android.app.Activity.performCreate(Activity.java:7136)
        at android.app.Activity.performCreate(Activity.java:7127)
        at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)
        at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2893)
        at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:3048)
        at android.app.servertransaction.LaunchActivityItem.execute(LaunchActivityItem.java:78)
        at android.app.servertransaction.TransactionExecutor.executeCallbacks(TransactionExecutor.ja
va:108)
        at android.app.servertransaction.TransactionExecutor.execute(TransactionExecutor.java:68)
        at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1808)
        at android.os.Handler.dispatchMessage(Handler.java:106)
        at android.os.Looper.loop(Looper.java:193)
        at android.app.ActivityThread.main(ActivityThread.java:6669)
        at java.lang.reflect.Method.invoke(Native Method)
        at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)
2018-11-22 11:43:41.184 11275-11275/com.aboutobjects.curriculum.readinglist I/ReadingListApp: 18 bo
oks loaded
```

You'll notice that it told us specifically which one failed (thanks to our unique messages): `Failed to load Json from files/BooksAndAuthors.json`.

It also told us where it failed: `BookListActivity.loadJsonFromFiles(BookListActivity.kt:57)` as well as the full stack trace that got it there. This part may be more or less useful depending on the interaction between various threads.

In this case, it also told us the `Caused by` was `malformed JSON at line 157 column 3`

Remove the bad line we added from the `saveJson`, clear logcat and re-deploy.

So why did it still fail?

Like last time, it attempted to read the un-corrected json first. That failed, so it read the assets version.

This time, however, it saved a good copy over the top of the bad one. If you re-deploy again, everything should be fine (though you now only have the original 18 books).

> It's important that you keep this in mind if you are deploying changes to your customers that affect how the data is saved and restored.

If you look at your TODO list, we have one more. Let's complete that. It's in your `ReadingListAdapter.onBindViewHolder`. This one would only happen when we are trying to bind an item that we hadn't already accounted for. Currently, it would never happen - but it could down the road as you are making tweaks.

Let's use a `w`arning log statement this time.

```
else -> {
    Log.w(ReadingListApp.TAG, "Trying to bind to unknown type: ${holder.binding}")
}
```

Now, if we add a new type to `getItemViewType` and forget to update `onBindViewHolder`, we'll get a warning; which looks something like this:

```
2018-11-22 12:30:26.455 11888-11888/com.aboutobjects.curriculum.readinglist W/ReadingListApp: Trying
 to bind to unknown type: com.aboutobjects.curriculum.readinglist.databinding.ItemFakeBindingImpl@96
7fcda
```

> No matter how good our tools get, you will always rely on some amount of logging. If you use services like Fabric or Google Play, the logs may be all you have to go on to debug a customer issue. Sometimes it's all about seeing something out of place or comparing two logs and noticing that something looks different between them.

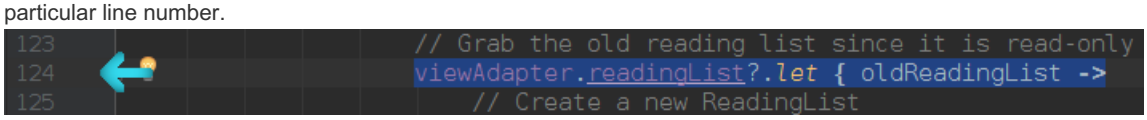
Debugger

Other times, you want to step through the code and try to understand what is happening.

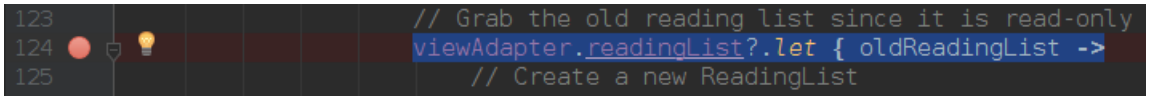
Let's look at an example.

Open `BookListActivity` and scroll down to `onActivityResult`.

In the margin near the line numbers, click here to set a breakpoint. Make sure you match the content of the line rather than the particular line number.

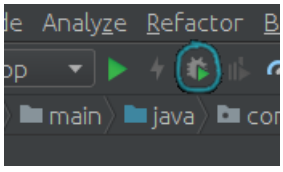


When set it will look like this



You can click it again to turn it on/off. Leave it on for now.

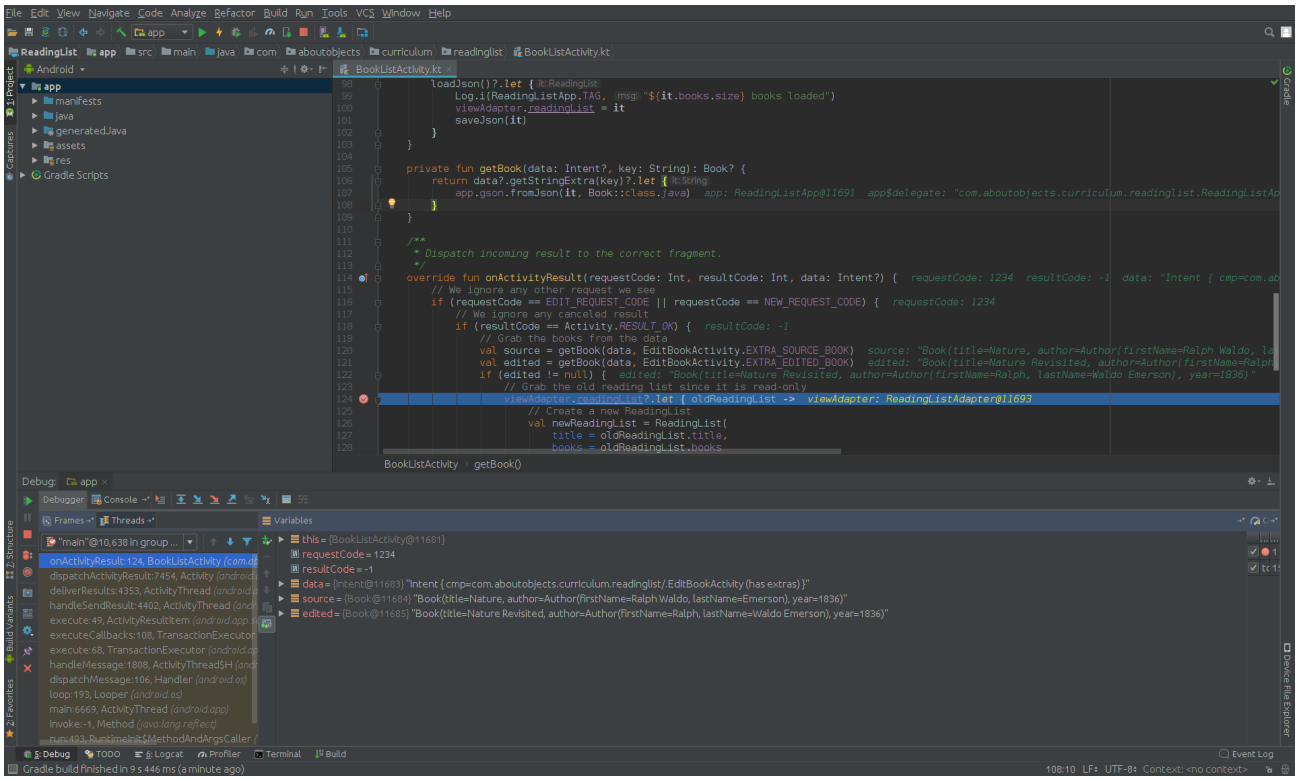
A breakpoint indicates where you would like execution to pause, so that you can inspect what is going on. In this case, we want the program to pause when we have an `edited` book that we are about to save.



Near the green `Run 'app'` "play" button is a `Debug 'app'` button. You can identify it because it looks like a gray bug with a play button on it.

Click that button to launch a debugging session.

This will deploy your application. Edit a book and click save.



When the breakpoint is activated, the IDE will show a lot of useful information.

Editor

First, you will notice that the IDE highlights the line where you set the breakpoint.

You will also notice that it shows some hints about values. For example, next to the `source` and `edited` lines, it tells you what those values are.

Variables

Below the editor, you will see a `Variables` window.

It contains a tree-view of all the variables currently in scope. Expand `edited`, then expand `author`. Notice how it shows the values of your nested models.

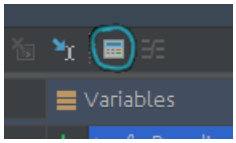
New Watch

What if you wanted to know about some variable that wasn't displayed?

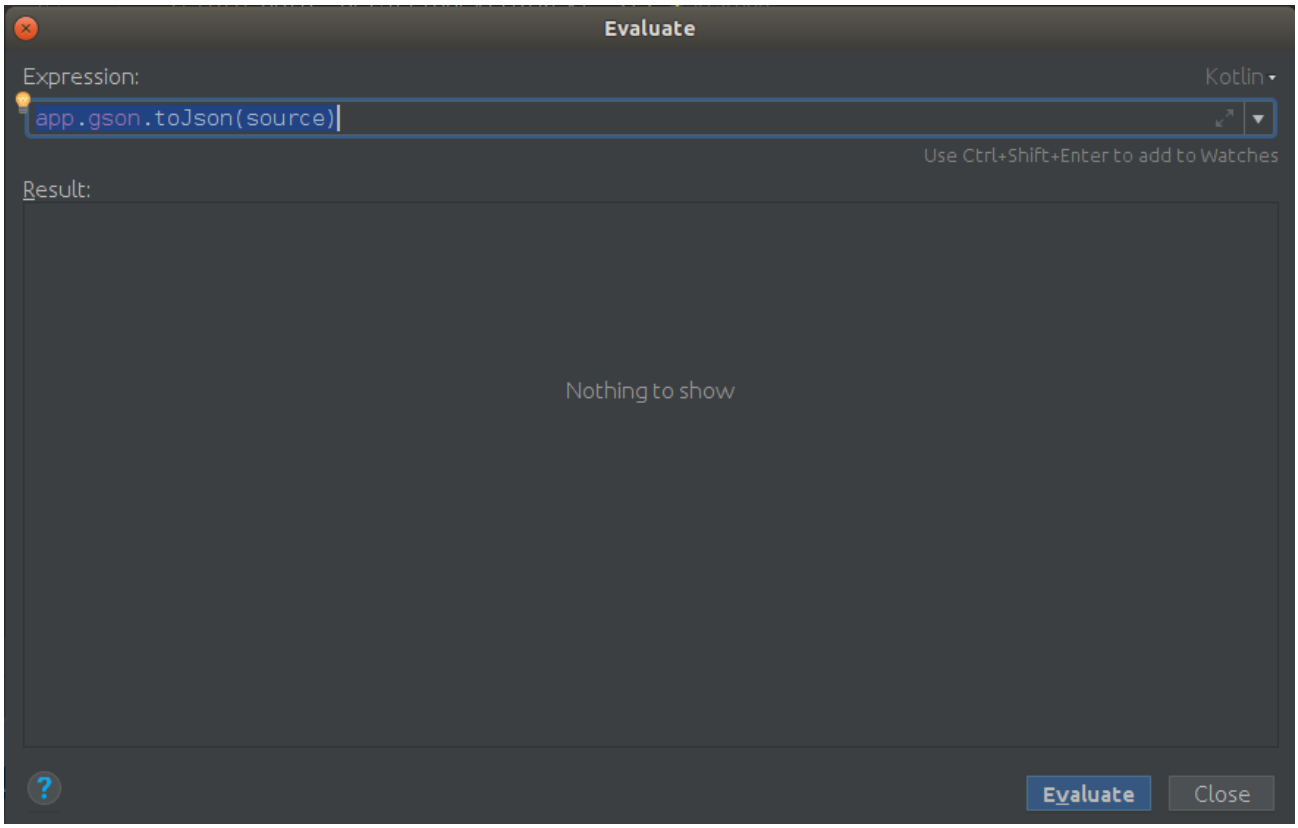
Right-click in that window, and choose `New Watch`. In the new box, type `ReadingListApp.TAG` and hit enter.

Evaluate Expression

What if you want to evaluate some expression?



Click this button. The tooltip says `Evaluate Expression`.



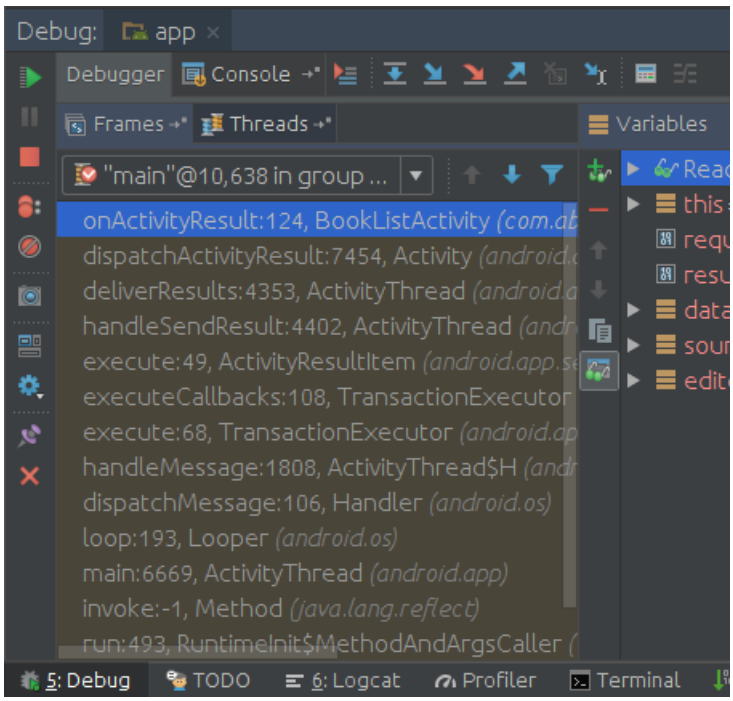
You can enter most expressions that would be valid if they were entered in the source code where the breakpoint is currently at.

Make sure the text box has focus, and enter `app.gson.toJson(source)` and click `Evaluate`.

You'll notice that the results look more like our `Variables` window than the pretty printing that `Gson` would normally provide for us.

Go ahead and close that window.

Frames



To the left of the `Variables` is a tabbed section within another tabbed section. The top tabbed section has `Debugger` and `Console`. When `Debugger` is selected, the inside one has `Frames` and `Threads`.

The `Frames` tab shows the stack trace that got us to the current line of code.

Here, we can see that are currently in the `main` (UI) thread and that we are in `onActivityResult` line 124 of `BookListActivity`.

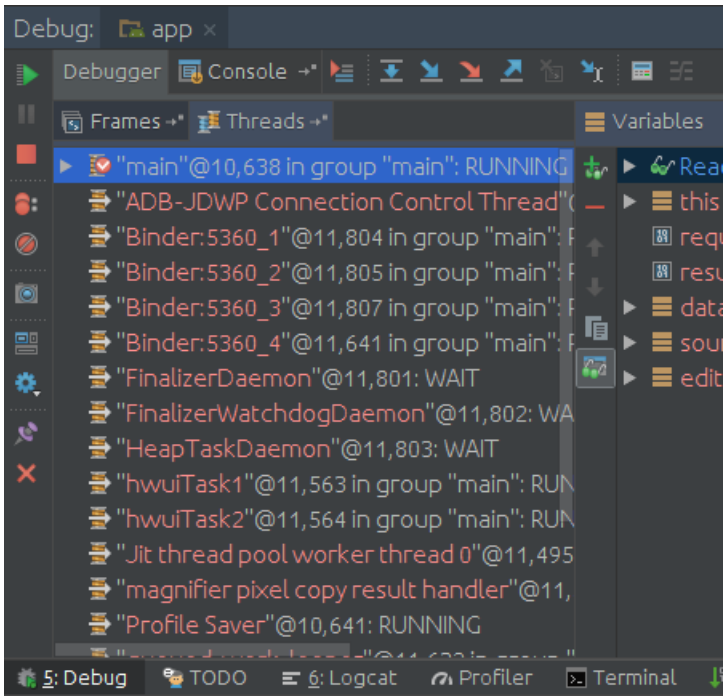
> Don't be concerned if your line numbers are off slightly. Extra carriage returns and comments can change this spacing. In addition, the IDE will automatically change your imports to be more optimial as you use more or less classes within a single package.

Further, we can see it was called from `dispatchActivityResult` line 7454 of `Activity`. Click it.

Notice how it takes you directly to that source code?

Go ahead and click back on the first `onActivityResult` line to go back to where we were.

Threads



If you click on the Threads tab, and collapse `main`, you will see a list of threads.

> While you may not be creating a lot of threads yet, there are still multiple threads being created behind the scenes. It's important to realize that *all* apps are multi-threaded, whether you intend them to be or not.

You'll use the `Frames` tab most of the time. Switch back to it.

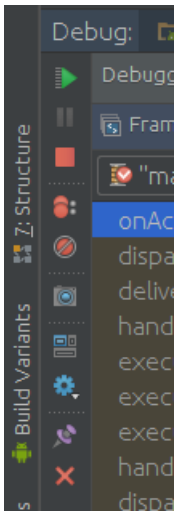
Console

The `Console` tab is similar to logcat. It also includes information about the `adb` commands the IDE is executing on your behalf.

If you start working from the command line, this can be a useful view to understand why your commands may not be working the same way.

Switch back to the `Debugging` tab.

Debugging Controls



On the far left of the screen you will see your debugging controls.

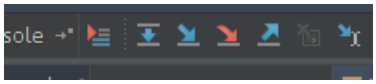
There is a green `Resume Program` button. This should ***NOT*** be confused with the green `Run 'app'` "play" button at the top of the screen. If you click the `Run 'app'` button instead, it will stop the debugging session and redeploy your application.

The `Resume Program` button will continue execution until it finds another breakpoint.

There is also a `Pause Program` and `Stop 'app'` button.

`Pause Program` will allow you to pause execution wherever it is right now. That can be useful if you want to figure out what it is doing.

`Stop 'app'` will disconnect the debugger from your application.



There are also some controls across the top of the window.

If you have ever used a debugger before, these will be familiar. The most useful ones to you will most likely be `Step Over`, `Step Into` and `Step Out`. If you are ever unsure which icon is which, hover over them for the tooltips.

`Step Over` will allow you to go to the next line.

`Step Into` will let you go into the method and examine it.

`Step Out` will allow you to return from a Step Into.

Let's take a look at a couple of these.

Press the `Step Over` and notice that it took you to the next line (outside of the lambda):

