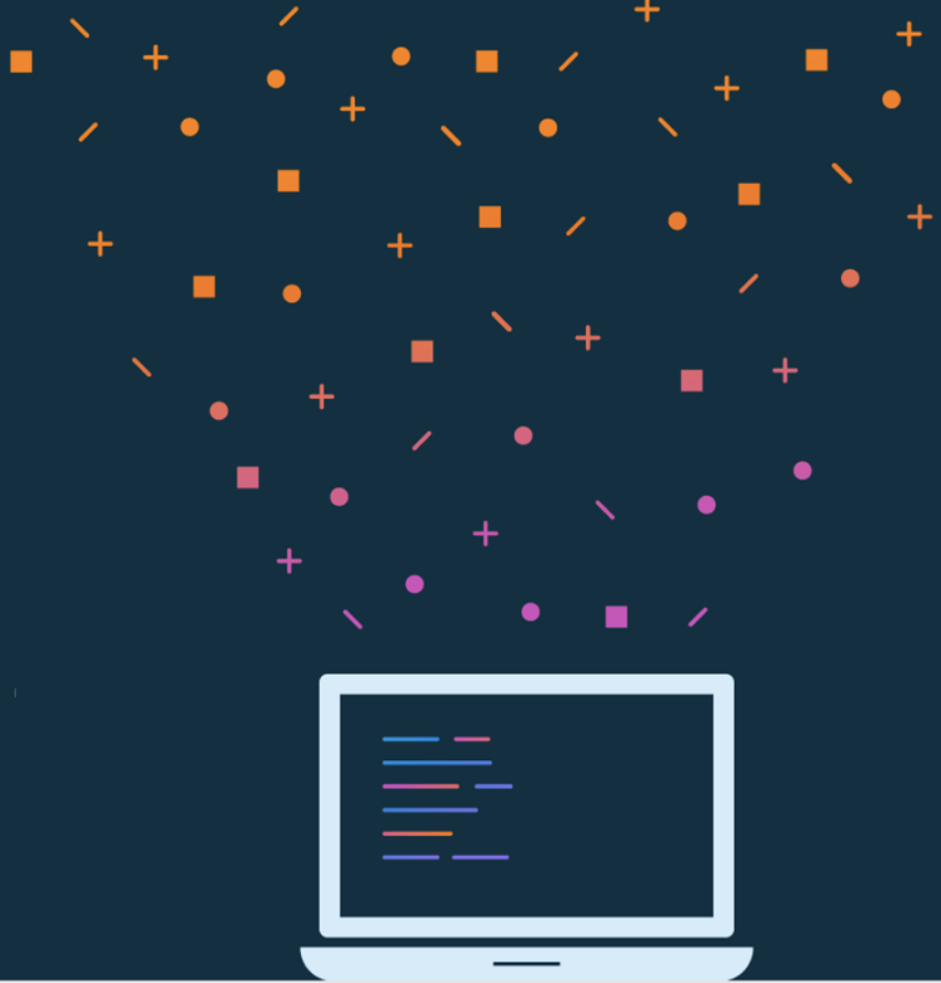
Lesson 5: Layouts



About this lesson

Lesson 5: Layouts

- Layouts in Android
- ConstraintLayout
- Additional topics for ConstraintLayout
- Data binding
- Displaying lists with RecyclerView
- Summary

Layouts in Android

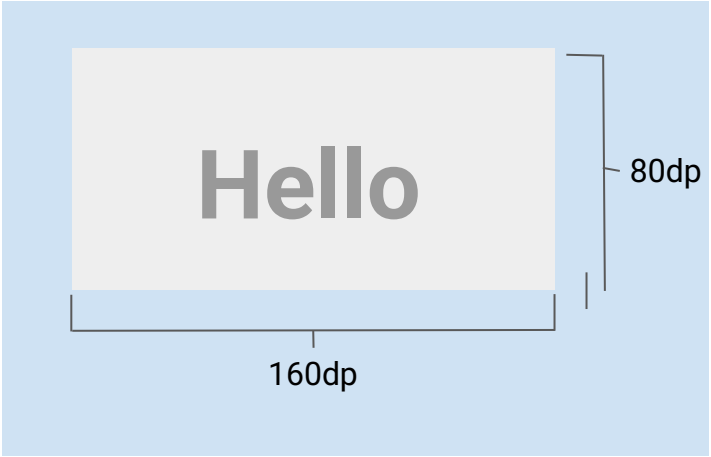
Android devices

- Android devices come in many different form factors.
- More and more pixels per inch are being packed into device screens.
- Developers need the ability to specify layout dimensions that are consistent across devices.

Density-independent pixels (dp)

Use dp when specifying sizes in your layout, such as the width or height of views.