```
super.onActivityResult(requestCode, resultCode, data)
```

Press the `Step Into` and notice it took you into the `FragmentActivity.onActivityResult`.

Press the `Step Out` and notice it takes us back to where we were (actually to the `}` right after).

Finally, press the `Resume Program`. Since we have no more breakpoints, the application is running normally and won't trigger again unless you edit another book.

Attaching the Debugger

What if you already have the program running and you want to attach the debugger without restarting the application?

We can do that.

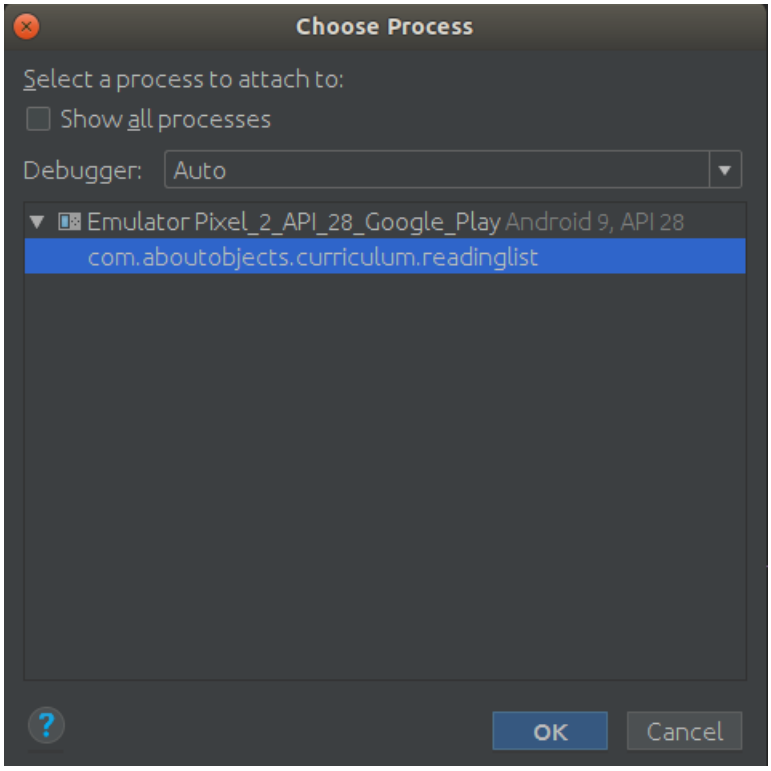
First, let's get ourselves into that state.

If your application is currently running, click the red `Stop 'app'` button at the top of the screen so we can start this exercise from scratch.

Deploy (not Debug) your application, with the `Run 'app' button`.

Edit a book and confirm that the debugger doesn't interject.

From the Toolbar, click on `Run | Attach debugger to Android process`. It's probably at the bottom of the menu.



It should auto-select your application. If not, select it.

Click `OK`.

Edit another book and save.

The debugger should have interrupted your saving process. Go ahead and click on `Resume program`.

> Note: If you attach and stop the debugger repeatedly, this dialog may eventually not show your application as an option. If that happens, you will have to stop your application and restart it.

Profilers

While outside the scope of this class, you can find more information about the profilers available in the IDE on the Android Site.

#

- Current Repo: v1-Debugging
- Continue to Threading -->

Drawing.md

Goal

Information about custom drawing.

Table of Contents

- Drawable xml files
- onDraw

- Advanced Graphics

There are various ways you can do custom drawing within your application. Before you do so, however, you should always make sure that it is absolutely necessary. While custom components can give you the exact look you are looking for, they will rarely auto-adjust to newly released versions of Android or updated theme guidelines.

While the Android documentation still shows the Activity Lifecycle and the Fragment Lifecycle, it no longer seems to host an image for the View Lifecycle. There are some available in Google Images - but perhaps it is better if we just try it and see what happens?

Let's consider a real world example. How about we customize your reading list?

Drawable xml files

Android allows us to specify drawable resources in XML.

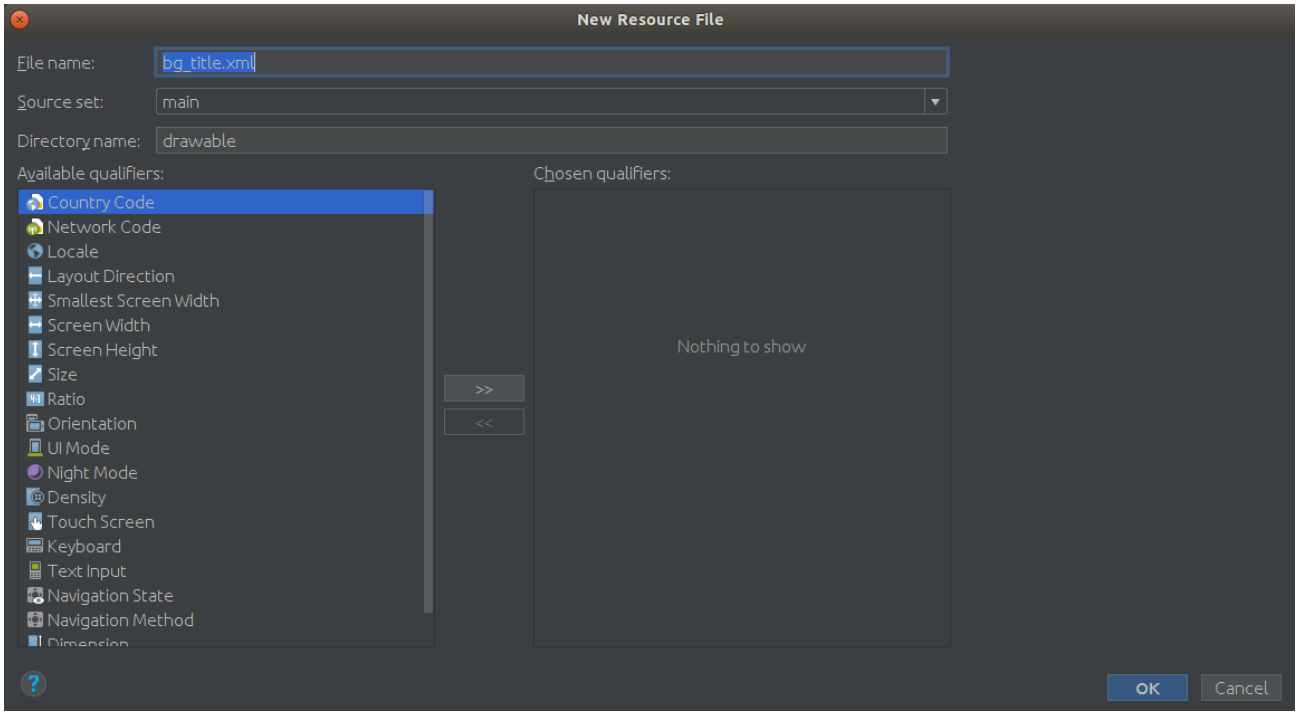
colors.xml

First, let's define a couple colors. Open your `colors.xml`. Add these

```
<color name="cornflower">#6495ed</color>
<color name="lightGray">#d3d3d3</color>
<color name="hanBlue">#446ccf</color>
```

bg_title.xml

Now, right-click on your `res/drawable` folder and choose `New | Drawable resource file`.



Name it `bg_title.xml` and click `OK`.

It may start you in `Design` mode. Switch to `Text` mode. We're going to replace the XML.

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <gradient
        android:startColor="@color/cornflower"
        android:centerColor="@color/lightGray"
        android:endColor="@color/hanBlue"
        android:angle="90"/>
    <corners
        android:radius="0dp"/>
</shape>
```

Documentation for this format can be found on the Android Developer site.

item_title.xml

Open your `item_title.xml` and add a new background element to the TextView:

```
android:background="@drawable/bg_title"
```

> Remember that auto-complete it your friend.

Deploy and see the change.

onDraw

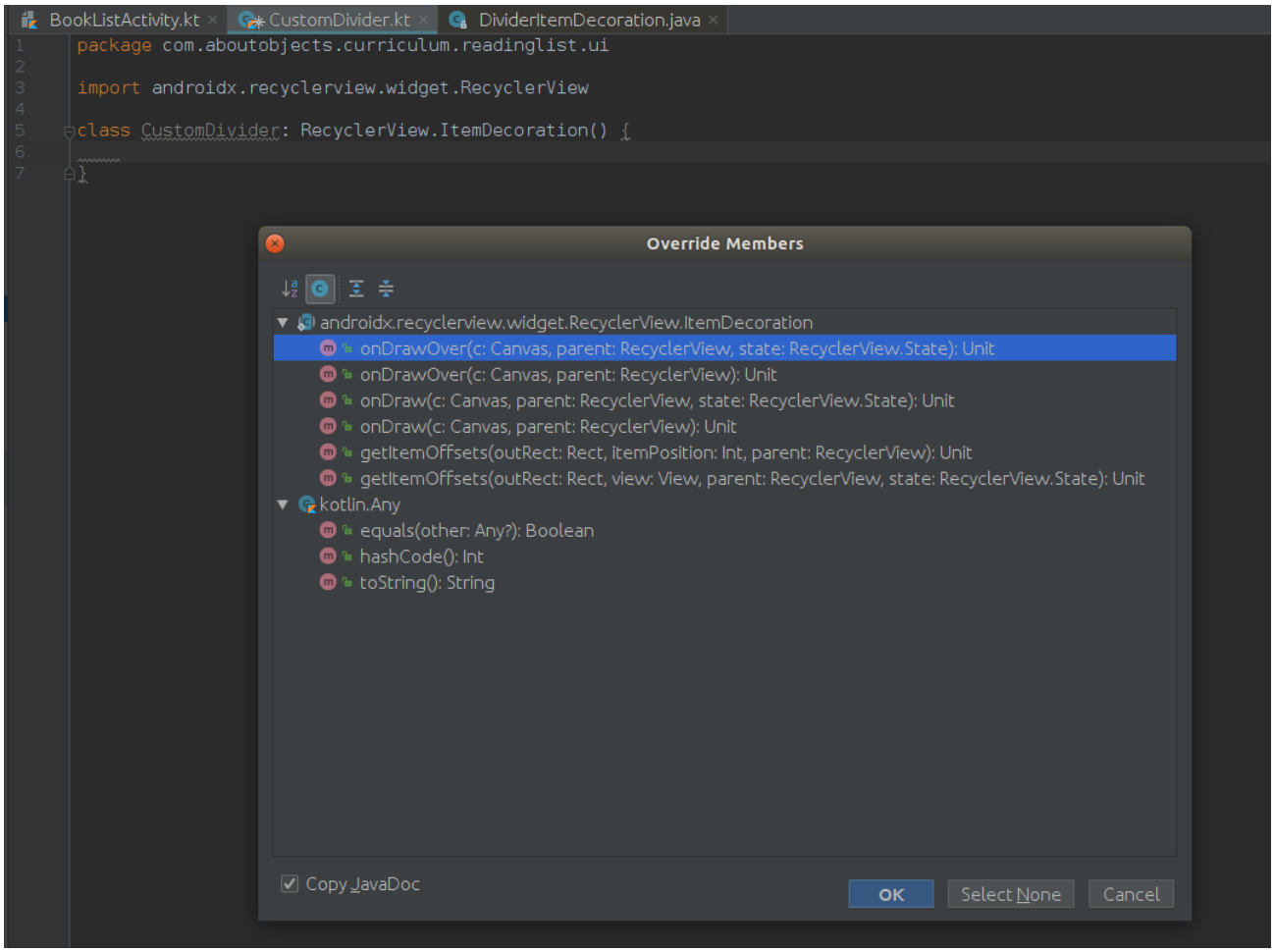
Let's try some custom drawing.

Create a new class in your `ui` package called `CustomDivider`.

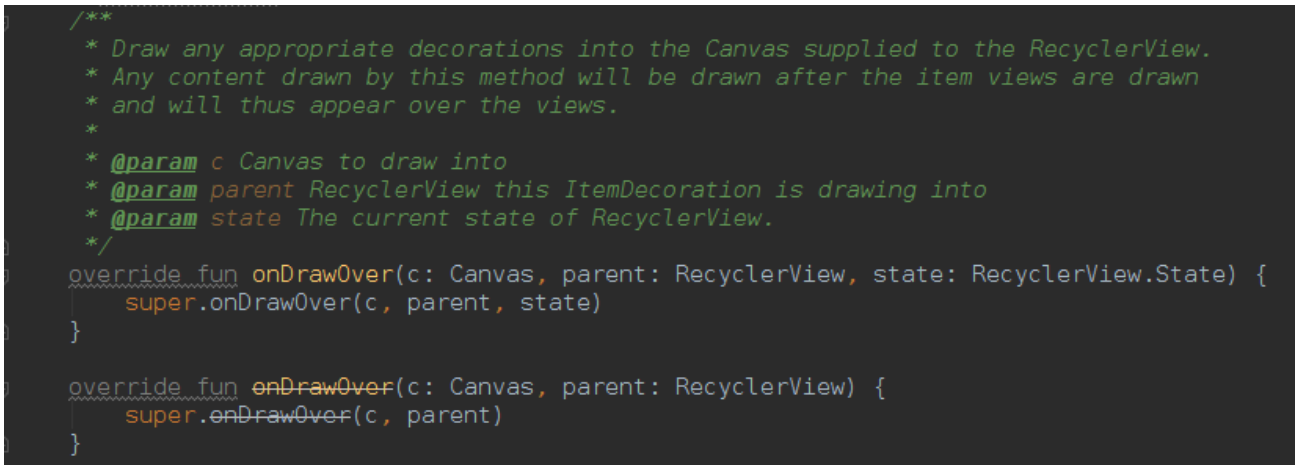
We'll extend the `RecyclerView.ItemDecoration` class. We'll also pass a `Context` in as a parameter.

```
package com.aboutobjects.curriculum.readinglist.ui

import android.content.Context
import androidx.recyclerview.widget.RecyclerView

class CustomDivider(context: Context): RecyclerView.ItemDecoration() {
}
```

If you attempt to override any methods at that point, you will notice that we have a few `onDraw` methods available to us. Unfortunately, it does *NOT* tell you which ones are deprecated.



Select all of the `ItemDecoration` methods, make sure `Copy Javadoc` is selected and click `OK`.



```
    /**
     * Draw any appropriate decorations into the Canvas supplied to the RecyclerView.
     * Any content drawn by this method will be drawn after the item views are drawn
     * and will thus appear over the views.
     *
     * @param c Canvas to draw into
     * @param parent RecyclerView this ItemDecoration is drawing into
     * @param state The current state of RecyclerView.
     */
    override fun onDrawOver(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
        super.onDrawOver(c, parent, state)
    }

    override fun ~~onDrawOver~~(c: Canvas, parent: RecyclerView) {
        super.~~onDrawOver~~(c, parent)
    }
```

Remove any method that has a strikethrough on it. You should be left with only 3 methods.

```
override fun onDrawOver(c: Canvas, parent: RecyclerView, state: RecyclerView.State)
override fun onDraw(c: Canvas, parent: RecyclerView, state: RecyclerView.State)
override fun getItemOffsets(outRect: Rect, view: View, parent: RecyclerView, state: RecyclerView.Sta
te)
```

Before we go too far off in the weeds, let's understand when these are called.

Add a custom log message to each function.

Then. in your `BookListActivity` replace this line:

```
addItemDecoration(DividerItemDecoration(this@BookListActivity, DividerItemDecoration.VERTICAL))
```

with

```
addItemDecoration(CustomDivider(context = this@BookListActivity))
```

Enable `Logcat` and watch for your `ReadingListApp` tag.

Deploy your application and scroll through the reading list.

What did you notice?

These methods get called so frequently, we want to make sure that they execute very fast. There is more to it than that. Any memory leak in this code can also skyrocket quickly. Some of the blogs will tell you that `onDraw` is only called once - but Logcat has just shown you otherwise.

Open your `CustomDivider`.

Reduce memory

One of the ways we can reduce memory is by avoiding re-allocation of variables that don't change. For example, instead of specifying a `Paint` object every single time the `onDraw` method is called, we can set it up once and re-use it on each call.

Let's setup a couple variables to use.

Since we have one method that will draw "under" and one that will draw "over" (you did read those Javadocs, right?), let's setup two sets of variables so that we can clearly see what is happening.

```
private val overPaint = Paint()
private val underPaint = Paint()
private val thickness = TypedValue.applyDimension(
    TypedValue.COMPLEX_UNIT_DIP,
    5.5f,
    context.resources.displayMetrics
)
private val center = thickness / 2

init {
    overPaint.color = Color.WHITE
    overPaint.strokeWidth = thickness / 3

    underPaint.color = Color.BLUE
    underPaint.strokeWidth = thickness
}
```

Here we setup two `Paint` objects and set the "under" one to BLUE and the "over" one to WHITE. We are setting the thickness of the BLUE one to 5.5 dp and the WHITE one to 1/3rd of that. Since both will be centered, the end result will be a WHITE line centered within a thicker BLUE line.

getItemOffsets

Let's start by updating our `getItemOffsets`. We'll simply make enough room for our line.

```
override fun getItemOffsets(outRect: Rect, view: View, parent: RecyclerView, state: RecyclerView.State) {
    outRect.set(0, 0, 0, thickness.toInt())
}
```

The IDE will show you that this is `outRect.set(left, top, right, bottom)`.`

> Unless you are debugging, you probably don't want to keep active logging in these calls since they will slow down the rendering.

drawLines, onDraw, onDrawOver

Since both `onDraw` and `onDrawOver` are going to draw a centered line, and the only difference is the paint color... let's simplify rather than duplicate.

First, we'll create a generic method that also takes a `Paint` variable.

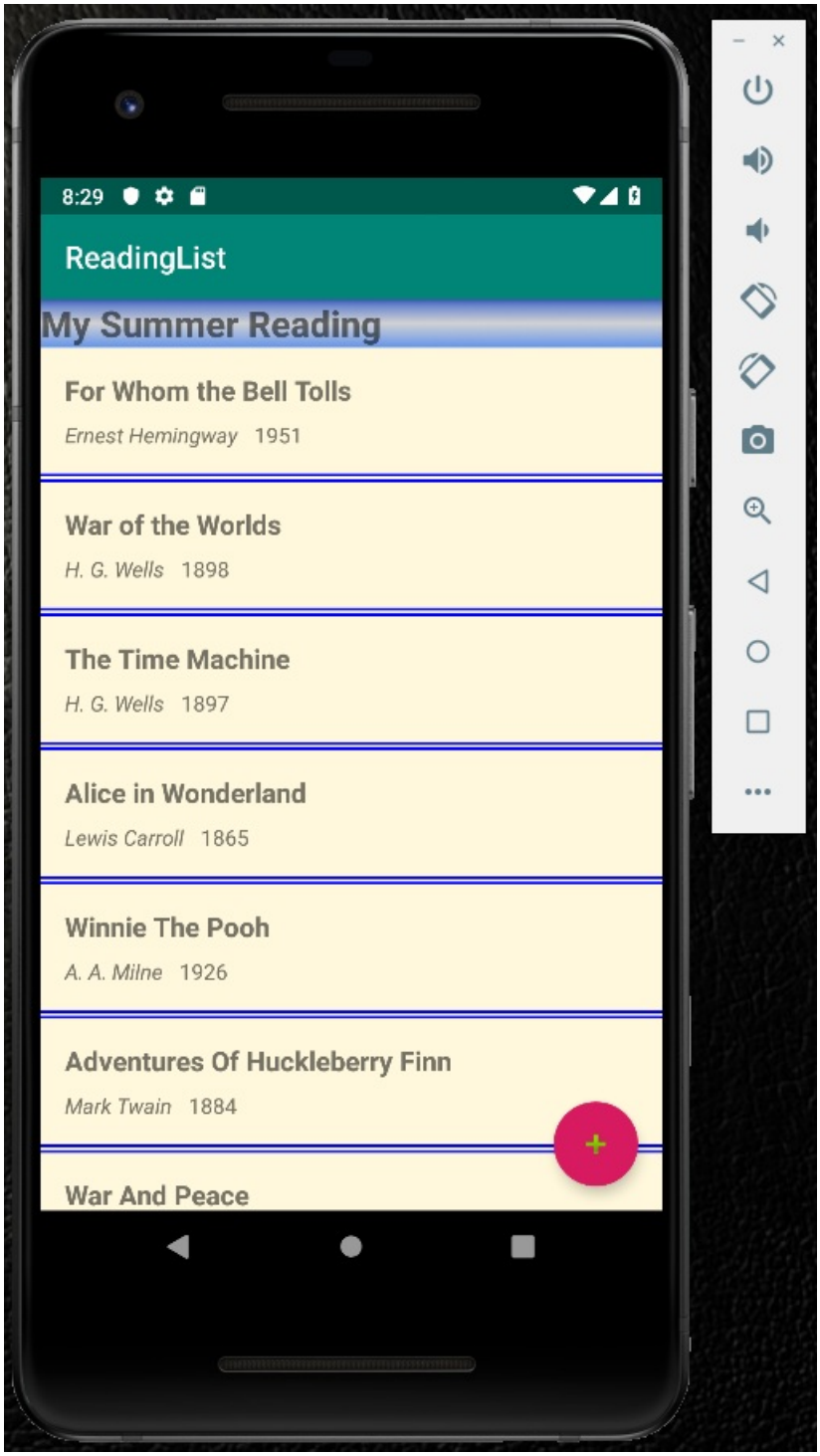
```
private fun drawLines(c: Canvas, parent: RecyclerView, paint: Paint) {
    for (i in 0 until parent.childCount) {
        val view = parent.getChildAt(i)
        c.drawLine(view.left.toFloat(),
            view.bottom.toFloat() + center,
            view.right.toFloat(),
            view.bottom.toFloat() + center,
            paint)
    }
}
```

The `c.drawLine(startX, startY, stopX, stopY, paint)` wants us to specify the center. If startY and stopY are different, you will get a diagonal line. We are specifying that the startY/stopY should be below the previous view, at the center of our thickness.

Now we can implement our `onDraw` and `onDrawOver`

```
override fun onDraw(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
    drawLines(c, parent, underPaint)
}

override fun onDrawOver(c: Canvas, parent: RecyclerView, state: RecyclerView.State) {
    drawLines(c, parent, overPaint)
}
```

Deploy your application.



Advanced Graphics

This just scratches the surface of custom drawing. If you are interested in more advanced graphic development, here are some additional resources to look into.

- Sceneform/ARCore

> Sceneform makes it straightforward to render realistic 3D scenes in AR and non-AR apps, without having to learn

> OpenGL.

- OpenGL

> OpenGL is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware.

- Vulkan

> Vulkan is a low-overhead, cross-platform API for high-performance, 3D graphics.

- LWJGL

> LWJGL is a Java library that enables cross-platform access to popular native APIs useful in the development of graphics (OpenGL, Vulkan), audio (OpenAL) and parallel computing (OpenCL) applications.

#

- Current Repo: v1-Drawing
- Continue to Fragments -->

Fragments.md

Goal

Convert our Activity-based UI to a Fragment-based UI.

Table of Contents

- Creating the Layouts
- Creating the Fragments
- Our v2 Activity
  - v2 Layout
  - BookListV2Activity
  - AndroidManifest
- supportFragmentManager
- Business Logic Decisions
  - BookListService
- Completing the Fragments

In the next chapter, we are going to take a look at animation.

Before we can animate the transition, we need to switch from an Activity-based application to a Fragment-based application.

Currently, we have our `BookListActivity` and our `EditBookActivity`. Each of these are standalone, only one being active at a time.

The Fragment design allows us to have one or more active at the same time. We end up with one Activity that owns whichever Fragments are currently active.

Why does this matter? Well, in our case specifically, we are going to ask the Fragment Manager to show the animation when switching between two fragments.

Normally, you would refactor the activity to become a fragment; but in this case, we will make new classes to make it obvious what the differences are.

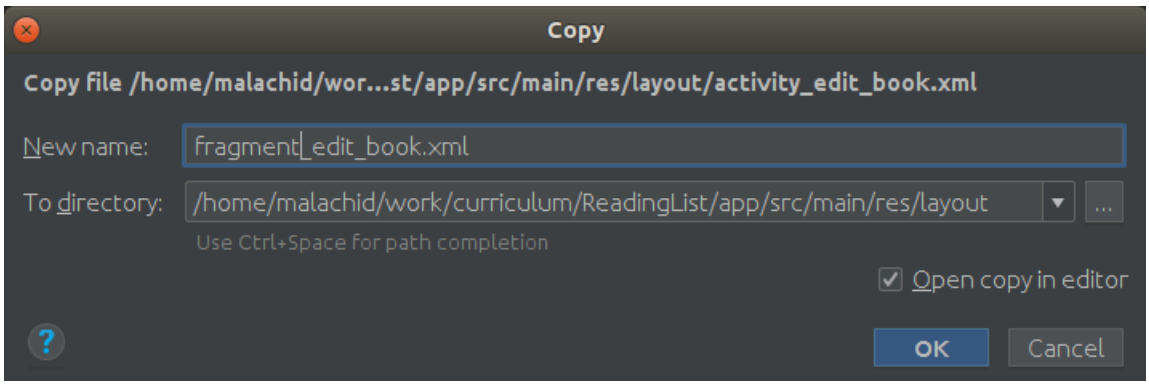
Creating the Layouts

Our layouts are exactly the same in both Activity-based and Fragment-based. In a production app, you would just rename the file. In our case, we will keep both around around so it's clear what the difference is between the two approaches.

fragment_edit_book.xml

Right-click on your `res/layout/activity_edit_book.xml` and choose `Copy`.

Right-click on your `res/layout` folder and choose `Paste`.



Enter `fragment_edit_book.xml` as the new name and click `OK`.

fragment_book_list.xml

Copy `activity_book_list.xml` to `fragment_book_list.xml`

Deploy your application to regenerate binding files.

Creating the Fragments

Just like before, we are going to create new classes to represent these layouts.

EditBookFragment

Now, you might be tempted to use the `New | Fragment | Fragment (Blank)` option here, but it adds a lot of unnecessary cruft that you don't need. Let's do it by hand instead.

Just like you've done before, create a new Kotlin class in your `ui` package called `EditBookFragment`.

It will extend `androidx.fragment.app.Fragment`.

```
package com.aboutobjects.curriculum.readinglist.ui

import androidx.fragment.app.Fragment

class EditBookFragment: Fragment() {
}
```

Add a new var to hold our binding class.

```
private lateinit var binding: FragmentEditBookBinding
```

Override the `onCreateView` method to reference our new layout.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundl
e?): View? {
    binding = DataBindingUtil.inflate(inflater, R.layout.fragment_edit_book, container,false)
    return binding.root
}
```

BookListFragment

Create a `BookListFragment` and do the same for `R.layout.fragment_book_list`.

```
package com.aboutobjects.curriculum.readinglist.ui

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.databinding.DataBindingUtil
import androidx.fragment.app.Fragment
import com.aboutobjects.curriculum.readinglist.R
import com.aboutobjects.curriculum.readinglist.databinding.FragmentBookListBinding

class BookListFragment: Fragment() {

    private lateinit var binding: FragmentBookListBinding

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: B
undle?): View? {
        binding = DataBindingUtil.inflate(inflater, R.layout.fragment_book_list, container, false)
        return binding.root
    }
}
```

Our v2 Activity

While we are talking about changing from an Activity-based layout to a Fragment-based layout; we actually still need at least one Activity to launch our application.

v2 Layout

activity_book_list_v2.xml

Create a new layout, `activity_book_list_v2.xml` with a `layout` root element.

This one will be much simpler than our previous ones. It's just going to contain a single container.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>

    </data>
    <FrameLayout
        android:id="@+id/container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</layout>
```

Deploy your application to regenerate the binding files.

BookListV2Activity

BookListV2Activity

And for the class to represent that layout, create a new `BookListV2Activity` in your `readinglist` package.

We will once again extend `AppCompatActivity`, which in turn extends `FragmentActivity`.

```
package com.aboutobjects.curriculum.readinglist

import androidx.appcompat.app.AppCompatActivity

class BookListV2Activity: AppCompatActivity() {
}
```

Use the Quick Fix to add the activity to the manifest.

AndroidManifest

AndroidManifest.xml

Open your `AndroidManifest.xml` and copy the `intent-filter` from `.BookListActivity` to `.BookListV2Activity`.

```
<activity android:name=".BookListV2Activity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Highlight the entire `intent-filter` block of code for the original `BookListActivity`. Use `Ctrl+/` (or `Command+/` on a Mac) to comment out that block of code.

You can deploy your application now, but it won't do much. The old application code is still here, but it is no longer being launched.

> If you choose to keep both, you can add an `android:label` to the `activity` element to give them each different names in the phones' application drawer.
>
> If you wanted to keep both active by default, you *could* change your configuration under `Run | Edit Configurations` to specify which one should start up.

supportFragmentManager

To switch between fragments, we use the `supportFragmentManager`.

build.gradle

In your module-level `build.gradle` add another KTX dependency.

```
implementation 'androidx.fragment:fragment-ktx:1.0.0'
```

This dependency gives us extensions for managing fragment transactions more cleanly.

Sync your application.

Back in BookListV2Activity

Add our binding var

```
private lateinit var binding: ActivityBookListV2Binding
```

Override the `onCreate(savedInstanceState: Bundle?)` method. There are a lot of `onCreate` methods, so make sure you grab the correct one.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = DataBindingUtil.setContentView(this, R.layout.activity_book_list_v2)

    savedInstanceState?.let {
        Log.d(ReadingListApp.TAG, "UI already loaded")
    } ?: let {
        supportFragmentManager
            .beginTransaction()
            .add(R.id.container, BookListFragment())
            .commitAllowingStateLoss()
    }
}
```

This is similar to what we did before. We are now adding a check for the `savedInstanceState`. Most of the time, developers do not put that initial block in - but we will include it to help us better understand what is happening.

In the case where the `savedInstanceState` is `null`, we then request that the `supportFragmentManager` (based on the KTX component we just added) add a new instance of our `BookListFragment` to the `@+id/container` in our `activity_book_list_v2.xml`.

Deploy your application.

We should probably make the new version do something?

Business Logic Decisions

Before, where logic should exist was obvious. Now, we have decisions to make.

While it is clear that editing should be done in the `EditBookFragment`, should we read the `ReadingList` in the `BookListV2Activity` or in the `BookListFragment`?

Surprisingly, it could be done either way and you will find that people will not always agree. In fact, you might not agree with yourself a few months into the project.

There may actually be another option for us. What if neither class does?

BookListService

We are going to introduce the concept of a `BookListService`. While it will not extend `android.app.Service`, it does have similar goals in mind.

A typical Android `Service` is defined as `A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface.`

So while we are not extending that class, it is typical for developers to use the same terminology as the intention is clear to everyone. If we were to call it `BookListUtil`, for example, it starts to become a kitchen sink that gets muddied with everything to do with book lists.

So let's be clear. Our `BookListService` will be for saving and loading of list of books; and will have no UI.

service package

Right-click on your `readinglist` package and create a `New | Package` called `service`.

BookListService.kt

In it, create a new class called `BookListService`.

```
package com.aboutobjects.curriculum.readinglist.service

import com.aboutobjects.curriculum.readinglist.ReadingListApp

class BookListService(val app: ReadingListApp) {
}
```

You will notice that we are passing in our `ReadingListApp` as a parameter. This will serve as our UI-less `Context`.

> Note: In a production application, it is recommended that you would use dependency injection using something like Dagger, but that is out of scope for this chapter.

ReadingListApp

Open your `ReadingListApp` and create a val for the new service.

```
val bookListService: BookListService by lazy {
    BookListService(app = this)
}
```

The first time the `bookListService` is referenced, it will call the `BookListService` constructor. If we remove the `by lazy`, it would be initialized immediately, probably before the UI is even loaded.

Copying Logic

Now, we will copy a lot of the logic from the original `BookListActivity` into the new generic `BookListService`. You'll want to have both classes open.

JSON_FILE

We'll copy the Json filename into `BookListService`.

```
companion object {
    const val JSON_FILE = "BooksAndAuthors.json"
}
```

BehaviorSubject

Add a variable in `BookListService` (not in the `companion object`) to store the current reading list.

```
val readingList: BehaviorSubject<ReadingList> = BehaviorSubject.create()
```

A `BehaviorSubject` allows us to cache the current value. You can read the current value, or wait for a value to be set.

loadFrom functions

Next, we'll grab our old `load..` functions from `BookListActivity`. With some minor changes to the method names and the logs, add them to our `BookListService`.

```
private fun loadFromAssets(): Maybe<ReadingList> {
    return try{
        val reader = InputStreamReader(app.assets.open(JSON_FILE))
        Maybe.just(app.gson.fromJson(reader, ReadingList::class.java))
    } catch (e: Exception) {
        Log.d(ReadingListApp.TAG, "Failed to load Json from assets/${BookListService.JSON_FILE}",e)
        Maybe.empty<ReadingList>()
    }
}

private fun loadFromFiles(): Maybe<ReadingList> {
    return try {
        val reader = FileReader(File(app.filesDir, JSON_FILE))
        Maybe.just(app.gson.fromJson(reader, ReadingList::class.java))
    } catch (e: Exception) {
        Log.d(ReadingListApp.TAG, "Failed to load Json from files/${BookListService.JSON_FILE}",e)
        Maybe.empty<ReadingList>()
    }
}
```

You'll also notice that we are referencing the `app` for the context.

loadJson

Next, we'll grab the logic that was in both `loadJson()` and `onCreate` and put them into a new `BookListService.init` block.

```
init {
    val loadDisposable = loadFromFiles()
        .switchIfEmpty(loadFromAssets())
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeBy(
            onSuccess = readingList::onNext,
            onError = { Log.w(ReadingListApp.TAG, "Error loading books", it) },
            onComplete = { Log.w(ReadingListApp.TAG, "No books found") }
        )
}
```

> It's generally a good idea to put this block about the function definitions; but it could go above or below the var/val definitions.

This function will get called when the class is initialized, so might get executed before we have any UI (depending on the `by lazy` above), so we are not showing any UI errors. Besides, remember that our Service does not use the UI.

The `onSuccess` might look a little weird. Here we are using a method reference shortcut to call `readingList.onNext(it)`. We can't do the same thing for the logs because the parameters don't match.

> You'll also notice that while we do grab the `loadDisposable`, we don't do anything with it. We're grabbing it to keep the IDE from complaining, but there is no concept of application-level cleanup. While there is a `onTerminate` function on our `ReadingListApp`, it will never get called in production.

saveJson

Next, we'll copy our `saveJson` function over and rename.

```
fun save(readingList: ReadingList): Completable {
    return try{
        val writer = FileWriter(File(app.filesDir, BookListService.JSON_FILE))
        app.gson.toJson(readingList, writer)
        writer.flush()
        writer.close()
        this.readingList.onNext(readingList)
        complete()
    }catch(e: Exception){
        Log.d(ReadingListApp.TAG, "Failed to save Json to files/${BookListService.JSON_FILE}",e)
        error(e)
    }
}
```

> You'll use the `io.reactivex.Completable.complete` import.

The one key difference here is that we are adding the `this.readingList.onNext(readingList)` before we return success. This allows us to make sure they are in sync.

You might remember that we also did that in the `init` function above. Let's change the `init` function to call this `save` function instead:

```
.subscribeBy(
    onSuccess = { readingList -> save(readingList) },
    onError = { Log.w(ReadingListApp.TAG, "Error loading books", it) },
    onComplete = { Log.w(ReadingListApp.TAG, "No books found") }
)
```

onActivityResult

Lastly, let's copy the edit section from `onActivityResult` over into a new `edit` function.

```
fun edit(source: Book?, edited: Book): Completable {
    return readingList.value?.let { oldReadingList ->
        // Create a new ReadingList
        val newReadingList = ReadingList(
            title = oldReadingList.title,
            books = oldReadingList.books
                .toMutableList()
                .filterNot {
                    it.title == source?.title
                    && it.author == source?.author
                    && it.year == source?.year
                }.plus(edited)
                .toList()
        )
        // Save the results
        save(newReadingList)
    } ?: error(IllegalArgumentException("Reading List not found"))
}
```

You might be wondering why we are stopping at the `save` call. If you look at the original code, it would next update the UI. Since

the Service doesn't deal with the UI, we will still want to do that in the caller.

Completing the Fragments

EditBookFragment

Open `EditBookFragment`.

Reference our Service

Let's first create a couple variables to hold our new service.

```
private var app: ReadingListApp? = null
private var bookListService: BookListService? = null
```

And then set them up when the Fragment is attached. Make sure to override the correct method as there are two with the same name.

```
override fun onAttach(context: Context) {
    super.onAttach(context)
    override fun onAttach(context: Context) {
        super.onAttach(context)
        app = context.applicationContext as? ReadingListApp
        bookListService = app?.bookListService
    }
}
```

We'll make sure to clean up when we are done.

```
override fun onDetach() {
    super.onDetach()
    bookListService = null
    app = null
}
```

newInstance

We need a way to create an instance of this class with for a specified `Book`. While we *could* use the constructor, that causes problems when the system attempts to recreate the UI (say on low memory or rotation).

Instead, we will create a static method to be called. This is similar to what we did earlier for the `Intent`.

```
companion object {
    val PARAM_SOURCE_BOOK = "${EditBookFragment::class.java}.name::source"

    @JvmStatic
    fun newInstance(app: ReadingListApp, source: Book? = null): EditBookFragment {
        return EditBookFragment().apply {
            arguments = Bundle().also { bundle ->
                source?.let { book ->
                    bundle.putString(PARAM_SOURCE_BOOK, app.gson.toJson(book))
                }
            }
        }
    }
}
```

You'll notice that we added a `@JvmStatic`. This is a Kotlin annotation that tells the compiler to make sure it can be called statically from Java code as well.

We are also passing in the `app` because, as a static context, the instance-level one does not exist yet.

Retrieve the source book

To retrieve the parameter value, we add a couple more variables (outside of the `companion object`).

```
private val paramSource: String? by lazy { arguments?.getString(PARAM_SOURCE_BOOK) }
private val sourceBook: Book? by lazy { app?.gson?.fromJson(paramSource, Book::class.java) }
```

At this point, the `app` does exist.

onCreate

We will want to copy some logic over from `EditBookActivity.onCreate` into `EditBookFragment.onCreateView`.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundl
e?): View? {
    binding = DataBindingUtil.inflate(inflater, R.layout.fragment_edit_book, container,false)
    sourceBook?.let {
        binding.book = EditableBook(source = it)
    } ?: let {
        binding.book = EditableBook()
    }
    return binding.root
}
```

onCreateOptionsMenu

Copy logic from `onCreateOptionsMenu` (note the signature is different)

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.menu_edit_book, menu)
}
```

onActivityCreated

Unlike before, we need to tell the system that we want to show those options, or our save icon will not appear.

Override `onActivityCreated`.

```
override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
    setHasOptionsMenu(true)
}
```

onOptionsItemSelected

Our `onOptionsItemSelected` will use the same *concept* as before, but in a slightly different way.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when(item.itemId) {
        R.id.action_save -> {
            binding.book?.let { editableBook ->
                bookListService?.let { service ->
                    service.edit(
                        source = editableBook.source,
                        edited = editableBook.edited() )
                        .subscribeOn(Schedulers.newThread())
                        .observeOn(AndroidSchedulers.mainThread())
                        .subscribeBy(
                            onComplete = {
                                // It's updated, so we are done
                                activity?.supportFragmentManager?.popBackStack()
                            },
                            onError = { t ->
                                t.message?.let { msg ->
                                    Snackbar.make(binding.root, msg, Snackbar.LENGTH_LONG)
                                        .show()
                                }
                            }
                        )
                }
            }
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

> You'll import `com.google.android.material.snackbar.Snackbar`.

If we successfully edit the book, we `popBackStack`, ie return back to the previous screen. This is basically hitting the back arrow.

BookListFragment

Open your `BookListFragment`.

Reference our Service and Fragment Manager

Like with `EditBookFragment`, let's first create a couple variables to hold our new service. We'll also reference our Fragment Manager since we will be re-using it.

```
private var app: ReadingListApp? = null
private var bookListService: BookListService? = null
private var fragManager: FragmentManager? = null
```

And then set them up when the Fragment is attached. Make sure to override the correct method as there are two with the same name.

```
override fun onAttach(context: Context) {
    super.onAttach(context)
    app = context.applicationContext as? ReadingListApp
    bookListService = app?.bookListService
    fragManager = (context as BookListV2Activity).supportFragmentManager
}
```

Make sure to do our cleanup.

```
override fun onDetach() {
    super.onDetach()
    bookListService = null
    app = null
    fragManager = null
}
```

viewAdapter

Continuing to copy functionality over, we are going to make a new version of our `viewAdapter`.

```
    private val viewAdapter = ReadingListAdapter(
        bookClicked = { book ->
            app?.let { readingListApp ->
                fragManager?.let {
                    it.beginTransaction()
                    .replace(R.id.container, EditBookFragment.newInstance(
                        app = readingListApp,
                        source= book))
                    .addToBackStack(null)
                    .commit()
                }
            }
        }
    )
```

> You'll notice that the IDE suggests removing the `let` on the `fragManager?.let` line. While it is possible to do so, that results in adding a `?` before every period. Feel free to try it and see.

Since this *could* be called before the `app` is configured, we have to account for that.

Before, we used `.add` to put our current fragment into the `R.id.container`. Now we will use `.replace` to swap it out for the editing page. We are also using `.addToBackstack` so that our `popBackStack` will allow us to return to this page.

onCreate

We'll continue copying from `BookListActivity.onCreate` into `BookListFragment.onCreateView`.