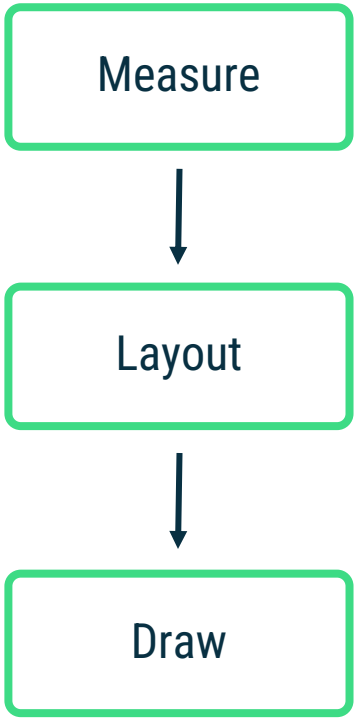
- Density-independent pixels (dp) take screen density into account.
- Android views are measured in density-independent pixels.
- dp = $\frac{\text{(width in pixels * 160)}}{\text{screen density}}$



Screen-density buckets

Density qualifier	Description	DPI estimate
ldpi (mostly unused)	Low density	~120dpi
mdpi (baseline density)	Medium density	~160dpi
hdpi	High density	~240dpi
xhdpi	Extra-high density	~320dpi
xxhdpi	Extra-extra-high density	~480dpi
xxxhdpi	Extra-extra-extra-high density	~640dpi

Android View rendering cycle



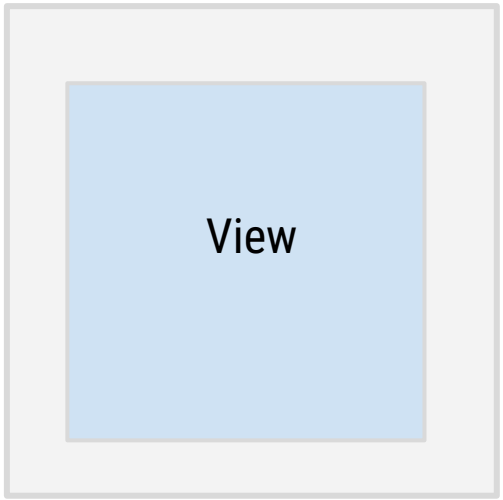
Drawing region

What we see:

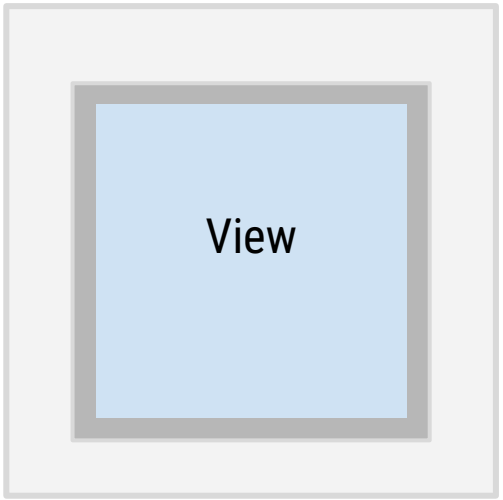
How it's drawn:

View margins and padding

View with margin



View with margin and padding



ConstraintLayout

Deeply nested layouts are costly

- Deeply nested ViewGroups require more computation
- Views may be measured multiple times
- Can cause UI slowdown and lack of responsiveness

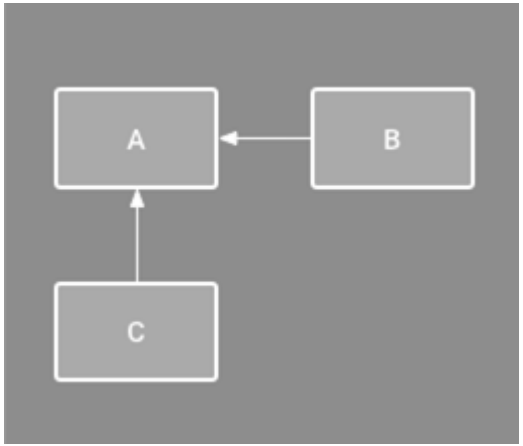
Use ConstraintLayout to avoid some of these issues!

What is ConstraintLayout?

- Recommended default layout for Android
- Solves costly issue of too many nested layouts, while allowing complex behavior
- Position and size views within it using a set of constraints

What is a constraint?

A restriction or limitation on the properties of a View that the layout attempts to respect



Relative positioning constraints

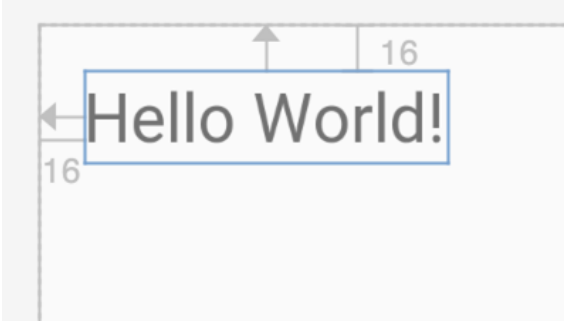
Can set up a constraint relative to the parent container

Format:

`layout_constraint<`**`SourceConstraint`**`>_to<`**`TargetConstraint`**`>Of`

Example attributes on a TextView:

```
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintLeft_toLeftOf="parent"
```

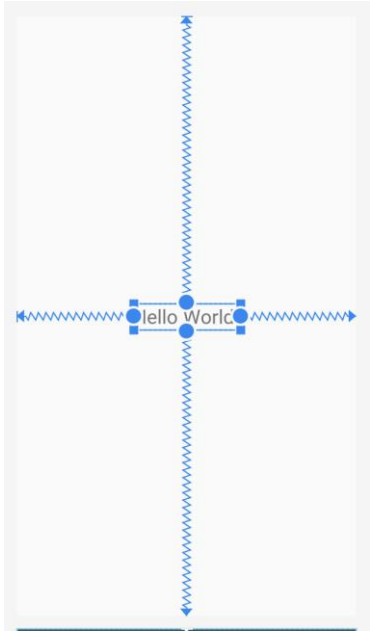


Relative positioning constraints

Relative positioning constraints

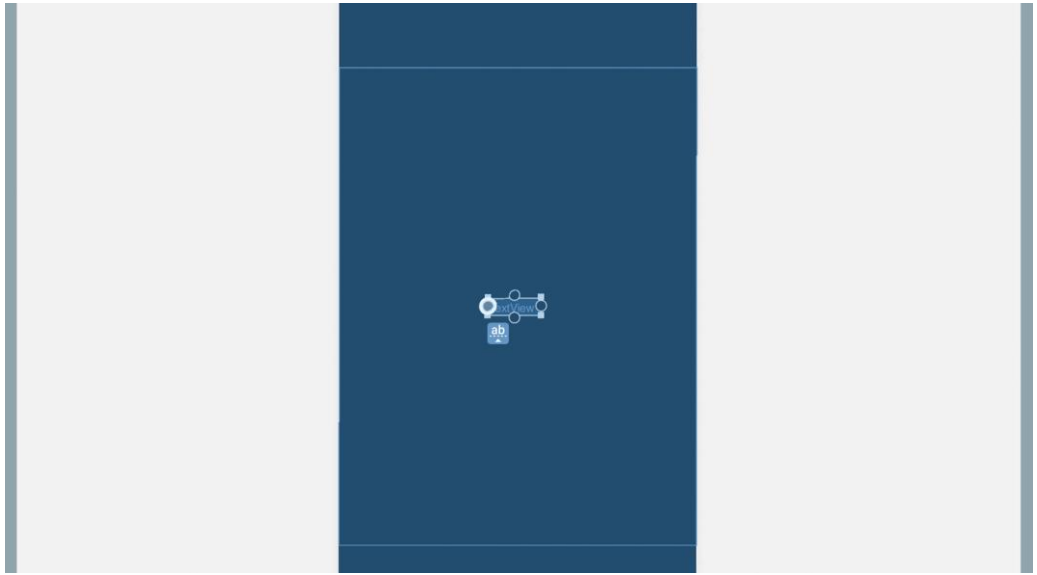
Simple ConstraintLayout example

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

    <TextView
        ...

app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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



Layout Editor in Android Studio

You can click and drag to add constraints to a View.