Our recycler looks the basically the same. We just need to deal with the Context differently.

```
binding.recycler.apply {
    setHasFixedSize(true)
    activity?.let {
        layoutManager = LinearLayoutManager(it)
        addItemDecoration(CustomDivider(context = it as Context))
    }
    adapter = viewAdapter
}
```

For the FAB, we want to mimic what we did for our `bookClicked` listener above; but without a 'source' book.

```
binding.fab.setOnClickListener {
    app?.let { readingListApp ->
        fragManager?.let {
            it.beginTransaction()
            .replace(R.id.container, EditBookFragment.newInstance(app = readingListApp))
            .addToBackStack(null)
            .commit()
        }
    }
}
```

Monitoring books being loaded

To monitor for books being loaded, we want to add our disposal variable up above first.

```
private var disposable: Disposable? = null
```

Then watch the `BehaviorSubject` for changes and update our `viewAdapter` inside `onCreateView`.

```
bookListService?.let { service ->
    disposable = service.readingList
        .toFlowable(BackpressureStrategy.LATEST)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe {
            Log.i(ReadingListApp.TAG, "${it.books.size} books loaded")
            viewAdapter.readingList = it
        }
}
```

And lastly, make sure to dispose of it when we don't need it anymore.

```
override fun onDetach() {
    super.onDetach()
    disposable?.dispose()
    bookListService = null
    app = null
    fragManager = null
}
```

Deploy your application and make sure everything is working.

#

- Current Repo: v1-Fragments
- Continue to Animation -->

Home.md

Table of Contents

1. Introduction: In this chapter, we will discuss some background on Android and install the tools.
2. Basic UI: Generate a basic UI project, learn how the pieces fit together and deploy it to the emulator.
3. Data: Learn how to load, save and convert data.
4. Unit Testing: Learn about test-driven development.
5. Advanced UI: Add some common UI elements, learn about databinding and multi-page applications.
6. Debugging: Learn about debugging and logging.
7. Threading: Do some processing on the background thread then update the UI thread.
8. Drawing: Information about custom drawing.
9. Fragments: Convert our Activity-based UI to a Fragment-based UI.
10. Animation: Explore some basic animations.
11. Networking: Learn how to send and retrieve data on the network.

#

- Current Branch: v1

Introduction.md

Goal

In this chapter, we will discuss some background on Android and install the tools.

Table of Contents

- Course Introduction
- Background
  - Android Versioning
  - AOSP vs OEM
  - JDK Compatability
  - Expect Change
- Tools
  - Android Studio
  - Android Emulator
  - Command Line

Course Introduction

TBD

Background

Android Versioning

Android is versioned three different ways. Each of them have their own uses, and you will need to understand each of them. This information can be found at Wikipedia if you need to reference it later.

Code name	Version number	Linux kernel version[3]	Initial release date	API level
(No codename)[4]	1.0	?	September 23, 2008	1
Petit Four[4]	1.1	2.6	February 9, 2009	2
Cupcake	1.5	2.6.27	April 27, 2009	3
Donut[5]	1.6	2.6.29	September 15, 2009	4
Eclair[6]	2.0 – 2.1	2.6.29	October 26, 2009	5 – 7
Froyo[7]	2.2 – 2.2.3	2.6.32	May 20, 2010	8
Gingerbread[8]	2.3 – 2.3.7	2.6.35	December 6, 2010	9 – 10
Honeycomb[9]	3.0 – 3.2.6	2.6.36	February 22, 2011	11 – 13
Ice Cream Sandwich[10]	4.0 – 4.0.4	3.0.1	October 18, 2011	14 – 15
Jelly Bean[11]	4.1 – 4.3.1	3.0.31 to 3.4.39	July 9, 2012	16 – 18
KitKat[12]	4.4 – 4.4.4	3.10	October 31, 2013	19 – 20
Lollipop[13]	5.0 – 5.1.1	3.16	November 12, 2014	21 – 22
Marshmallow[14]	6.0 – 6.0.1	3.18	October 5, 2015	23
Nougat[15]	7.0 – 7.1.2	4.4	August 22, 2016	24 – 25
Oreo[16]	8.0 – 8.1	4.10	August 21, 2017	26 – 27
Pie[17]	**9.0**	4.4.107, 4.9.84, and 4.14.42	August 6, 2018	28
Legend: Old version	Older version, still supported	**Latest version**		

- Code Name is usually referred to when talking about Android versions. For example, "did your phone get the Oreo update"
- Version Number is used when talking about compatibility. For example, "our app supports 4.4 - 8.1"
- API Level is a technical detail. For example, if you look at the API documentation, you can filter on the top-left by the API level you support.

The other aspect is version distribution. You can review the most recent statistics on the Distribution Dashboard.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.1%
4.2.x		17	1.5%
4.3		18	0.4%
4.4	KitKat	19	7.6%
5.0	Lollipop	21	3.5%
5.1		22	14.4%
6.0	Marshmallow	23	21.3%
7.0	Nougat	24	18.1%
7.1		25	10.1%
8.0	Oreo	26	14.0%
8.1		27	7.5%

This chart does not show any version that has less than 0.1% user base over the last week. When trying to determine which API levels you need to support, this can come in very handy if you don't have any actual user analytics.

For example, in the chart above, you can see that nothing below API 22 has 10% market share. If you look closely, you will also notice that the newest release, API 28 is not listed either. That is because not enough devices have that version yet.

AOSP vs OEM

You might be asking yourself why the latest version is not on all devices.

Let's look at a hypothetical situation to better understand how this process works.

Google releases the latest version as part of the Android Open Source Project. Some proprietary bits are not released, but we will ignore that for now. The AOSP is a set of git repositories.

An OEM (Samsung, Motorola, etc) will then make their own changes to the source code. This would be proprietary changes to frameworks, additional applications, et cetera.

The OEM then makes additional changes for the carrier (AT&T, Verizon, etc). This would be disabling certain competing features (like Tethering or Visual Voicemail), making changes specific to their networks (usually at the device driver) and modifying/adding applications to give a common look and feel.

The device then has to be tested and certified. This process usually takes awhile.

Then the device has to be marketed, usually to fit in with a large holiday like Black Friday or Christmas.

At this point you are probably thinking to yourself that you understand why it would take so long to get a device, but why does it take so long to update a device you already have?

Although updates can be sent OTA (Over The Air) to your device, they often have to go through most, if not all, of those steps. Each company in the process might initiate an update - but it still needs to be integrated and tested before it is released.

On top of that, there is sometimes a newer device getting ready to be released. Older devices tend to fall by the wayside. While not a hard and fast rule, you should not expect any hardware to get more than two *major* OS updates.

JDK Compatability

Rather than go into a long and complicated history of which version of Java was supported on which version of Android, let's focus

on development you would do right now.

The currently released version of Java (as of Sept 2018) is JDK11.

Android supports a large subset of JDK8. According to Use Java 8 language features, many of the features are only supported on API24 and above. Some functionality, like `MethodHandle.invoke` require API26.

It is important to note that while that page states that it supports all JDK7 features, there are some JDK5 features (like `ServiceLoader`) that are not properly supported due to the way Android applications are packaged.

If you are coming from a Java background, your best option today is to plan on Java 8, but be prepared to tweak things that don't quite work correctly.

> If you are writing annotation processors, you could get away with writing them in Java 11 - and have those processors output Java 8 code.

Expect Change

Every year Google holds it's annual developer conference, Google I/O. Usually, new devices and OS versions are announced or hinted at. New APIs are announced and others deprecated. Things that were encouraged yesterday are discouraged today, and a new way is shown.

If you look at the version history above, you will see that there is basically a new version every year. With each new version comes (sometimes incompatible) changes.

Sometimes these changes are cosmetic and change how the application looks. Sometimes if affects performance, security or privacy.

No one is expected to implement all of the latest features immediately - but it is a good idea to be aware of them. Not only could they help you develop your application; but sometimes they are mandatory changes.

Expect that the APIs and ecosystems and even the core frameworks will change over time. When designing your application, try to be modular so that you can swap things out. Try to hardcode as little as possible, so that it is easier to update. Try to test your application on the newer versions of the OS as soon as possible, so you know if something will break.

Tools

Android Studio

JetBrains launched the Java IDE IntelliJ in 2001. Since that time, they have extended their portfolio to 22 products, including WebStorm, RubyMine, PyCharm and AppCode; to name just a few.

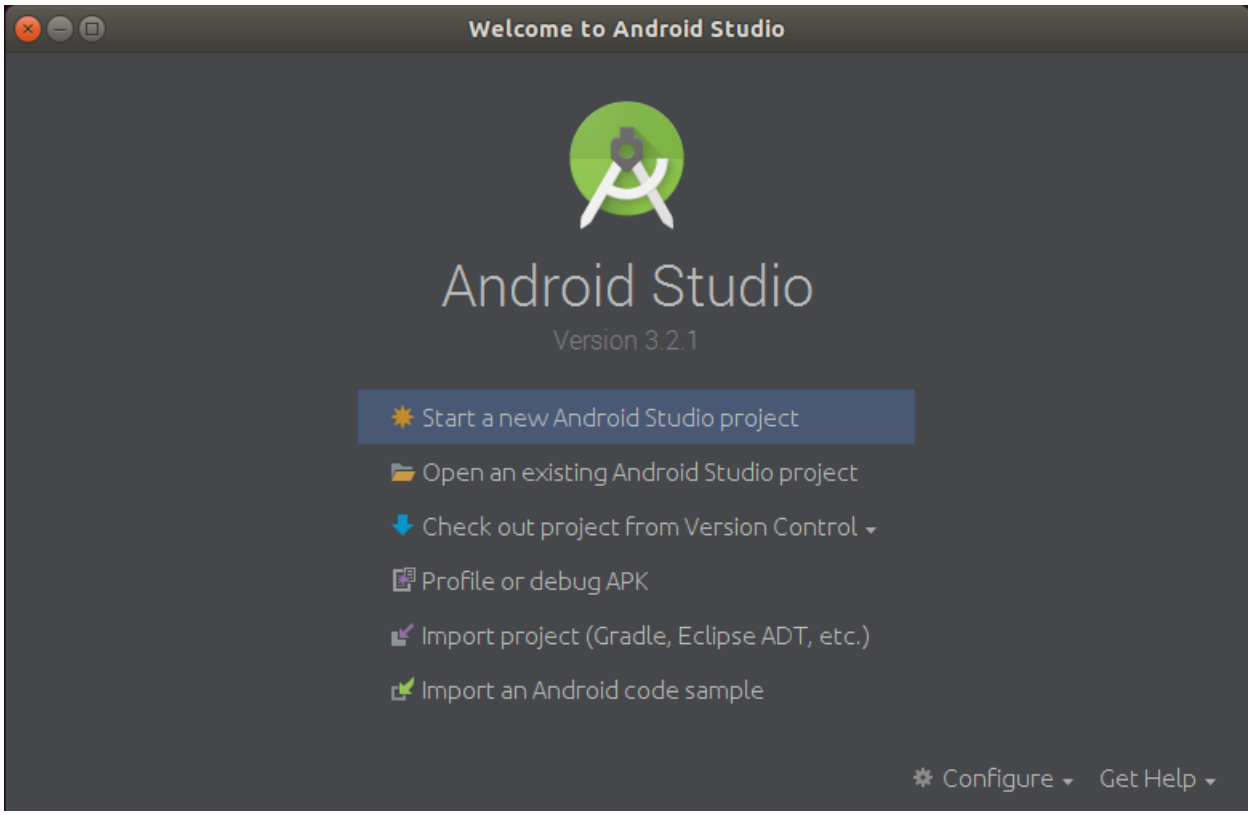
In 2013, Google worked with JetBrains to develop Android Studio. If you are familiar with their other products, Android Studio will feel similar. While the other IDEs are available to install and update via the JetBrains Toolbox, Android Studio needs to be installed and updated manually.

Installation

Android Studio is available for Windows, Mac and Linux. Go to the Android Studio Developers Page and scroll to the bottom of the page. There you will find the current system requirements.

If your system meets those requirements, go back to the top of the page, and click the button to install the recommended version. It should be the 64-bit full IDE install.

Once installed, launch Android Studio.



> If you need to start Android Studio from the command line (say through a shell script), you can cd into the `bin` subdirectory of the install and run `./studio.sh` or `studio.bat`. Most people should not need to do so.

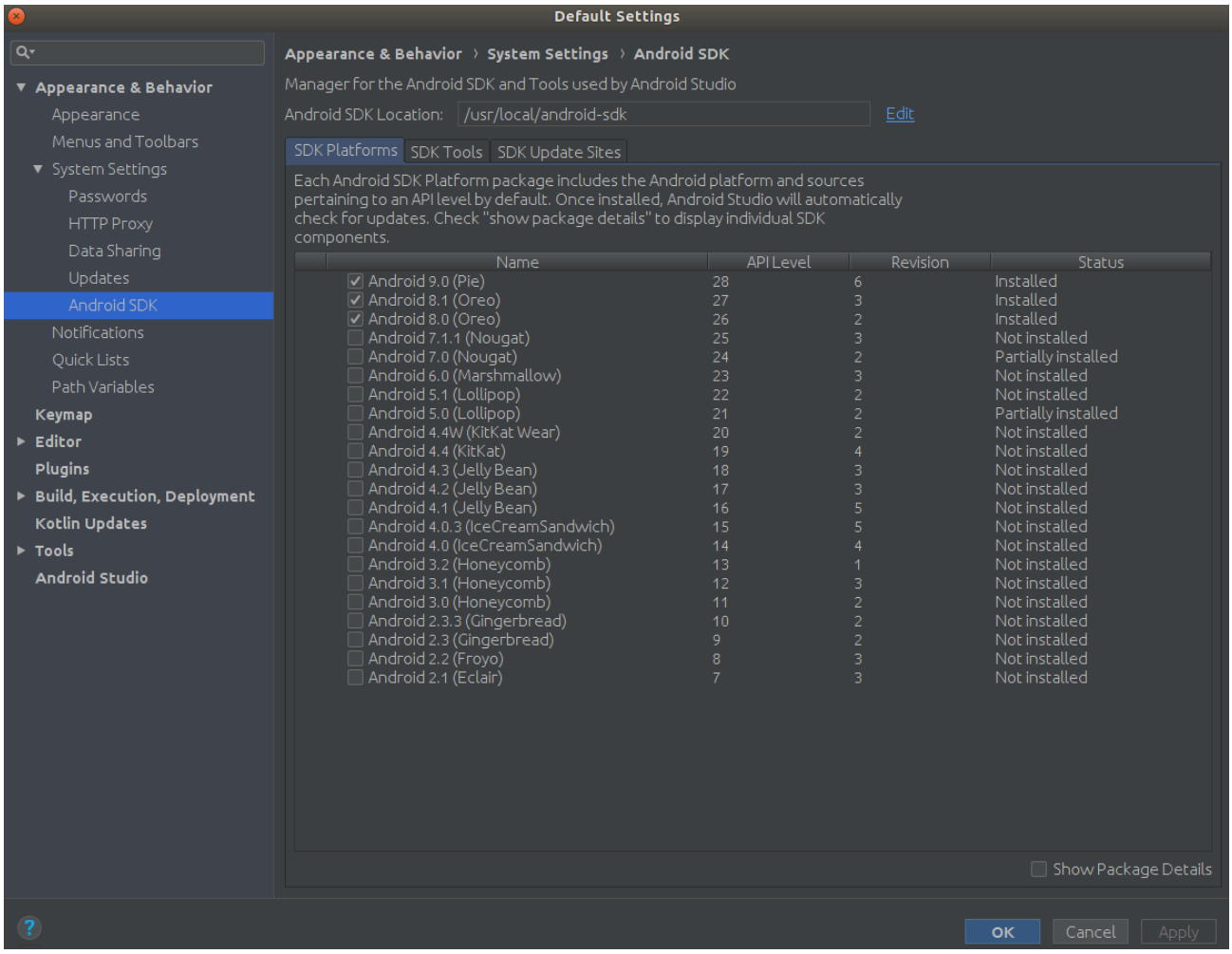
Update the SDK Manager

On the bottom right of the dialog, click on `Configure` then `SDK Manager`.

The SDK Manager is where you will manage the core Android OS versions and tools available to the IDE.

SDK Platforms

The dialog starts on the `SDK Platforms` tab. Your window will look something like this:

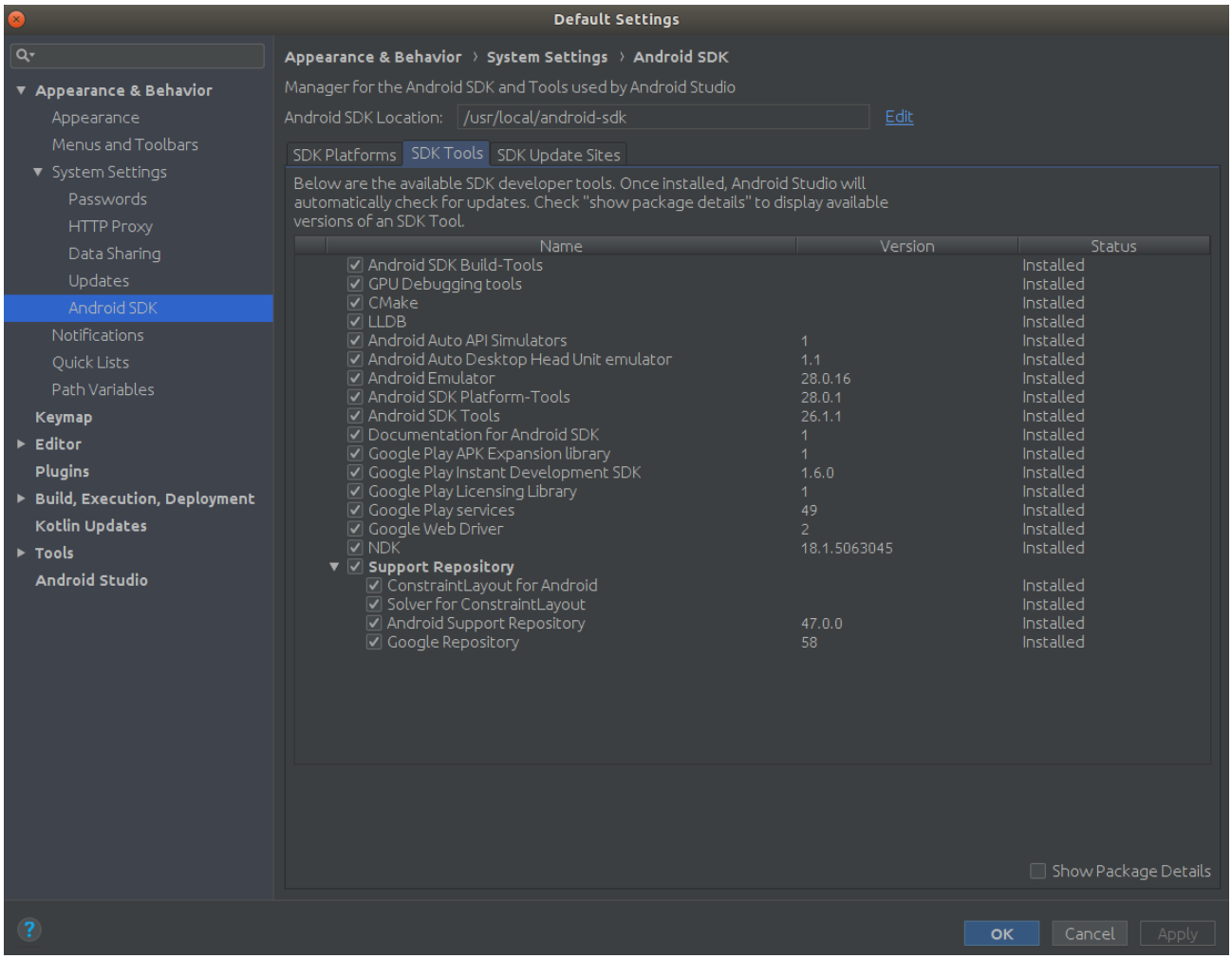


For production applications, you would want to install each version you support so that you can test customer issues. For now, just enable the `Android 9.0 (Pie)` line if it is not already enabled and click `Apply`.

> If it is already installed when you open this page (as indicated in the far right column), the `Apply` button will be commented out and you can skip this step. This might be the case if you already had the IDE installed.