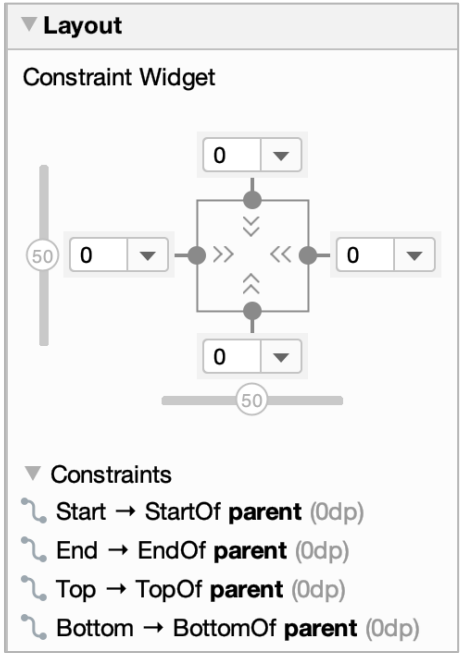


Constraint Widget in Layout Editor

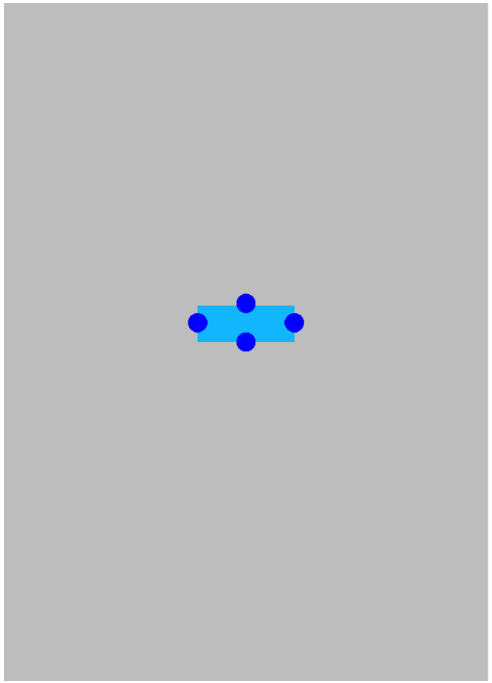
Fixed

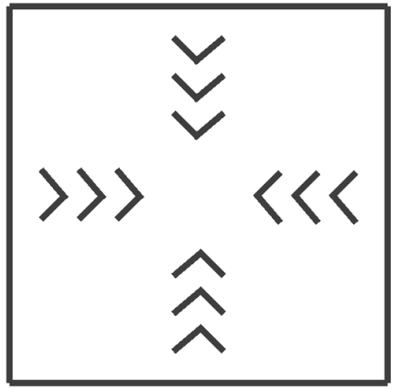
Wrap content

Match constraints



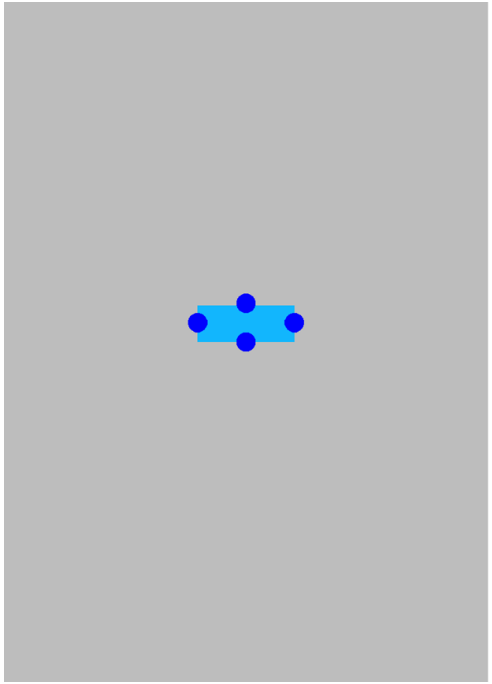
Wrap content for width and height

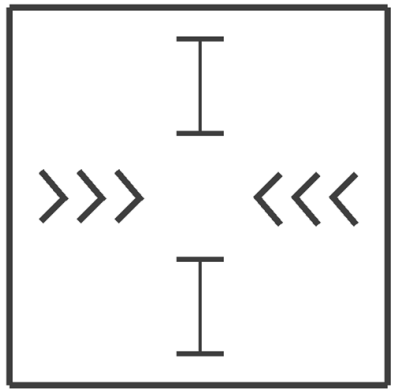




```
layout_width    wrap_content
layout_height   wrap_content
```

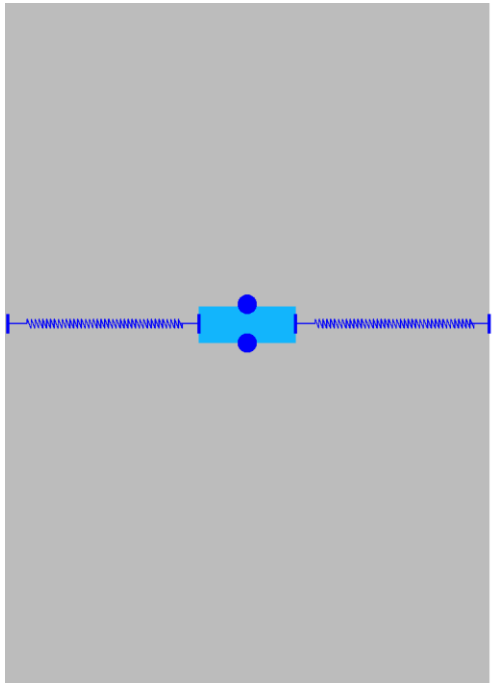
Wrap content for width, fixed height

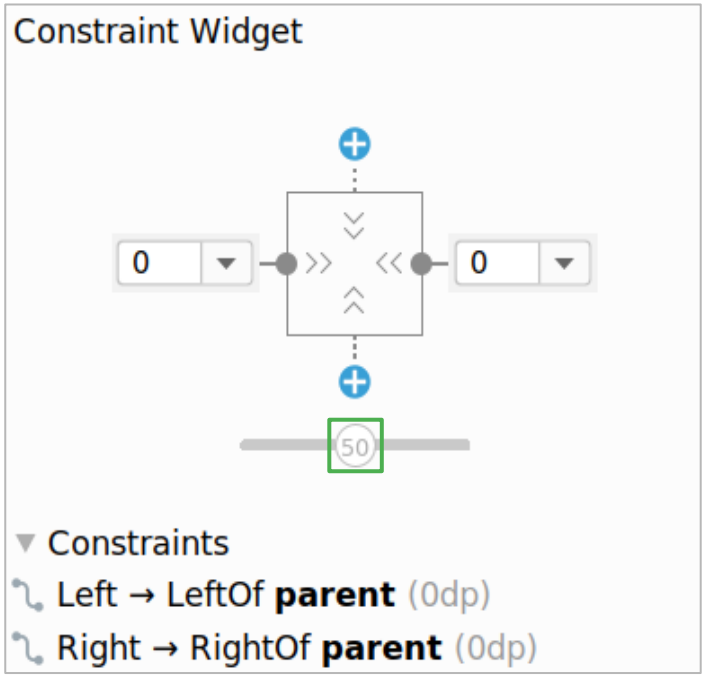




```
layout_width    wrap_content
layout_height   48dp
```

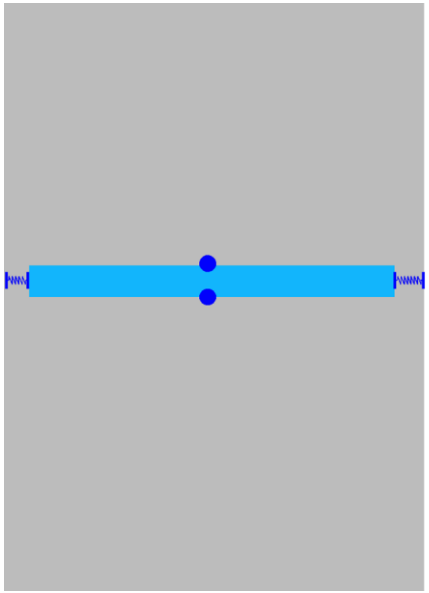
Center a view horizontally

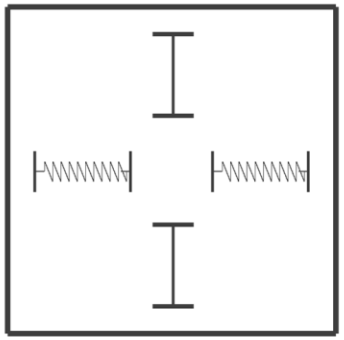




Use match_constraint

Can’t use `match_parent` on a child view, use `match_constraint` instead





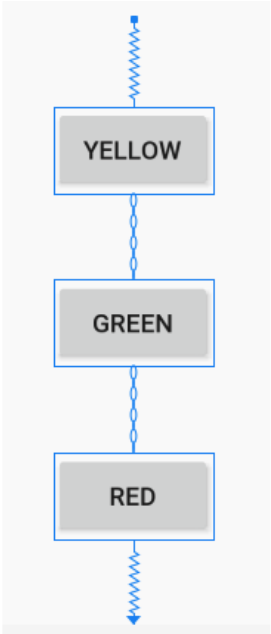
```
layout_width    0dp(match_constraint)
layout_height   48dp
```

Chains

- Let you position views in relation to each other
- Can be linked horizontally or vertically
- Provide much of LinearLayout functionality