SDK Tools

Once that has finished, we will move on to the `SDK Tools` tab. Click it. Your window will look something like this:



For production development, I would recommend enabling as many of these as possible. For today, make sure at least the following are enabled:

- Android SDK Build-Tools
- Android Emulator
- Android SDK Platform-Tools
- Android SDK Tools
- Documentation for Android SDK
- Google Play services
- ConstraintLayout for Android
- Solver for ConstraintLayout
- Android Support Repository
- Google Repository

When you are done, click `OK` to return to the main dialog.

Android Emulator

Another useful tool is the AVD (Android Virtual Device) Manager. It allows you to setup and manage Emulators. Since the AVD Manager can only be run from a working project, we will cover that in the next chapter.

Command Line

While we are going to be building from the IDE, it is also possible to build from the command line.

#

- Current Repo: v1-Introduction
- Continue to Basic UI -->

Networking.md

Goal

Learn how to send and retrieve data on the network.

Table of Contents

- Updating our Models
- RESTful Server
- Retrofit
    - Defining our API
    - Calling our API
- Improvements

Earlier we were looking at how to know which book is being edited. We briefly touched on the idea that we could add identifiers to the data so that we knew which book it was.

While we chose to not take that approach early on, it becomes imperative when we start talking to REST servers. When using RESTful services, we identify the resource element by an identifier, whether that is `tom` or `12345`, usually the latter.

Updating our Models

To better support this functionality, we will update our existing application to have IDs. Since we have broken our nested json into multiple models, it will make it easier to follow.

First, download this file to replace your current `app/src/main/assets/BooksAndAuthors.json`. Make sure to overwrite the original file. This version of the data has `id` fields embedded for each list, book and author.

Clear your application cache and deploy your application.

Although we changed the json source, the models don't yet know about the change.

Add the *optional* `id` parameter to the `ReadingList` model. We'll also want to add a `bookIds` field, for talking with the server.

```
package com.aboutobjects.curriculum.readinglist.model

data class ReadingList(
    val title: String? = null,
    val books: List<Book> = emptyList(),
    val bookIds: List<Int> = emptyList(),
    val id: Int? = null
)
```

Why are these values optional?

If we were to create a *new* `ReadingList`, we don't yet know what any of the values should be.

For the `Book` model, we'll add both the `id` and `authorId` parameters.

```
package com.aboutobjects.curriculum.readinglist.model

data class Book(
    val title: String? = null,
    val author: Author? = null,
    val authorId: Int? = null,
    val year: String? = null,
    val id: Int? = null
)
```

And for the `Author` model, just add the `id`

```
val id: Int? = null
```

Deploy your application. Everything still OK?

Let's open your `BookListService.edit` function and change our `filterNot` criteria. Since we have an identifier now, we no longer need all of that other criteria. There should only ever be one match.

```
fun edit(source: Book?, edited: Book): Completable {
    return readingList.value?.let { oldReadingList ->
        // Create a new ReadingList
        val newReadingList = ReadingList(
            title = oldReadingList.title,
            books = oldReadingList.books
                .toMutableList()
                .filterNot { it.id == source?.id}
                .plus(edited)
                .toList()
        )
        // Save the results
        save(newReadingList)
    } ?: error(IllegalArgumentException("Reading List not found"))
}
```

> This function will change a bit when we start talking to the network.

Clear your app storage, re-deploy your app and make sure that you can still edit books.

RESTful Server

At this point, the instructor will start up this RESTful server for us to talk to.

We will want to add a buildConfigField to represent the server we are going to talk to.

Open your module-level `build.gradle` and add a new `buildConfigField`. Note that although we are putting ours inside the `default` block, you could easily point to one server for production and another for your debug application.

```
buildConfigField "String", "CURRICULUM_SERVER", '"http://192.168.122.1:4567"'
```

The specific address you use will come from the instructor.

> If you are running the `ao-android-curriculum-server` yourself, launch http://localhost:4567/hello and it will tell you what address to use.

Sync your project.

Child IDs vs Full Population

It is quite common for a RESTful server to only return the IDs of children so as to avoid sending unnecessary information. For an example of this, visit `<CURRICULUM_SERVER>/lists` in your browser.

> IE using the IP address in the example above, `http://192.168.122.1:4567/lists`

You notice how each book is represented by only an ID? Let's look at a specific book. Visit `<CURRICULUM_SERVER>/books/12`.

Here we see that the author is represented by only an ID. `<CURRICULUM_SERVER>/authors/10`

If you were to retrieve the entire Reading List for display, you would make 1 call for the Reading List, 18 more calls for the Books, and 15 more calls for the Authors.

34 network calls for a single UI display is a bit much, don't you think?

When you encounter situations like that, make sure to call attention to it. The backend teams can likely add functionality for specific use cases.

For example, visit `<CURRICULUM_SERVER>/books/12?full=true` and notice how the Author is populated?

Similarly, you can visit `<CURRICULUM_SERVER>/lists?full=true` to view everything.

Obviously, if you have millions of Reading Lists, that would not be acceptable either. You have to draw a balance between low bandwidth / high performance calls and the number of calls required.

The main API page located at `<CURRICULUM_SERVER>/hello` will indicate which API endpoints support the `?full=true` method.

Retrofit

While Java and Android both have built-in libraries for doing HTTP and network calls, using them to make RESTful calls can be quite tedious as you have to implement a lot of boilerplate code.

Wouldn't it be nice if you could skip all that and just define the API contract?

That's exactly what Retrofit allows you to do.

In your project-level `build.gradle`, let's add a new `retrofitVersion`.

```
project.ext {
    sampleVariable = "This is a sample variable"
    retrofitVersion = '2.5.0'
}
```

Then in your module-level `build.gradle` add the dependencies.

```
implementation "com.squareup.retrofit2:retrofit:${rootProject.ext.retrofitVersion}"
implementation "com.squareup.retrofit2:adapter-rxjava2:${rootProject.ext.retrofitVersion}"
implementation "com.squareup.retrofit2:converter-gson:${rootProject.ext.retrofitVersion}"
```

Sync your project.

AndroidManifest

For our application to be able to talk to the internet, we need to ask for permission.

Open your `AndroidManifest.xml` and add a new permission inside the `manifest` tag, before the `application` tag. Also add the `android:usesCleartextTraffic` to the `application` tag since our demo is not using HTTPS.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.aboutobjects.curriculum.readinglist">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
            android:usesCleartextTraffic="true">
```

Defining our API

CurriculumAPI

Right-click on your `service` package and add a new `CurriculumAPI` with a Kotlin type of `interface`.

This class will just define the interface of our API. More information about the API can be found here.

Simple GETs

First, let's define our simplest GET calls.

```
@GET("/lists")
fun getReadingLists( @Query("full") full: String = "true" ): Single<List<ReadingList>>

@GET("/books")
fun getBooks( @Query("full") full: String = "true" ): Single<List<Book>>

@GET("/authors")
fun getAuthors(): Single<List<Author>>
```

These methods take no required parameters (you can choose to turn off the full ID conversion we discussed earlier) and return a `Single<List<MODEL>>` of our models. The `Single` contract specifies that there can only be one result; which in our case is a List of models.

> You'll notice that the `full` parameter is not an option on the `/authors` endpoint, per the specification of our server.

GET by id

Next up, we'll look at the GET calls that require an identifier to be passed.

```
@GET("/lists/{id}")
fun getReadingList(
    @Path("id") id: Int,
    @Query("full") full: String = "true"
): Single<ReadingList>

@GET("/books/{id}")
fun getBook(
    @Path("id") id: Int,
    @Query("full") full: String = "true"
): Single<Book>

@GET("/authors/{id}")
fun getAuthor(@Path("id") id: Int): Single<Author>
```

These look the same as before, but we are passing in the extra `@Path` element to replace the named parameter in the url. We are also only returning a single model, rather than a list of them.

Create with PUT

Creating objects is a little different.

```
@PUT("/lists/create")
fun createReadingList(
    @Body model: ReadingList,
    @Query("full") full: String = "true"
): Single<ReadingList>

@PUT("/books/create")
fun createBook(
    @Body model: Book,
    @Query("full") full: String = "true"
): Single<Book>

@PUT("/authors/create")
fun createAuthor(@Body model: Author): Single<Author>
```

Here, we pass our source model, which Gson converts and uses as the body of the HTTP PUT call to the server. Once again, the `/authors/create` endpoint does not support the `full` parameter.

> These endpoints are the reason we added `ReadingList.bookIds` and `Book.authorId` to our models.

Update with POST

Similarly, we might want to update an existing model on the server.

> For these, the server asks that we use POST. If the server were to require PATCH, we would have to make the client match.

```
@POST("/lists/update/{id}")
fun updateReadingList(
    @Path("id") id: Int,
    @Body model: ReadingList,
    @Query("full") full: String = "true"
): Single<ReadingList>

@POST("/books/update/{id}")
fun updateBook(
    @Path("id") id: Int,
    @Body model: Book,
    @Query("full") full: String = "true"
): Single<Book>

@POST("/authors/update/{id}")
fun updateAuthor(
    @Path("id") id: Int,
    @Body model: Author
): Single<Author>
```

This time we are using the all the different parts from above together into a single HTTP POST call.

Remove with DELETE

What if we wanted to delete an item?

```
@DELETE("/lists/delete/{id}")
fun deleteReadingList(@Path("id") id: Int): Completable

@DELETE("/books/delete/{id}")
fun deleteBook(@Path("id") id: Int): Completable

@DELETE("/authors/delete/{id}")
fun deleteAuthor(@Path("id") id: Int): Completable
```

This time we just use the id for an HTTP DELETE call. You might notice that we are using a `Completable` instead of a `Single`? That's because the server does not return any models to us. Actually, it just returns the word "ok".

Find with Forms

We have one last method

```
@FormUrlEncoded
@POST("/authors/find")
fun findAuthor(
    @Field("firstName") firstName: String?,
    @Field("lastName") lastName: String?
): Single<Author>
```

This time we are doing an HTTP POST call, but as `application/x-www-form-urlencoded`. This would be the same as if you had a form on a webpage and those fields were being populated by the user.

You might also notice that both of them are specified as `String?` instead of `String` this time. The server endpoint specifies that the parameters on this call are each optional; whereas on the other calls they are required.

> As a side note, you would not normally implement every single method on your REST server. We are only showing these for completeness.

CurriculumService

Right-click on your `service` package and add a new `CurriculumService` with a Kotlin type of `class`. Add our `ReadingListApp` as a parameter to it.

```
package com.aboutobjects.curriculum.readinglist.service

import com.aboutobjects.curriculum.readinglist.ReadingListApp

class CurriculumService(val app: ReadingListApp) {
}
```

This class will provide the concrete implementation of our API.

Let's add a couple variables to tie things together.

```
private val retrofit = Retrofit.Builder()
    .baseUrl(BuildConfig.CURRICULUM_SERVER)
    .addConverterFactory(GsonConverterFactory.create(app.gson))
    .addCallAdapterFactory(RxJava2CallAdapterFactory.createWithScheduler(Schedulers.io()))
    .build()

private val backend = retrofit.create(CurriculumAPI::class.java)
```

> You'll use the `com.aboutobjects.curriculum.readinglist.BuildConfig` import.

Here, we build a new private `Retrofit` instance using our previously defined server endpoint. We also specify that POJOs should be converted using our Gson configuration and that all network calls should be bound to the io thread.

We then create a `backend` variable that matches our API contract we built, using Retrofit.

> You might be wondering why we aren't implementing the API directly. Maybe we want to add debug logging, don't want to expose some of the methods, or maybe we want change the parameters...
>
> That being said, you *could* make that backend variable publicly available directly and remove the need for this class entirely. It just removes your ability to tweak or mock behavior.

Let's add a function to retrieve our `ReadingList`.

```
fun getReadingList(id: Int): Single<ReadingList> {
    return backend.getReadingList(id = id)
}
```

For creating the `Author` we create a local `Author` as a model; which the server will return with a populated `id`.

```
private fun createAuthor(author: Author): Single<Author> {
    return backend.createAuthor(
        model = Author(
            firstName = author.firstName,
            lastName = author.lastName
        )
    )
}
```

As a reminder, this is why the `id` is optional.

But, what if that author already exists? Let's add another method to double-check that first.

```
private fun findOrCreateAuthor(source: Author): Single<Author> {
    return backend.findAuthor(
        firstName = source.firstName,
        lastName = source.lastName
    ).onErrorResumeNext{
        createAuthor(author = source)
    }
}
```

We will attempt to find the author by name. If that fails, then we will call our `createAuthor` function.

Now we can add a function to create a new book.

```
fun createBook(source: Book): Single<Book> {
    return findOrCreateAuthor(source = source.author ?: throw IllegalArgumentException("Source Autho
r missing"))
        .flatMap { author ->
            backend.createBook(
                model = Book(
                    title = source.title,
                    year = source.year,
                    authorId = author.id ?: throw IllegalArgumentException("Source Author ID missing
")
                )
            )
        }
}
```

We will rely on our `findOrCreateAuthor` to ensure that the Author already exists on the backend before we make the backend call to create the book with the Author's ID. If there is no Author specified in our `source` book, we'll just bail as we can't do anything without that required parameter.

If you recall, our UI currently allows the user to change the Author's name in the same screen as the Book's title and year. A more robust design might require those to be edited on completely different screens. For our current design, we need to take additional precautions.

Let's add a method to our `Author` model.

```
fun isSameAs(other: Author?): Boolean {
    return displayName().equals(other?.displayName())
}
```

Back in `CurriculumService`, we can now write the `updateBook` function and take advantage of the new `isSameAs`.

```
fun updateBook(source: Book, edited: Book): Single<Book> {
    if (source.id == null) {
        throw IllegalArgumentException("Source Book ID missing")
    }

    if (source.author?.id == null) {
        throw IllegalArgumentException("Source Author missing")
    }

    return when {
        // source author HAS id, but edited author does NOT
        source.author.isSameAs(edited.author) -> Single.just(source.author)
        // maybe we typed a different existing authors' name?
        edited.author != null -> findOrCreateAuthor(edited.author)
        // fail-safe
        else -> Single.just(source.author)
    }.flatMap { author ->
        backend.updateBook(
            id = source.id,
            model = Book(
                id = source.id,
                title = edited.title,
                year = edited.year,
                authorId = author.id ?: throw IllegalArgumentException("Source Author ID missing")
            )
        )
    }
}
```

We start off by doing some sanity checks. If the source id or source author id are missing, we don't make any backend calls. We could have done those both using let, but in this case doing the fail-fast first is much cleaner.

We then try to determine who the `Author` is. If the `source` and `edited` books both list the same author (remember, same display name) - then we will use the `source` version since it has an `id`. The edited version would not have one yet.

If they do not match, we will call our `findOrCreateAuthor` function using the new `edited` author information. If the `Author` exists on the server, we will use it. If not, we will create it.

As a final sanity check, and because you should *ALWAYS* have an `else`/`default` clause, we will default to using the original author if the new one is missing.

Once we have determined the author, then we will make a call to the server to update the book. There is another `throw` in there; but it is just a sanity check. Our `authorId` should not be null at this stage.

We're also going to need a way to update our `ReadingList` with the new `Book`s. The backend call requires that we pass a list of `bookIds`, so we need to iterate over our book list and replace the `Book` entries with an `Int` id.

```
fun addBookToReadingList(readingList: ReadingList, book: Book): Single<ReadingList> {
    if (book.id == null) {
        throw IllegalArgumentException("Book ID missing")
    }

    return backend.updateReadingList(
        id = readingList.id ?: throw IllegalArgumentException("Reading List ID missing"),
        // Send a version with just the IDs
        model = ReadingList(
            id = readingList.id,
            title = readingList.title ?: "",
            bookIds = readingList.books
                .mapNotNull { it -> it.id }
                .toMutableList()
                .plus(book.id)
                .distinct()
                .sorted()
                .toList()
        )
    )
}
```

Once we have converted the existing `List<Book>` to a `List<Int>`, we make the list mutable (ie: `MutableList<Int>`) and add the requested `Book` to the list as well. We make sure there are no duplicates (again, another sanity check) and re-sort them. Finally, we convert it back to a non-mutable `List<Int>` that the backend API requires.

> As you can see, you can customize how the backend calls are made rather than a 1:1 match.

Calling our API

Let's hook our new service up. If we were using dependency injection (like Dagger) we would be doing this a bit different; but for now, let's add it to our `ReadingListApp`.

```
val curriculumService: CurriculumService by lazy {
    CurriculumService(app = this)
}
```

Loading from the Server

Open our `BookListService` and add a new method

```
private fun loadFromServer(): Maybe<ReadingList> {
    return app.curriculumService
        .getReadingList(id = 0)
        .toMaybe()
        .onErrorComplete()
}
```

Since we aren't using any `FileReader` this time, it won't throw any Exceptions that we can catch. If there is a problem, it will come back in the `onError` callback.

And since we want to gracefully fall-over to something else rather than bailing out, we tell it to treat a failure to read the server the same way it would treat reading an empty cache file.

Now we can change our `init` block... maybe like:

```
init {
    val loadDisposable = loadFromServer()
        .switchIfEmpty(loadFromFiles())
        .switchIfEmpty(loadFromAssets())
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeBy(
            onSuccess = { readingList -> save(readingList) },
            onError = { Log.w(ReadingListApp.TAG, "Error loading books", it) },
            onComplete = { Log.w(ReadingListApp.TAG, "No books found") }
        )
}
```

So here, we will try to load from the server first. If that fails, we will try to load our cached copy. If that ALSO fails, we will try to load the original version shipped with the application.

To understand what this looks like, I added a couple additional logs, cleared the cache and started the application without the server running:

```
12-04 08:39:59.442 D/ReadingListApp( 6240): Reading from SERVER
12-04 08:39:59.497 D/ReadingListApp( 6240): Reading from FILES
12-04 08:39:59.499 D/ReadingListApp( 6240): Failed to load Json from files/BooksAndAuthors.json
12-04 08:39:59.499 D/ReadingListApp( 6240): java.io.FileNotFoundException: /data/user/0/com.aboutobj
ects.curriculum.readinglist/files/BooksAndAuthors.json (No such file or directory)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at java.io.FileInputStream.open0(Native Method)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at java.io.FileInputStream.open(FileInputStream.java
:231)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at java.io.FileInputStream.<init>(FileInputStream.ja
va:165)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at java.io.FileReader.<init>(FileReader.java:72)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at com.aboutobjects.curriculum.readinglist.service.B
ookListService.loadFromFiles(BookListService.kt:62)
12-04 08:39:59.499 D/ReadingListApp( 6240):     at com.aboutobjects.curriculum.readinglist.service.B
ookListService.<init>(BookListService.kt:29)
...clip...
12-04 08:39:59.499 D/ReadingListApp( 6240): Reading from ASSETS
12-04 08:39:59.768 I/ReadingListApp( 6240): 18 books loaded
```

We can see that it tried to read from the (temporarily down) server, then from the (not yet written) cache then finally from the assets copy.

> You might be asking why we'd go through so much effort? Why not just rely on the server copy? This is just to give you an idea of what might be required to support "Offline Mode". You need the local cache in order to read/write changes when the app is offline. You need the asset copy in order to seed the app if it is offline when they first launch it. It could also be useful if they chose to 'factory reset' your application.

Saving to the Server

Currently, our application does have the concept of creating new reading lists. The API supports it, so that is functionality you could add (maybe with some additional UI elements). For now, let's focus on what we need to do to update a single book entry on the server.

Our application currently supports functionality to both "Add Book" and "Edit Book". Both functions do their processing through `BookListService.edit`. The "Add Book" scenario just has the `source` set to null.

As such, that function is the one we will hijack to change the behavior.

To re-cap, this is the current *offline*-only function (from above):

```
fun edit(source: Book?, edited: Book): Completable {
    return readingList.value?.let { oldReadingList ->
        // Create a new ReadingList
        val newReadingList = ReadingList(
            title = oldReadingList.title,
            books = oldReadingList.books
                .toMutableList()
                .filterNot { it.id == source?.id}
                .plus(edited)
                .toList()
        )
        // Save the results
        save(newReadingList)
    } ?: error(IllegalArgumentException("Reading List not found"))
}
```

Let's replace it with a new *online*-only function:

```
fun edit(source: Book?, edited: Book): Single<Book> {
    return when (source) {
        null -> app.curriculumService.createBook(source = edited)
        else -> app.curriculumService.updateBook(source = source, edited = edited)
    }.flatMap { book ->
        app.curriculumService
            .addBookToReadingList(
                readingList = readingList.value ?: throw IllegalArgumentException("Reading List not
loaded"),
                book = book
            )
            .map { it -> Pair(it, book) }
            .doOnSuccess { (readingList, _) -> save(readingList) }
    }.map { (_, book) ->
        book
    }
}
```

If the `source` book is null, we call our `createBook` method. If it is not, we call `updateBook`. Both of those methods return a `Single<Book>`.

We take that result and tell the server to add the book to the reading list. Without this step, the new book would exist, but not be associated with our reading list.

Once we get a response on updating the reading list, we make sure to save it to our local cache. Finally, we return that book to the caller.

> You might be wondering about the `Pair(it, book)`, `(readingList, _)` and `(_, book)`. `Pair(it` *aka readingList*`, book)` is being used to return 2 values instead of just 1. Normally, only the `readingList` would get carried forward, but we need the `book` on a future step. Whenever we don't need to use a particular value, it can be replaced with `_` to represent "don't know, don't care". In this case, `save` only cares about the `readingList` and the `map` only cares about the book.

As a final change, since our old function returned `Completable` and the new one returns `Single<Book>`, let's update our `EditBookFragment.onOptionsItemSelected`. Just change the word `onComplete` to `onSuccess`.

Deploy your application and edit some books.

The other students in the classroom are currently sharing the same reading list as you. Are you seeing each others books?

Improvements

What improvements could be made to the application?

- Maybe a periodic refresh in case you didn't edit any books?
- Maybe a personalized reading list rather than a shared one?
- Maybe a share button so you could email the list to a friend?

- How about a separate page to edit the Author directly?
- Maybe you would like to specify a sort order or filtering rules?

#

- Current Repo: v1-Networking
- <-- Return Home

Threading.md

Goal

Do some processing on the background thread then update the UI thread.

Table of Contents

- Main Thread
    - ANR
- Background Thread
    - CalledFromWrongThreadException
        - Handler
- AsyncTask
- Anko
- Rx

Main Thread

You might recall that our `onActivityResult` is being called on the main (UI) thread.

Let's discuss for a moment what that means.

Each thread in the system is given a name. We saw some of these when we reviewed Debugging.

`main` is the name given to the primary thread responsible for the UI. There is a queue running on it, and it just keeps executing everything in that queue in order. When you ask for some activity to be opened, an image to be drawn, or touch the screen - each of these things add new events to that queue to be processed. The more items in that queue, the slower and less responsive your application (and in fact the phone) *feels*, because it takes longer and longer to finally get to that draw call you requested or to respond to that touch event.

ANR

An ANR (or Application Not Responding) happens when the delay is so great that your application does not respond to a request in a timely fashion (generally about 5 seconds).

Let's introduce an arbitrary delay so that you can see how this happens.

Open `BookListActivity`.

We'll add the code change inside `saveJson` to mimic a long write operation.

Inside the `try` block, we will introduce a 20-second delay (it's taking a VERY long time to write!).

```
private fun saveJson(readingList: ReadingList) {
    try{
        val writer = FileWriter(File(filesDir, JSON_FILE))
        app.gson.toJson(readingList, writer)
        writer.flush()
        writer.close()

        TimeUnit.SECONDS.sleep(20L)
    }catch(e: Exception){
        Log.d(ReadingListApp.TAG, "Failed to save Json to files/$JSON_FILE",e)
    }
}
```

> Use the `java.util` import, not the `icu` import.

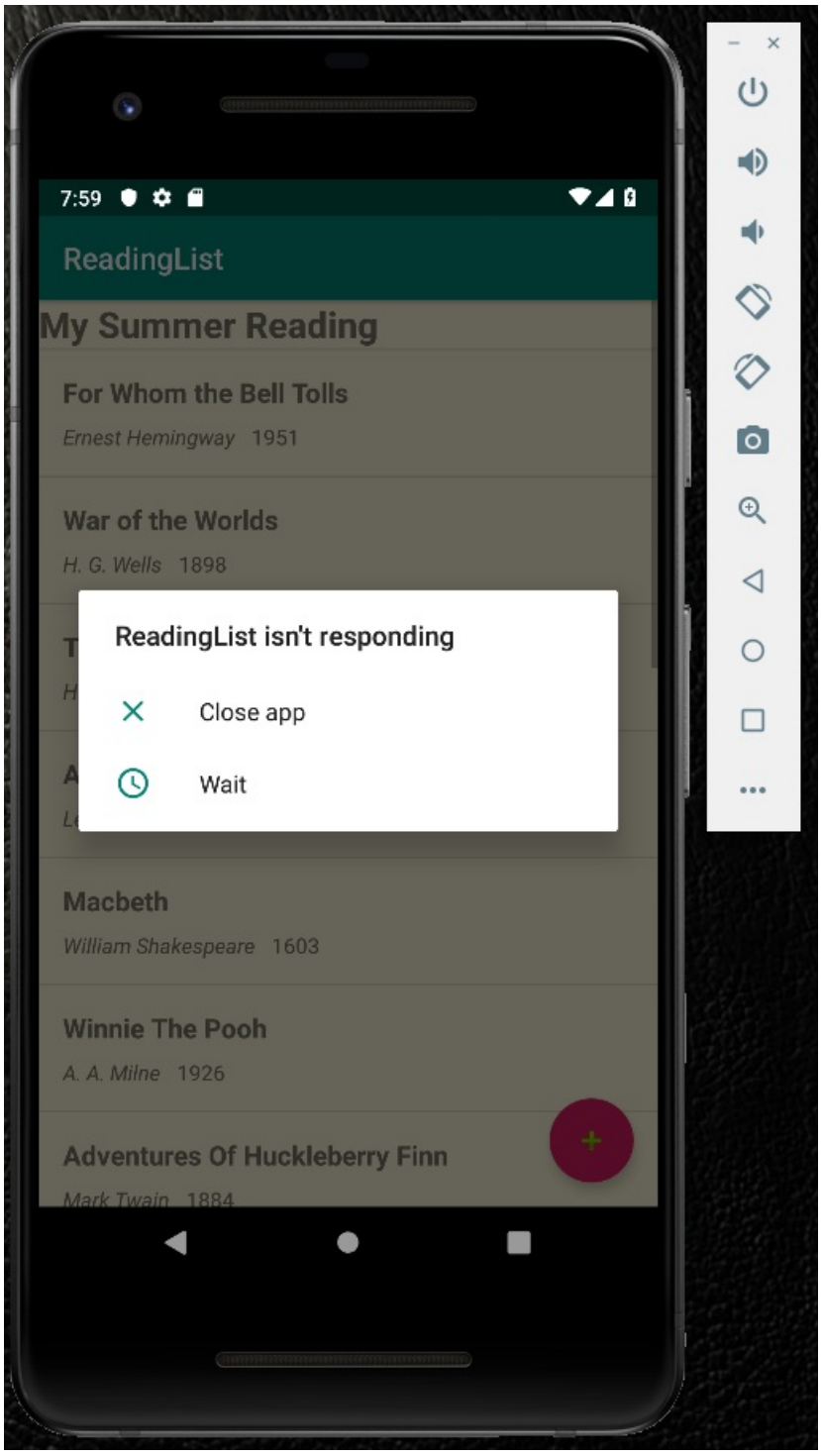
> Note that this delay is *inside* a try/catch block.

Deploy your application. You will see that you will see a mostly-blank screen for a long time, then it will eventually load.

Why didn't it crash?

Remember that the ANR is caused by failure to respond in time. We need an additional event to be requested.

Redeploy your application, but this time click the back button while the screen is mostly blank. You will notice that the button stays highlighted before eventually the application throws up the ANR message.



At this stage, it is up to the user whether to wait or close your application. Some users will wait, in which case your application will continue. Unfortunately, since applications that have this kind of problem tend to have it repeatedly, I think most people will just click close and some will even give you a negative review.

If they do close the application, Google will report the crash in your dashboard and it is up to you to figure out why it is happening.

Go ahead and remove the new line of code.

Background Thread

Since the "main" thread is our foreground UI thread, you can think of all other threads as background threads. They are useful if you want to do some long running process without causing an ANR or slowing down the UI.

So why wouldn't we just put everything in a background thread?

CalledFromWrongThreadException

Inside the end of your `onCreate`, add this snippet

```
thread {
    Log.d(ReadingListApp.TAG, "On thread: ${Thread.currentThread().name}")
    TimeUnit.SECONDS.sleep(2L)
    viewAdapter.readingList = ReadingList(
        title ="Empty Book List"
    )
}
```

The intention here is to start a new thread, log which thread it is, wait 2 seconds then update the adapter.

Open your `Logcat` tab and clear your filtering.

Deploy your application. The app should crash.

You'll have to scroll through your (unfortunately verbose) logs, but you should see

```
2018-11-26 08:36:51.461 8666-8666/com.aboutobjects.curriculum.readinglist I/ReadingListApp: 18 books
 loaded
2018-11-26 08:36:51.478 8666-8684/com.aboutobjects.curriculum.readinglist D/ReadingListApp: On threa
d: Thread-4
```

followed by

```
    --------- beginning of crash
2018-11-26 08:36:53.479 8666-8684/com.aboutobjects.curriculum.readinglist E/AndroidRuntime: FATAL E
XCEPTION: Thread-4
    Process: com.aboutobjects.curriculum.readinglist, PID: 8666
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created
a view hierarchy can touch its views.
        at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:7753)
        at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:1225)
        at android.view.View.requestLayout(View.java:23093)
        at android.view.View.requestLayout(View.java:23093)
        at android.view.View.requestLayout(View.java:23093)
        at android.view.View.requestLayout(View.java:23093)
        at android.view.View.requestLayout(View.java:23093)
        at android.view.View.requestLayout(View.java:23093)
        at androidx.constraintlayout.widget.ConstraintLayout.requestLayout(ConstraintLayout.java:324
9)
        at android.view.View.requestLayout(View.java:23093)
        at androidx.recyclerview.widget.RecyclerView.requestLayout(RecyclerView.java:4202)
        at androidx.recyclerview.widget.RecyclerView$RecyclerViewDataObserver.onChanged(RecyclerView
.java:5286)
        at androidx.recyclerview.widget.RecyclerView$AdapterDataObservable.notifyChanged(RecyclerVie
w.java:11997)
        at androidx.recyclerview.widget.RecyclerView$Adapter.notifyDataSetChanged(RecyclerView.java:
7070)
        at com.aboutobjects.curriculum.readinglist.ui.ReadingListAdapter.setReadingList(ReadingListA
dapter.kt:22)
        at com.aboutobjects.curriculum.readinglist.BookListActivity$onCreate$4.invoke(BookListActivi
ty.kt:112)
        at com.aboutobjects.curriculum.readinglist.BookListActivity$onCreate$4.invoke(BookListActivi
ty.kt:27)
        at kotlin.concurrent.ThreadsKt$thread$thread$1.run(Thread.kt:30)
```

This error tells us `CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.`

So, we can't just always use the background threads either. While we usually want to do any processing in a background thread, we often have to update the UI from the `main` thread.

Before we remove that temporary block...

Handler

One possible way of throwing items onto the UI queue is using the `Handler`. While it *can* be used for background threads, it's generally used for the UI thread.

Define a new variable

```
private val handler = Handler(Looper.getMainLooper())
```

> Make sure to use the `android.os` import not the `java.util` one.

The `Looper.getMainLooper` parameter is what tells the `Handler` to use the `main` thread.

Then, update our `thread` block

```
thread {
    Log.d(ReadingListApp.TAG, "On thread: ${Thread.currentThread().name}")
    TimeUnit.SECONDS.sleep(2L)
    handler.post {
        Log.d(ReadingListApp.TAG, "Now on thread: ${Thread.currentThread().name}")
        viewAdapter.readingList = ReadingList(
            title ="Empty Book List"
        )
    }
}
```

Here we are just posting at the end of the queue. There are other methods for more fine grained control.

Set your `Logcat` filter back to `ReadingListApp` and redeploy.

You'll see that your application loads the original list of books, then after 2 seconds, switches to the empty list of books.

In logcat, you'll see

```
2018-11-26 08:47:44.883 9442-9442/com.aboutobjects.curriculum.readinglist I/ReadingListApp: 18 books
 loaded
2018-11-26 08:47:44.905 9442-9460/com.aboutobjects.curriculum.readinglist D/ReadingListApp: On threa
d: Thread-4
2018-11-26 08:47:46.919 9442-9442/com.aboutobjects.curriculum.readinglist D/ReadingListApp: Now on t
hread: main
```

Seems simple enough. Let's look at a couple of different ways to handle this.

Remove the `thread` block as well as the `handler` variable.

> Don't forget to Optimize Imports whenever you remove blocks of code.

AsyncTask

Another possible method that used to be popular was the `AsyncTask`. While we will not walk through an example of using it, we will briefly discuss it in case you run into it while examining legacy code.

You would extend `AsyncTask` and provide a code block that would execute first on the UI thread, then another code block on a background thread, then another code block on the UI thread.

The common use case was downloading data from the internet in the background thread, then updating the UI on the UI thread.

Unfortunately, common patterns of usage led to unnecessary calls on the UI thread (remember what happens when it gets clogged up?) as well as calls being made on objects that no longer existed (because the network calls took so long to execute).

If you run into it in legacy code, I would highly recommend tagging it for refactoring (`// @TODO`). If you would like more information on `AsyncTask` you can review the JavaDocs.

Anko

Since JetBrains wrote Kotlin and they also wrote Anko, you may be tempted to give it a try. After all it looks like a trivial implementation of what `AsyncTask` was supposed to do.

Unfortunately, as of this writing, Issue #650 shows that it is not yet compatible with AndroidX and assumes the legacy Android

Support library packages.

Rx

The ReactiveX website says:

> ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming

There are multiple Rx libraries available to us.

- RxJava - primary library used to implement ReactiveX on Java-based platforms
- RxKotlin - additional Kotlin extension functions
- RxAndroid - Android specific extensions, like the `mainThread`
- RxBinding - Android UI specific extensions

build.gradle

To get started, let's update our module-level `build.gradle`.

```
implementation "io.reactivex.rxjava2:rxjava:2.2.4"
implementation 'io.reactivex.rxjava2:rxkotlin:2.3.0'
implementation 'io.reactivex.rxjava2:rxandroid:2.1.0'
```

We won't be using the RxBinding at this time.

Sync your project.

BookListActivity

One of the biggest advantages of using Rx is the ability to chain events. Let's try that out by making our load functions happen on a background thread. Of course, we'll have to switch back to the UI thread before we update the UI.

Open `BookListActivity`.

We're going to be modifying multiple methods here, so let's walk through them one at a time.

loadJsonFromAssets

Let's start with `loadJsonFromAssets`. Currently, this is called from the UI thread, which means we are reading this XML file on the UI thread. That's unfortunate. Let's fix that.

First, we we will change our return type. Instead of returning a raw `ReadingList?` we want to return a `Maybe<ReadingList>`. The `Maybe` rx class signifies that it might return 0 elements, 1 element or an error.

Once we do that, we need to update our implementation to return a Maybe version.

```
private fun loadJsonFromAssets(): Maybe<ReadingList> {
    return try{
        val reader = InputStreamReader(assets.open(JSON_FILE))
        Maybe.just(app.gson.fromJson(reader, ReadingList::class.java))
    } catch (e: Exception) {
        Log.d(ReadingListApp.TAG, "Failed to load Json from assets/$JSON_FILE",e)
        Maybe.empty<ReadingList>()
    }
}
```

> You wouldn't normally swallow errors, but we explicitly do not want an error here killing the application. If we wanted to throw an error instead, we could use `Maybe.error`.

How is this version different than the previous version? This version will allow us to control the threading. We will see that shortly.

loadJsonFromFiles

Do the same thing to `loadJsonFromFiles`.

```
private fun loadJsonFromFiles(): Maybe<ReadingList> {
    return try {
        val reader = FileReader(File(filesDir, JSON_FILE))
        Maybe.just(app.gson.fromJson(reader, ReadingList::class.java))
    } catch (e: Exception) {
        Log.d(ReadingListApp.TAG, "Failed to load Json from files/$JSON_FILE",e)
        Maybe.empty<ReadingList>()
    }
}
```

loadJson

For `loadJson`, we again want to return a `Maybe<ReadingList>` but this time we will rely on the `Maybe.switchIfEmpty` keyword to complete the logic.

```
private fun loadJson(): Maybe<ReadingList> {
    return loadJsonFromFiles()
        .switchIfEmpty(loadJsonFromAssets())
}
```

Disposable

We are going to add a variable to hold a `Disposable`.

```
private var loadJsonDisposable: Disposable? = null
```

We'll get to this in just a moment.

onCreate

Now we need to update our `onCreate` to utilize the new method. Replace the `loadJson` block inside `onCreate` with this:

```
loadJsonDisposable = loadJson()
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeBy(
        onSuccess = {
            Log.i(ReadingListApp.TAG, "${it.books.size} books loaded")
            viewAdapter.readingList = it
            saveJson(it)
        },
        onError = {
            Log.e(ReadingListApp.TAG, "Error loading books: ${it.message}", it)
            it.message?.let {
                Snackbar.make(binding.recycler, it, Snackbar.LENGTH_LONG)
                    .show()
            }
        },
        onComplete = {
            Log.w(ReadingListApp.TAG, "Unable to load any books")
        }
    )
```

> For the `Snackbar` import, you will use `com.google.android.material.snackbar.Snackbar`.

Admittedly, this code snippet seems longer. It is, however, doing a lot more.

Here, we are using `RxAndroid` to specify that we want to load the json on a new background thread, but call the onSuccess method on the main UI thread. If we successfully load *either* of the json files, we log it, update the adapter and save it. If there is an error, we are currently logging it and showing a snackbar at the bottom of the page. If we fail to load any books, but there is no error (we are currently swallowing them after all) then it will just log a warning.

Note that we assign the return value to our `loadJsonDisposable`?

onDestroy

Override `onDestroy`

If that variable is set, we want to dispose of it. This allows us to tell the system we no longer care about the results of that file being loaded if we are shutting down.

```
override fun onDestroy() {
    loadJsonDisposable?.dispose()
    super.onDestroy()
}
```

saveJson

You'll notice that we are still saving the json on the UI thread. Let's change that.

Looking at our `saveJson` method, we take a parameter, but don't return anything. What we will want to use for this is `Completable` which allows us to specify success/error.

```
private fun saveJson(readingList: ReadingList): Completable {
    return try{
        val writer = FileWriter(File(filesDir, JSON_FILE))
        app.gson.toJson(readingList, writer)
        writer.flush()
        writer.close()
        Completable.complete()
    }catch(e: Exception){
        Log.d(ReadingListApp.TAG, "Failed to save Json to files/$JSON_FILE",e)
        Completable.error(e)
    }
}
```

We just specify that we are `complete` or that there was an `error` (as well as update the return type).

We call this from two places.

onCreate

```
saveJson(it)
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe()
```

We're already logging the failure, so we won't do any extra handling here. Worst case scenario, next time we start the app it starts with the same content it did this time.

onActivityResult

In `onActivityResult` it's a different story. If we fail to save the changes, then that book won't exist the next time we launch the app. In this case, we will show a snackbar if there is an error, and we will only add the book to the UI if it succeeded.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    // We ignore any other request we see
    if (requestCode == EDIT_REQUEST_CODE || requestCode == NEW_REQUEST_CODE) {
        // We ignore any canceled result
        if (resultCode == Activity.RESULT_OK) {
            // Grab the books from the data
            val source = getBook(data, EditBookActivity.EXTRA_SOURCE_BOOK)
            val edited = getBook(data, EditBookActivity.EXTRA_EDITED_BOOK)
            if (edited != null) {
                // Grab the old reading list since it is read-only
                viewAdapter.readingList?.let { oldReadingList ->
                    // Create a new ReadingList
                    val newReadingList = ReadingList(
                        title = oldReadingList.title,
                        books = oldReadingList.books
                            .toMutableList()
                            .filterNot {
                                it.title == source?.title
                                && it.author == source?.author
                                && it.year == source?.year
                            }.plus(edited)
                            .toList()
                    )
                    // Save the results
                    saveJson(newReadingList)
                        .subscribeOn(Schedulers.newThread())
                        .observeOn(AndroidSchedulers.mainThread())
                        .subscribeBy(
                            onComplete = {
                                // And update our view
                                viewAdapter.readingList = newReadingList
                            },
                            onError = {
                                it.message?.let {
                                    Snackbar.make(binding.recycler, it, Snackbar.LENGTH_LONG)
                                        .show()
                                }
                            }
                        )
                }
            }
        }
    }
    super.onActivityResult(requestCode, resultCode, data)
}
```

Now, this introduces an interesting side effect. If it takes a long time to save the file, it would then take a long time for their newly edited book to appear in the list. A better approach would probably be to add it with some indicator that it is pending; then update it once it is complete. We'll leave that as an exercise for you to do at your leisure.

We've changed our method signatures pretty drastically. Maybe we should run our `test` and `androidTest`?

#

- Current Repo: v1-Threading
- Continue to Drawing -->

Unit-Testing.md

Goal

Learn about test-driven development.

Table of Contents

- State of Android Testing

- Unit Tests
- Integration and Instrumentation Tests
- UI Tests
- End-2-End Tests

State of Android Testing

Compared to other platforms, Android has had a bit of a reliability problem when it comes to testing.

I'll give you an example. I was working for this company and we had a ton of tests. The problem was that our software interacted with other applications in the system, and we wanted to also test that functionality. We turned to Calabash and Cucumber and re-wrote all of our tests. Cucumber was great, as the Product Owners were able to understand what the test was meant to do and make corrections. Calabash gave us the power to extend the functionality to do what we needed. Unfortunately, it relied on *how* Android worked under the covers. During a major OS revision (which, you remember happens about once a year) it broke Calabash and all of our tests. Our ability to run *any* test was completely broken overnight.

Similarly, at another company, the prior developers jumped on the bandwagon to convert all tests over to Espresso. When I tried to understand why functionality of the application was broken, I realized that the tests were doing things like "Set the text field to XYZ and verify it is XYZ", which didn't validate any of the applications actual business logic.

Google took a look at some of these common problems and came up with a list of recommendations. The most quoted one has to do with the Testing Pyramid.

To paraphrase, they recommend that *about 70% of your tests are straight Unit Tests, 20% are Integration Tests and 10% UI Tests*.

Sometimes there is confusion between Integration and Instrumentation. Sometimes there is confusion between UI Tests and End-2-End Tests. We'll try to talk about each level of testing and give examples.

Google's document also equates Unit Tests to Small Tests, Integration to Medium Tests and UI Tests to Large Tests.

Unit Tests

Unit Tests should validate small specific pieces of functionality or business logic.

For our purposes, we will also assume that Unit Tests are running from the build - but *NOT* on a device or emulator.

ExampleUnitTest

You might remember that our codebase already has one that came with the autogenerated Hello World?

Open the `app/java/com (test)/aboutobjects/curriculum/readinglist/ExampleUnitTest`. You'll notice that the breadcrumb points to `app/src/test/java` not `app/src/main/java`.

```
package com.aboutobjects.curriculum.readinglist

import org.junit.Test

import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

In the default unit test, it is validating that addition works correctly. Not all that useful, but it helps us understand how this works.

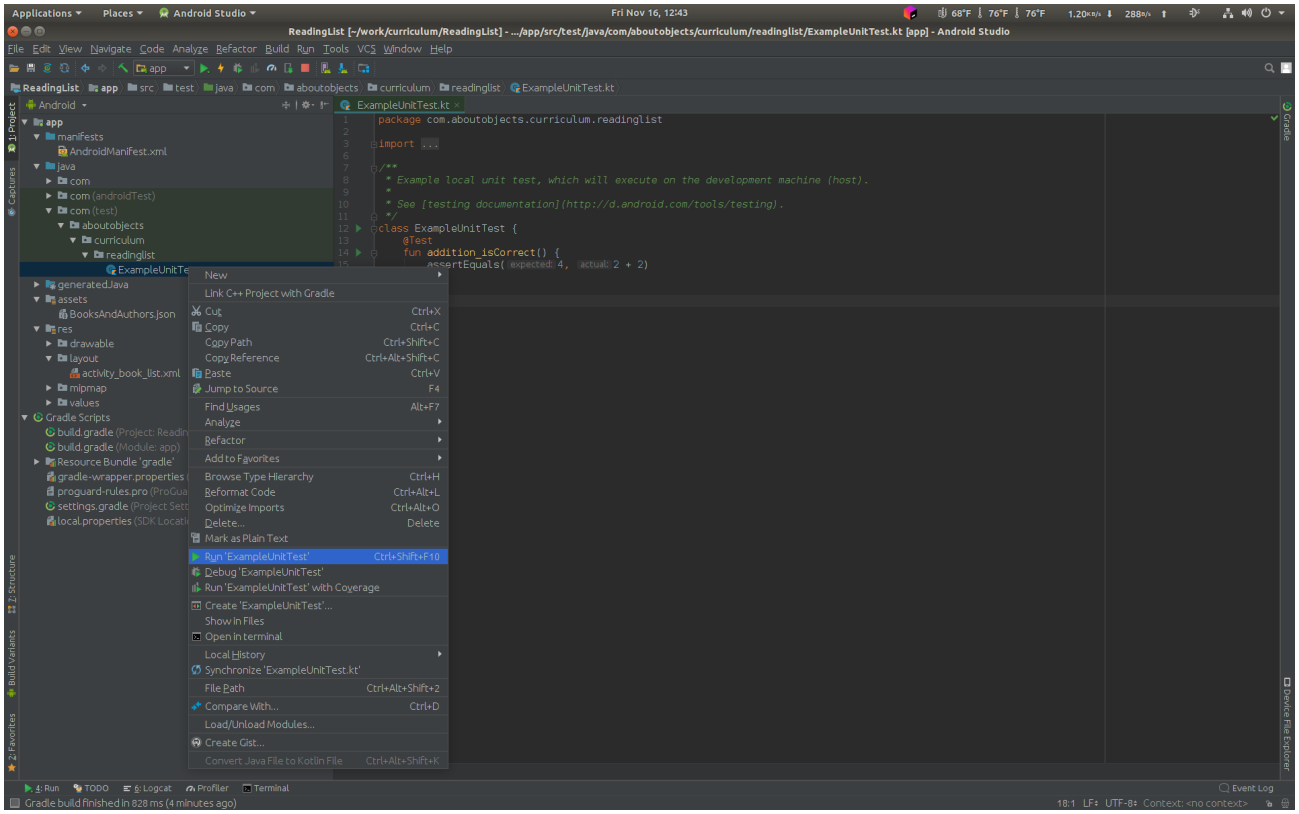
You'll notice that the function is annotated with `@Test` to tell the system that this method is a test to be run.

Also, in the `assertEquals(4, 2 + 2)` line, we are "asserting that 4 is the result of 2 + 2". IE: the `4` is the expected result that we are trying to match and the `2 + 2` is the item we are trying to test.

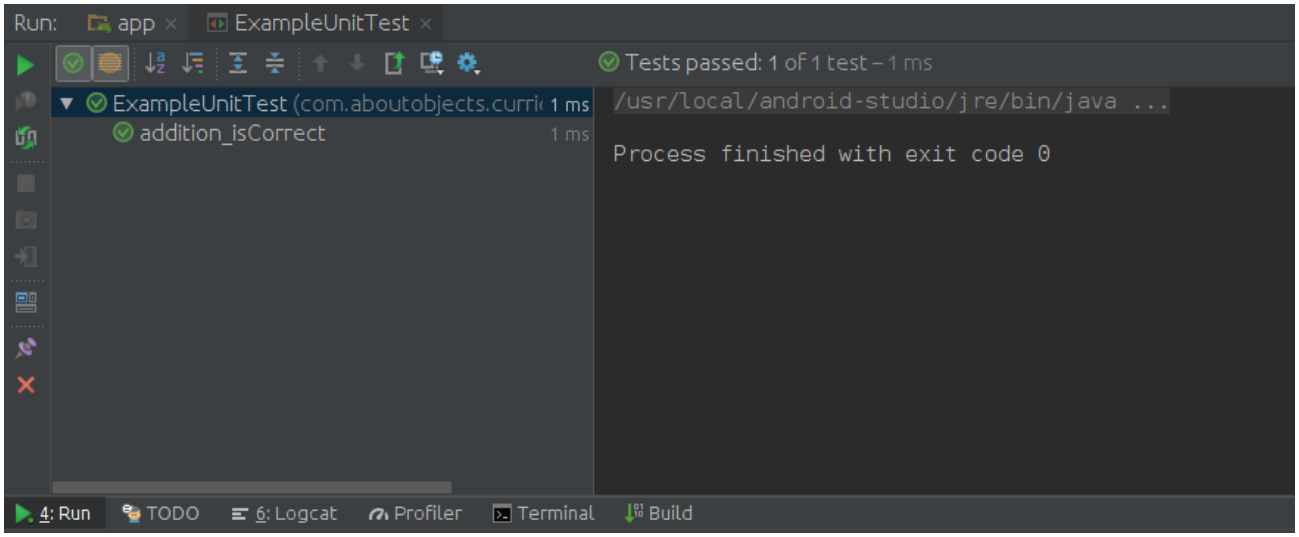
If you have ever used JUnit before, that's all this is.

Running ExampleUnitTest

To run the individual test from the IDE, right-click on it and choose `Run ExampleUnitTest`.

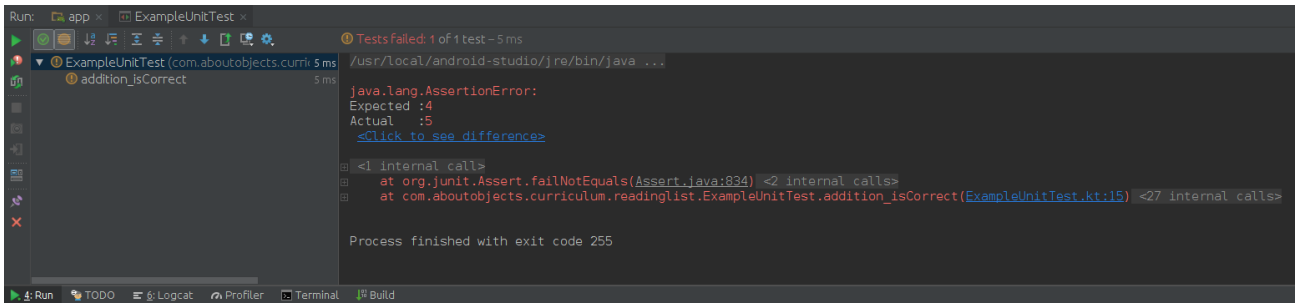


At the bottom of the screen, a window will pop up showing you the results of each `@Test`-annotated method in your test class.



> If you right-click on the `readinglist` package, you can choose to run all tests in that package at once.

What happens if you change the test so that it fails? Maybe change it to `2 + 3` but leave the expected result as `4`?



There are a few interesting things now.

- The icon next to ExampleUnitTest tells you that at least one test in that class failed
- The icon next to addition_isCorrect tells you that particular test failed
- The right side of the screen tells you that 1 test failed out of 1 total
- It tells you that running all the tests took 5ms (might be different for you)
- It tells you that the Expected result was 4, but the Actual result was 5
- And it tells you exactly where the error occurred. In fact, the `ExampleUnitTest.kt:15` is a clickable link.

Go ahead and change the results back to `2 + 2` and verify that the error goes away when you re-run the test.

> Did you also notice that the `app` pulldown in the toolbar next to the play button now shows your test instead of the app? If you wanted to go back to deploying your application, you would need to switch that pulldown from `ExampleUnitTest` back to `app`.

More useful test?

Let's be honest. We all knew that basic addition was going to work. Why don't we add something more useful? How about something to do with our models we created?

Right-click on the `readinglist` package containing `ExampleUnitTest` and choose `New | Package`. Choose `model` to match our main source tree. This has the added advantage of giving us access to package-private members, if the need arises.

Inside the new `model` package, choose `New | Kotlin File/Class`. Name it `ReadingListTest` and give it a type of `Class`.

```
package com.aboutobjects.curriculum.readinglist.model

class ReadingListTest {
}
```

Remember how we specified that our ReadingList could have an empty book list? Let's test that.

```
@Test
fun no_books() {
    val readingList = ReadingList(
        title = "no_books_title"
    )
    Assert.assertEquals(0, readingList.books.size)
}
```

> Note: Make sure to use the `org.junit` import, not the `junit.framework` import.

Here, we are creating a new `ReadingList` model without specifying a `books` attribute. We then verify that the model has 0 books (rather than `null`) because of the `emptyList()` we put in the model constructor.

Run the tests.

Let's try another one.

```
@Test
fun two_books() {
    val readingList = ReadingList(
        title = "two books title",
        books = listOf(
            Book(),
            Book(title = "second book title")
        )
    )
    Assert.assertEquals(2, readingList.books.size)
    Assert.assertEquals(null, readingList.books[0].title)
    Assert.assertEquals("second book title", readingList.books[1].title)
}
```

Here, we are adding two books but the first one doesn't even have a title. Note that the books array is 0-based, not 1-based.

Run the tests.

What about testing our model deserialization?

In this case, we are not testing the ReadingList model, but the Author model. Let's make a new AuthorTest class for this.

```
package com.aboutobjects.curriculum.readinglist.model

import com.google.gson.Gson
import org.junit.Assert
import org.junit.Test

class AuthorTest {
    @Test
    fun deserialize() {
        val json = "{ \"firstName\" : \"Ernest\", \"lastName\" : \"Hemingway\" }"
        val author = Gson().fromJson<Author>(json, Author::class.java)
        Assert.assertEquals("Ernest", author.firstName)
        Assert.assertEquals("Hemingway", author.lastName)
    }
}
```

Here, we are loading an `Author` model from the Json string.

Run the tests.

Remember - you can run individual tests, or entire packages (recursively).

You can find out more about JUnit tests here.

Assertions

Above, we are using stock JUnit assertions. There are other libraries that could be more intuitive for you.

Hamcrest

In your module-level `build.gradle`

```
testImplementation 'org.hamcrest:hamcrest-library:1.3'
```

If we use Hamcrest, then in our `ReadingListTest` we would replace

```
assertEquals("second book title", readingList.books[1].title)
```

with

```
assertThat(readingList.books[1].title, `is`(equalTo("second book title")))
```

It's slightly longer, but perhaps more intuitive.

Truth

In your module-level `build.gradle`

```
testImplementation 'androidx.test.ext:junit:1.0.0'
testImplementation 'androidx.test.ext:truth:1.0.0'
testImplementation 'com.google.truth:truth:0.42'
```

If we use Truth, then in our `ReadingListTest` we would replace

```
assertEquals("second book title", readingList.books[1].title)
```

with

```
assertThat(readingList.books[1].title).isEqualTo("second book title")
```

> Note: Since both Hamcrest and Truth use the same method names, they conflict with each other. If you want to use them in the same class, you will need to not use the static import on at least one of them. For example, you can use `MatcherAssert.assertThat` from Hamcrest and `Truth.assertThat` from Truth.

Integration and Instrumentation Tests

First, let's talk conceptually about the difference between integration tests and instrumentation tests. This can be a confusing area because many devs talk about them interchangeably.

In general, this category of tests run unit-like tests, but on a device. That device might be a physical device (your phone), emulator (like you have been using). It's also possible that we use a fake Android system (Robolectric) or that we mock out calls (Mockito).

In reality, these are often mixed.

For our purposes, this category will be tests that

- require the Android system to function (otherwise, just use a Unit Test)
- do not require the UI (otherwise they belong in the next section).

ExampleInstrumentedTest

Let's take a look at what was auto-generated.

Open the `app/java/com (androidTest)/aboutobjects/curriculum/readinglist/ExampleInstrumentedTest`. You'll notice that the breadcrumb points to `app/src/androidTest/java` not `app/src/main/java`.

You'll also notice upon opening the file that some of the auto-generated code is already marked as deprecated (strikethrough). Well, that's not ideal.

Before we go changing anything, run that test. This time, unlike during the Unit Tests, it will ask you which device to run the test on. We'll run it on our emulator.

Fixing the Deprecation

Now let's see what we can do about that deprecation.

If you hover your mouse over `AndroidJunit4` in `@RunWith(AndroidJUnit4::class)` you will see that it says `'AndroidJunit4' is deprecated. Deprecated in Java.`.

If we look at the JavaDocs we see that it says `use androidx.test.ext.junit.runners.AndroidJUnit4 instead.`

Remember our earlier discussion about the State of Android Testing and the other discussion about Expect Change? If you go to the current documentation for the Junit Runner, it still tells you to use `androidTestImplementation 'androidx.test:runner:1.1.0'`. However, if you go further down the page and click on the Release Notes you will see that it says `Deprecate androidx.test.runner.AndroidJUnit4 and replace with`

`androidx.test.ext.junit.runners.AndroidJUnit4`.

Lesson here - documentation is not always up to date.

So how do we fix it? This is the type of case where StackOverflow can be your friend. If you have not started using it yet, be sure to bookmark the site.

According to that S.O. answer, we will replace `androidTestImplementation 'androidx.test:runner:1.1.0'` with `androidTestImplementation "androidx.test.ext:junit:1.0.0"`.

Open your module-level `build.gradle`. Find the line that starts with `androidTestImplementation 'androidx.test:runner`. Currently, mine says `androidTestImplementation 'androidx.test:runner:1.1.1-alpha01'`, so already the S.O. answer is already out of date (Expect Change). Replace that entire line with:

```
androidTestImplementation "androidx.test.ext:junit:1.0.0"
```

Sync your project.

Go back to `ExampleInstrumentedTest`.

Remove the `AndroidJunit4` import (you may need to expand the imports to see it), and when prompted select the `ext` one.

Now, what about the `InstrumentationRegistry`? Let's take a shortcut this time. Ctrl+Click (or Command+Click on a Mac) on the struck-through word `InstrumentationRegistry`. You will notice in the Javadocs provided there that it says:

```
* @deprecated use {@link androidx.test.core.app.ApplicationProvider} or {@link
*     androidx.test.platform.app.InstrumentationRegistry} instead
```

Ok, so that is helpful - but which one to use?

This time, go back to `ExampleInstrumentedTest` and Ctrl+Click on the `getTargetContext` function itself.

Now it is more explicit.