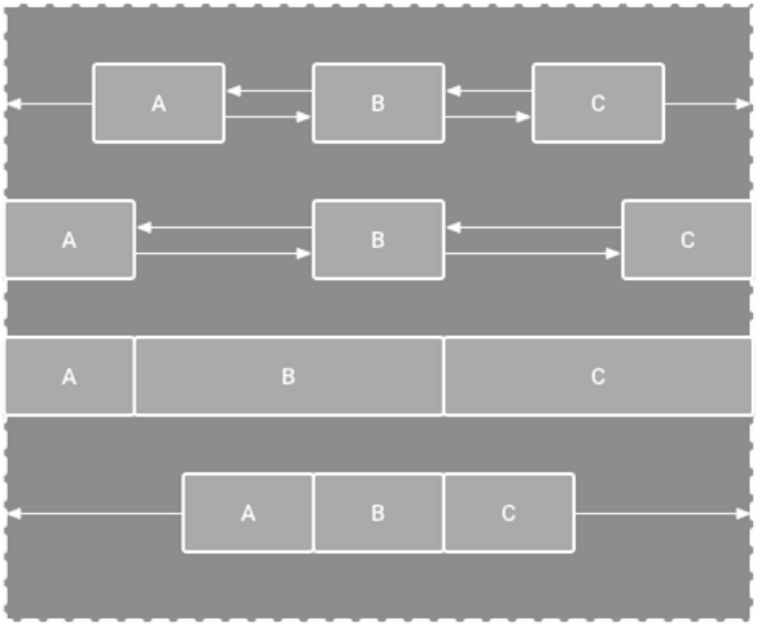
Create a Chain in Layout Editor

1. Select the objects you want to be in the chain.
2. Right-click and select **Chains.**
3. Create a horizontal or vertical chain.



Chain styles

Adjust space between views with these different chain styles.



Spread Chain

Spread Inside Chain

Weighted Chain

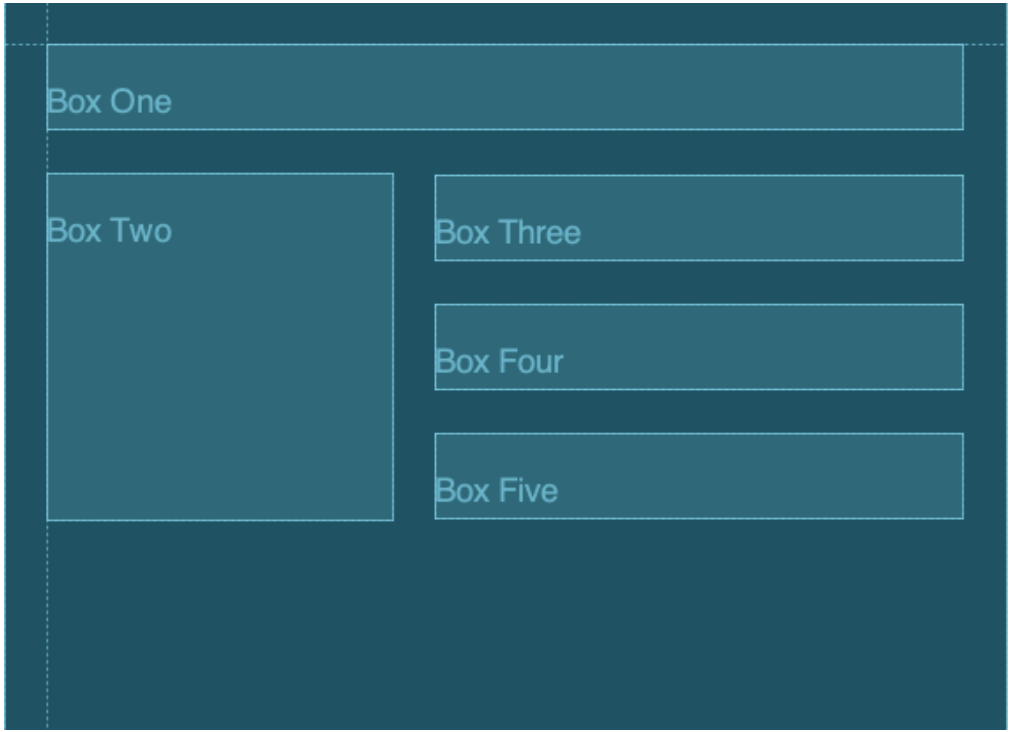
Packed Chain

Additional topics for ConstraintLayout

Guidelines

- Let you position multiple views relative to a single guide
- Can be vertical or horizontal
- Allow for greater collaboration with design/UX teams
- Aren't drawn on the device

Guidelines in Android Studio



Example Guideline

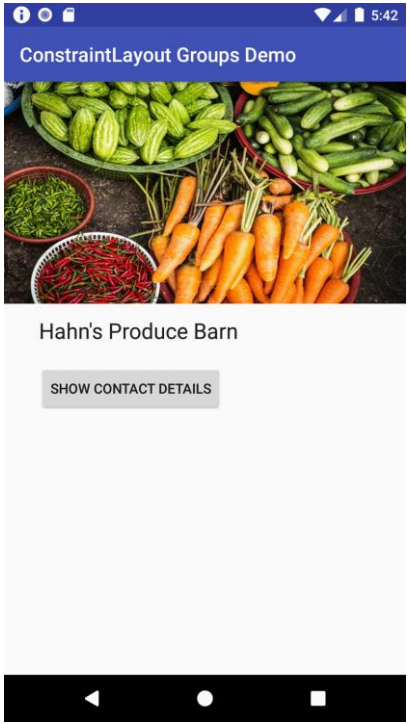
```
<ConstraintLayout>
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/start_guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_begin="16dp" />
    <TextView ...
        app:layout_constraintStart_toEndOf="@id/start_guideline" />
</ConstraintLayout>
```

Creating Guidelines

- `layout_constraintGuide_begin`
- `layout_constraintGuide_end`
- `layout_constraintGuide_percent`

Groups

- Control the visibility of a set of widgets
- Group visibility can be toggled in code



Example group

```
<androidx.constraintlayout.widget.Group
    android:id="@+id/group"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:constraint_referenced_ids="locationLabel,locationDetails"/>
```

Groups app code

```
override fun onClick(v: View?) {
    if (group.visibility == View.GONE) {
        group.visibility = View.VISIBLE
        button.setText(R.string.hide_details)
    } else {
        group.visibility = View.GONE
        button.setText(R.string.show_details)
    }
}
```

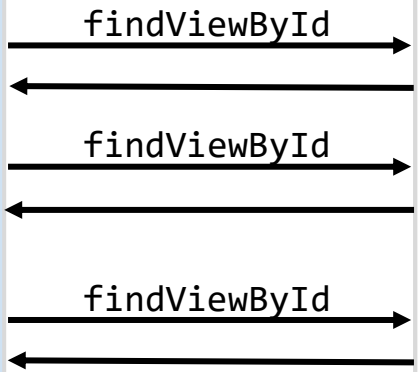
Data binding

Current approach: findViewById()

Traverses the `View` hierarchy each time

MainActivity.kt

```
val name = findViewById(...)
val age = findViewById(...)
val loc = findViewById(...)

name.text = …
age.text = …
loc.text = …
```



activity_main.xml

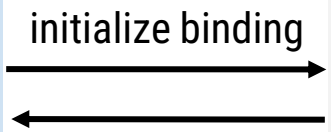
```
<ConstraintLayout … >
  <TextView
      android:id="@+id/name"/>
  <TextView
      android:id="@+id/age"/>
  <TextView
      android:id="@+id/loc"/>
</ConstraintLayout>
```

Use data binding instead

Bind UI components in your layouts to data sources in your app.

MainActivity.kt

```
Val binding:ActivityMainBinding

binding.name.text = …
binding.age.text = …
binding.loc.text = …
```



activity_main.xml

```
<layout>
   <ConstraintLayout … >
       <TextView
          android:id="@+id/name"/>
       <TextView
          android:id="@+id/age"/>
       <TextView
          android:id="@+id/loc"/>
   </ConstraintLayout>
</layout>
```

Modify build.gradle file

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

Add layout tag

```
<layout>
    <androidx.constraintlayout.widget.ConstraintLayout>
        <TextView ... android:id="@+id/username" />
        <EditText ... android:id="@+id/password" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Layout inflation with data binding

Replace this

```
setContentView(R.layout.activity_main)
```

with this

```
val binding: ActivityMainBinding = DataBindingUtil.setContentView(
    this, R.layout.activity_main)

binding.username = "Melissa"
```

Data binding layout variables

```
<layout>
    <data>
        <variable name="name" type="String"/>
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout>
        <TextView
            android:id="@+id/textView"
            android:text="@{name}" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