```
* @deprecated use {@link androidx.test.core.app.ApplicationProvider#getApplicationContext()}
*     instead.
```

Go back to your module-level `build.gradle` and add two more dependencies (which I found here)

```
androidTestImplementation 'androidx.test:core:1.0.0'
androidTestImplementation 'org.mockito:mockito-core:2.18.3'
```

Sync your project.

In `ExampleInstrumentedTest` replace `val appContext = InstrumentationRegistry.getTargetContext()` with the information from above:

```
val appContext = androidx.test.core.app.ApplicationProvider.getApplicationContext<Application>()
```

Note that we have to specify the default Application as the type here. If we were to have our own custom Application class, we could use it instead -- but we have to specify *something* that is or extends Application.

Place your cursor on `getApplicationContext` and use Alt+Enter (Option+Enter on a Mac) and select "Add import for androidx.test.core.app.ApplicationProvider.getApplicationContext"

End result will look something like this:

```
import androidx.test.core.app.ApplicationProvider.getApplicationContext
...
val appContext = getApplicationContext<Application>()
```

If you have not already done so, optimize your imports with Ctrl+Alt+O (or Ctrl+Option+O on a Mac).

Run that test.

Custom Application Class

Why would you need a custom application class?

It's often used to provide initialization that you only want to happen once during the lifetime of your application. If you have multiple activities, maybe you want something to be run once rather than every time the main screen is reloaded.

Let's put together a custom Application so that we can see how it changes the test above.

Open your `app/manifests/AndroidManifest.xml` (which actually points to `app/src/main/AndroidManifest.xml`). We are going to add an attribute to the `application` tag.

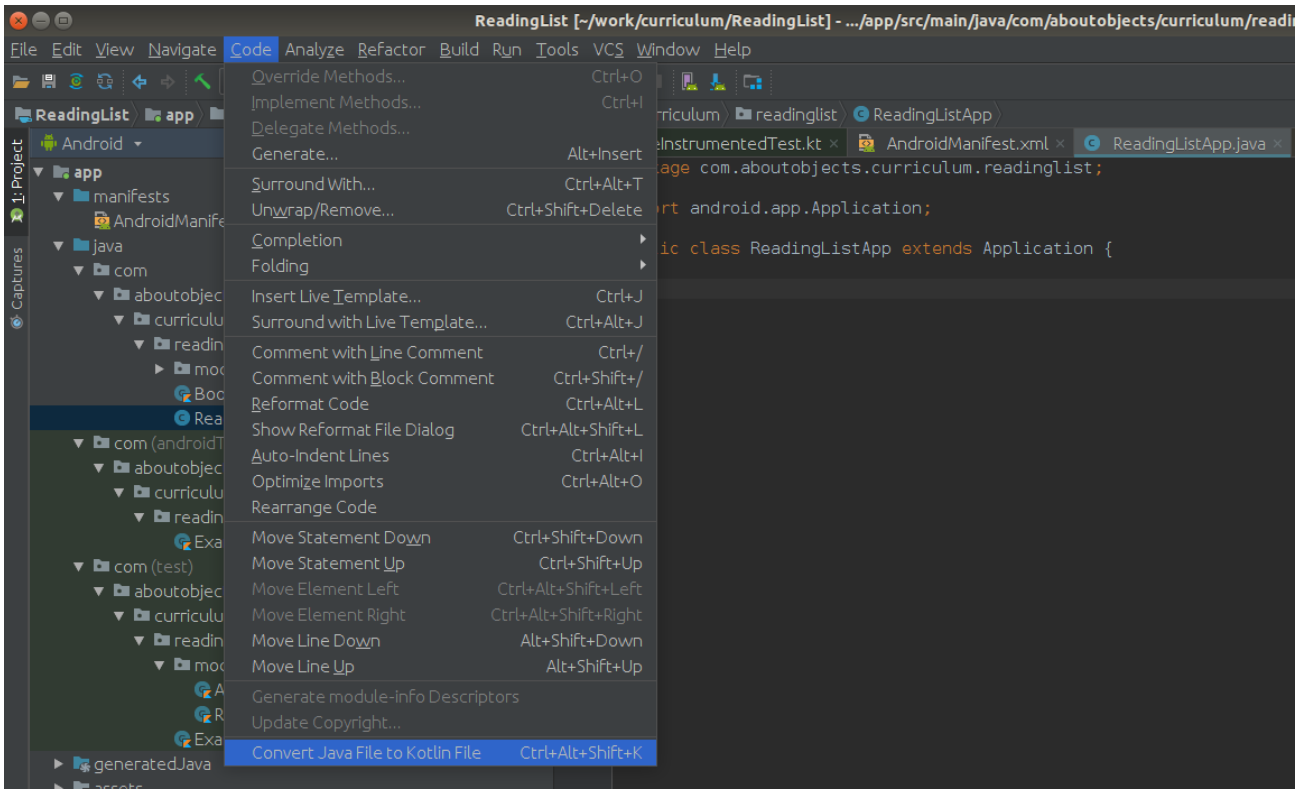
```
<application
        android:name=".ReadingListApp"
```

Remember to include the period at the beginning unless you want to specify the fully-qualified package name.

The `ReadingListApp` will be in red. Click on `ReadingListApp` and then Alt+Enter (or Option+Enter on Mac) to do a Quick Fix. Select `Create class ReadingListApp`. It will autogenerate a starting Application class for you... but in Java. Open it.

```
package com.aboutobjects.curriculum.readinglist;

import android.app.Application;

public class ReadingListApp extends Application {
}
```

In your menu bar, choose `Code | Convert Java File to Kotlin File`



You now have a starting Application class, in Kotlin

```
package com.aboutobjects.curriculum.readinglist

import android.app.Application

class ReadingListApp : Application()
```

Let's give it something to do. How about we move that Gson initialization from `BookListActivity` to `ReadingListApp`?

Update `ReadingListApp`

```
package com.aboutobjects.curriculum.readinglist

import android.app.Application
import com.google.gson.Gson
import com.google.gson.GsonBuilder

class ReadingListApp : Application() {
    val gson: Gson by lazy {
        GsonBuilder()
            .setPrettyPrinting()
            .create()
    }
}
```

Tell `BookListActivity` about the new Application class

```
private val app: ReadingListApp by lazy { application as ReadingListApp }
```

We do `by lazy` because otherwise, `application` can be null when accessed too early.

Remove your existing `private val gson` variable from `BookListActivity`.

Then change both references to `gson` to `app.gson`. They are both highlighted in red.

Why would we do this?

Imagine that we have dozens of activities, services and adapters. We can change the implementation of Gson in one place, and have it affect all of them.

Although more in-depth than we will cover here, that is the basic idea behind dependency injection frameworks, like Dagger.

Let's continue.

Open `ExampleInstrumentedTest`, change `Application` to `ReadingListApp` and optimize your imports.

Now, you could do `appContext.gson`, for example.

Run your test and make sure everything still works.

More useful instrumentation test?

So what would be a useful instrumentation test? Something that relies on knowledge of the Android system - but does not rely on the UI?

Remember when we wrote `"${it.books.size} books loaded"`? If we had been doing proper TDD (Test-Driven Development), we would have noticed earlier that the message looks odd when there is only 1 book loaded. Maybe we need to not only move that string to the resources, but choose which string based on how many books were loaded?

In this case, we are going to retrofit our application to be testable, and test it.

Let's start by defining our resources.

Open your `strings.xml` and create 3 new strings.

```
<string name="loaded_none">No books found.</string>
<string name="loaded_one">One book loaded.</string>
<string name="loaded_some">%1$d books loaded.</string>
```

In the `BookListActivity` FILE, but *AFTER* the last closing BRACKET (`}`) add a function

```
fun Context.getBooksLoadedMessage(numberOfBooks: Int): String {
    return when(numberOfBooks) {
        0 -> getString(R.string.loaded_none)
        1 -> getString(R.string.loaded_one)
        else -> getString(R.string.loaded_some, numberOfBooks)
    }
}
```

This probably looks very odd to Java developers. This is what Kotlin calls an Extension function. Technically the Java equivalent:

```
public static String getBooksLoadedMessage(Context context, int numberOfBooks)
```

The IDE just does a really nice job of making it look cleaner by making it look like the `Context` class had the function all along.

Inside our onCreate, replace

```
findViewById<TextView>(R.id.hello_text).text = "${it.books.size} books loaded"
```

with

```
findViewById<TextView>(R.id.hello_text).text = getBooksLoadedMessage(it.books.size)
```

Run the application (not the test) and make sure the application still functions.

Now, to test it. Although we are using that function to display something on the UI - we are going to test the function without the UI.

In the same package as `ExampleInstrumentedTest` add a `BookListActivityTest`.

Annotate the class with `@RunWith(AndroidJUnit4::class)` the same way the example was done. Remember to use the `ext` version.

Save yourself some headache and add an `import org.junit.Assert.*` in the import section at the top of the page. You shouldn't have to do this, but for some reason the IDE sometimes can't find it. It will be temporarily grayed out until we start using it. If you optimize imports, it will go away.

Let's add a couple tests now. We'll copy the `appContext` from our `ExampleInstrumentedTest` and add an `assertTrue` for each one.

```
@Test
fun loadedText_noBooks() {
    val appContext = ApplicationProvider.getApplicationContext<ReadingListApp>()
    assertTrue(appContext.getBooksLoadedMessage(0).contains("No books found"))
}

@Test
fun loadedText_oneBook() {
    val appContext = ApplicationProvider.getApplicationContext<ReadingListApp>()
    assertTrue(appContext.getBooksLoadedMessage(1).contains("One book loaded"))
}

@Test
fun loadedText_twoBooks() {
    val appContext = ApplicationProvider.getApplicationContext<ReadingListApp>()
    assertTrue(appContext.getBooksLoadedMessage(2).contains("2 books loaded"))
}
```

Run the tests.

You are probably noticing a lot of redundancy? Let's see if we can make it a little cleaner.

Pull the `appContext` out into it's own variable, and let's rename it `context`.

```
package com.aboutobjects.curriculum.readinglist

import androidx.test.core.app.ApplicationProvider
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.Assert.*

@RunWith(AndroidJUnit4::class)
class BookListActivityTest {
    val context = ApplicationProvider.getApplicationContext<ReadingListApp>()

    @Test
    fun loadedText_noBooks() {
        assertTrue(context.getBooksLoadedMessage(0).contains("No books"))
    }

    @Test
    fun loadedText_oneBook() {
        assertTrue(context.getBooksLoadedMessage(1).contains("One book"))
    }

    @Test
    fun loadedText_twoBooks() {
        assertTrue(context.getBooksLoadedMessage(2).contains("2 books"))
    }
}
```

Run the tests.

> A word of caution. Validating that the text matches today is not always a good plan. As product owners ask for text to be changed, it is really easy to quickly change the `strings.xml`, which is how it is designed. For that to then break the tests is a bit unmanageable - especially because you are not testing any business logic. In this particular case, we are trying to validate that we chose the correct one. You could further change it to say that the you are validating that they are matching the `context.getString` values; but then you are just reimplementing the code you are trying to test rather than testing that it did the correct thing. If one is wrong they would both be wrong.

As you might be able to tell, these tests are slower than the straight unit tests. That is in large part why it is recommended to have more unit tests than integration/instrumentation tests.

UI Tests

UI Tests take it a step further. They will be even slower as they require that the UI is shown and parts of the UI are evaluated. As such, there should be even less of them.

They would be useful for verifying that the UI behaves like we expect.

Let's create an example.

First, as suggested on the Espresso page, add these to your module-level `build.gradle`

```
androidTestImplementation 'androidx.test:runner:1.1.0'
androidTestImplementation 'androidx.test:rules:1.1.0'
```

We didn't add the third one because we already have it.

Sync your application.

In the same package as `BookListActivityTest`, let's create `BookListActivityUITests`.

Like before, we will annotate the class with `@RunWith(AndroidJUnit4::class)`.

Similar to when we did `import org.junit.Assert.*`, we shouldn't have to manually add the `import androidx.test.espresso.Espresso.*`, but...

```
package com.aboutobjects.curriculum.readinglist

import androidx.test.core.app.ActivityScenario
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.Test
import org.junit.runner.RunWith
import androidx.test.espresso.Espresso.*
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.isDisplayed
import androidx.test.espresso.matcher.ViewMatchers.withId

@RunWith(AndroidJUnit4::class)
class BookListActivityUITests {
    @Test
    fun lastLogin_isDisplayed() {
        val scenario = ActivityScenario.launch(BookListActivity::class.java)

        // Make sure the login_text is displayed
        onView(withId(R.id.login_text))
            .check(matches(isDisplayed()))
    }
}
```

The overall test function looks the same; but the contents of the test function are quite different.

Here, we are specifying that we want to launch our `BookListActivity`, grab the view that uses `android:id="@+id/login_text"` and verify that it is displayed.

Run `BookListActivityUITests`. Did it succeed?

If so, clear your application data and try again.

Debugging

So what's the problem?

Let's take a look at the error message.

```
androidx.test.espresso.base.DefaultFailureHandler$AssertionFailedWithCauseError: 'is displayed on th
e screen to the user' doesn't match the selected view.
Expected: is displayed on the screen to the user
Got: "AppCompatTextView{id=2131165283, res-name=login_text, visibility=VISIBLE, width=0, height=43,
has-focus=false, has-focusable=false, has-window-focus=true, is-clickable=false, is-enabled=true, is
-focused=false, is-focusable=false, is-layout-requested=false, is-selected=false, layout-params=andr
oidx.constraintlayout.widget.ConstraintLayout$LayoutParams@8b58bdc, tag=null, root-is-layout-request
ed=false, has-input-connection=false, x=540.0, y=183.0, text=, input-type=0, ime-target=false, has-l
inks=false}"

at dalvik.system.VMStack.getThreadStackTrace(Native Method)
at java.lang.Thread.getStackTrace(Thread.java:1538)
at androidx.test.espresso.base.DefaultFailureHandler.getUserFriendlyError(DefaultFailureHandler.java
:88)
at androidx.test.espresso.base.DefaultFailureHandler.handle(DefaultFailureHandler.java:51)
at androidx.test.espresso.ViewInteraction.waitForAndHandleInteractionResults(ViewInteraction.java:31
4)
at androidx.test.espresso.ViewInteraction.check(ViewInteraction.java:297)
at com.aboutobjects.curriculum.readinglist.BookListActivityUITests.lastLogin_isDisplayed(BookListAct
ivityUITests.kt:20)
at java.lang.reflect.Method.invoke(Native Method)
at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)
at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)
at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
```

```
at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
at androidx.test.ext.junit.runners.AndroidJUnit4.run(AndroidJUnit4.java:104)
at org.junit.runners.Suite.runChild(Suite.java:128)
at org.junit.runners.Suite.runChild(Suite.java:27)
at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
at org.junit.runner.JUnitCore.run(JUnitCore.java:115)
at androidx.test.internal.runner.TestExecutor.execute(TestExecutor.java:56)
at androidx.test.runner.AndroidJUnitRunner.onStart(AndroidJUnitRunner.java:388)
at android.app.Instrumentation$InstrumentationThread.run(Instrumentation.java:2145)
Caused by: junit.framework.AssertionFailedError: 'is displayed on the screen to the user' doesn't ma
tch the selected view.
Expected: is displayed on the screen to the user
Got: "AppCompatTextView{id=2131165283, res-name=login_text, visibility=VISIBLE, width=0, height=43,
has-focus=false, has-focusable=false, has-window-focus=true, is-clickable=false, is-enabled=true, is
-focused=false, is-focusable=false, is-layout-requested=false, is-selected=false, layout-params=andr
oidx.constraintlayout.widget.ConstraintLayout$LayoutParams@8b58bdc, tag=null, root-is-layout-request
ed=false, has-input-connection=false, x=540.0, y=183.0, text=, input-type=0, ime-target=false, has-l
inks=false}"

at androidx.test.espresso.matcher.ViewMatchers.assertThat(ViewMatchers.java:539)
at androidx.test.espresso.assertion.ViewAssertions$MatchesViewAssertion.check(ViewAssertions.java:10
3)
at androidx.test.espresso.ViewInteraction$SingleExecutionViewAssertion.check(ViewInteraction.java:41
5)
at androidx.test.espresso.ViewInteraction$2.call(ViewInteraction.java:279)
at androidx.test.espresso.ViewInteraction$2.call(ViewInteraction.java:265)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at android.os.Handler.handleCallback(Handler.java:873)
at android.os.Handler.dispatchMessage(Handler.java:99)
at android.os.Looper.loop(Looper.java:193)
at android.app.ActivityThread.main(ActivityThread.java:6669)
at java.lang.reflect.Method.invoke(Native Method)
at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)
```

That's a pretty long and somewhat unfriendly error. Let's focus on this part

```
Got: "AppCompatTextView{id=2131165283, res-name=login_text, visibility=VISIBLE, width=0, height=43,
has-focus=false, has-focusable=false, has-window-focus=true, is-clickable=false, is-enabled=true, is
-focused=false, is-focusable=false, is-layout-requested=false, is-selected=false, layout-params=andr
oidx.constraintlayout.widget.ConstraintLayout$LayoutParams@8b58bdc, tag=null, root-is-layout-request
ed=false, has-input-connection=false, x=540.0, y=183.0, text=, input-type=0, ime-target=false, has-l
inks=false}"
```

That shows us that it did in fact find the `AppCompatTextView` with the res-name of `login_text`... but if you look near the end you will see `text=`.

Remember how our application only shows the last login information on the second time you launch it? That's what we are running into here. When we clear the data, that field is empty, thus it is considered as not-displayed.

How do we fix it?

You might be tempted to change what the application does -- but remember, it behaves as we wanted it to. We need to change the test expectations.

Since what we want to validate is that it is NOT displayed the first time; but it IS displayed the second time - we have two things to validate.

Well, we could test it, restart it, then test it again - right?

```
@Test
fun lastLogin_isDisplayed() {
    val scenario = ActivityScenario.launch(BookListActivity::class.java)

    // Make sure the login_text is NOT displayed
    onView(withId(R.id.login_text))
        .check(matches(not(isDisplayed())))

    // Relaunch our activity a second time
    scenario.recreate()

    // Make sure the login_text is displayed
    onView(withId(R.id.login_text))
        .check(matches(isDisplayed()))
}
```

Run it. What happens?

```
text=Your last login was: 2018.11.19 at 14:05:01 PST
```

On the first pass, the text was set. Why?

It's because our SharedPreferences still have the value from the last run.

In our `BookListActivity`, add a new entry to our `companion object`:

```
const val PREF_FILE = "samplePrefs"
```

and use the new value in the function

```
private val prefs: SharedPreferences by lazy {
    getSharedPreferences(PREF_FILE, Context.MODE_PRIVATE)
}
```

Now, back in the `BookListActivityUITests` add a new `@Before` function to run before each test.

```
@Before
fun clearSharedPreferences() {
    val context = ApplicationProvider.getApplicationContext<ReadingListApp>()
    context.getSharedPreferences(BookListActivity.PREF_FILE, Context.MODE_PRIVATE)
        .edit { clear() }
}
```

Run the test a few times, and you will see that it now functions correctly. Each time it runs *from a test*, it clears the shared preferences, validates that the field is empty, restarts the activity and then verifies that it is no longer empty.

> As you can probably tell, you are no longer in the realm of just validating one tiny piece of business logic. There are lots of moving parts and if any one of them goes wrong, it looks like everything fell apart. Besides performance issues, this is a big reason why you should have less of these tests than of the others. These tests are less reliable than the others, more likely to break, and going to cost you more time to maintain.

There are, of course, other things you can do with Espresso. If we had buttons you could do something like

```
onView(withId(R.id.our_button))
    .perform(click())
```

One thing to keep in mind when you start doing these kinds of tests is the amount of time it takes to perform those tests will affect whether the next steps succeed or fail. Often, developers will turn off animation on a device so that transitions happen more quickly. It's also common to extend the functionality to wait for certain views to be on the screen before they move to the next step - which can lead to a deadlock.

You can find more on Espresso testing here.

Some additional Espresso dependencies you might find useful:

```
androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0'
androidTestImplementation('androidx.test.espresso:espresso-contrib:3.1.0') {
    exclude module:'recyclerview'
}
androidTestImplementation 'androidx.test.espresso:espresso-intents:3.1.0'
androidTestImplementation 'androidx.test.espresso:espresso-accessibility:3.1.0'
androidTestImplementation 'androidx.test.espresso:espresso-web:3.1.0'
androidTestImplementation 'androidx.test.espresso.idling:idling-concurrent:3.1.0'

// The following Espresso dependency can be either "implementation"
// or "androidTestImplementation", depending on whether you want the
// dependency to appear on your APK's compile classpath or the test APK
// classpath.
androidTestImplementation 'androidx.test.espresso:espresso-idling-resource:3.1.0'
```

End-2-End Tests

No discussion on testing would be complete without at least discussing the topic of End-2-End testing. This would be where the tests start the application, run through all the functionality and validate that no new regressions have occurred.

While this is a good goal, the reality is that these kinds of tests rarely stay up-to-date and when they do it is at a large financial and temporal cost. As we saw with the UI tests, even just validating that a field is on the screen (or not) can take an enormous amount of effort, saying nothing about validating what the text actually said in all possible scenarios.

If you absolutely have to write automated End-2-End tests, you can do it with Espresso as we did in the last step. You can also tests your tests with Firebase Test Lab.

The recommendation, if possible, would be to leave End-2-End tests to a QA team. They tend to have entire matrices of running different combinations of tests with different inputs and looking for regressions. You would be better off finding things before it got that far.

#

- Current Repo: v1-Unit-Testing
- Continue to Advanced UI -->