In `MainActivity.kt`:

```
binding.name = "John"
```

Data binding layout expressions

```
<layout>
    <data>
        <variable name="name" type="String"/>
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout>
        <TextView
            android:id="@+id/textView"
            android:text="@{name.toUpperCase()}" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Displaying lists with RecyclerView

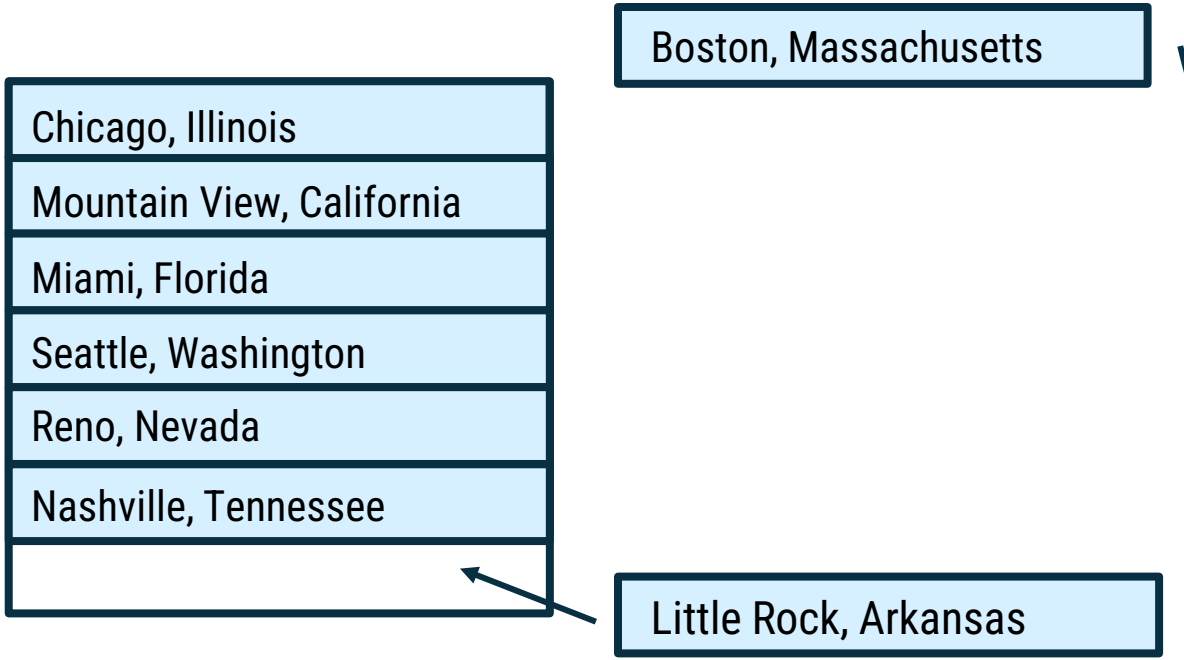
RecyclerView

- Widget for displaying lists of data
- "Recycles" (reuses) item views to make scrolling more performant
- Can specify a list item layout for each item in the dataset
- Supports animations and transitions

RecyclerView.Adapter

- Supplies data and layouts that the RecyclerView displays
- A custom Adapter extends from `RecyclerView.Adapter` and overrides these three functions:
  - `getItemCount`
  - `onCreateViewHolder`
  - `onBindViewHolder`

View recycling in RecyclerView



If item is scrolled offscreen, it isn't destroyed. Item is put in a pool to be recycled.

onBindViewHolder binds the view with the new values, and then the view gets reinserted in the list.

Add RecyclerView to your layout

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/rv"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Create a list item layout

res/layout/item_view.xml

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TextView
        android:id="@+id/number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</FrameLayout>
```

Create a list adapter

```
class MyAdapter(val data: List<Int>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>()
{
    class MyViewHolder(val row: View) : RecyclerView.ViewHolder(row) {
        val textView = row.findViewById<TextView>(R.id.number)
    }

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
        val layout = LayoutInflater.from(parent.context).inflate(R.layout.item_view,
                     parent, false)
        return MyViewHolder(layout)
    }
    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        holder.textView.text = data.get(position).toString()
    }
    override fun getItemCount(): Int = data.size
```

Set the adapter on the RecyclerView

In `MainActivity.kt`:

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

    val rv: RecyclerView = findViewById(R.id.rv)
    rv.layoutManager = LinearLayoutManager(this)

    rv.adapter = MyAdapter(IntRange(0, 100).toList())
}
```

Summary

Summary

In Lesson 5, you learned how to:

- Specify lengths in dp for your layout
- Work with screen densities for different Android devices
- Render Views to the screen of your app
- Layout views within a ConstraintLayout using constraints
- Simplify getting View references from layout with data binding
- Display a list of text items using a RecyclerView and custom adapter

Learn more

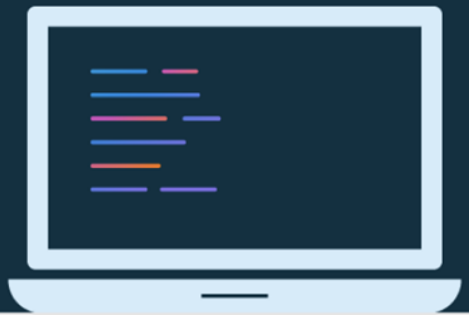
- Pixel density on Android
- Spacing
- Device metrics
- Type scale
- Build a Responsive UI with ConstraintLayout
- Data Binding Library
- Create dynamic lists with RecyclerView

Pathway

Practice what you've learned by completing the pathway:

Lesson 5: Layouts

Lesson 6: App navigation



About this lesson

Lesson 6: App navigation

- Multiple activities and intents
- App bar, navigation drawer, and menus
- Fragments
- Navigation in an app
- More custom navigation behavior
- Navigation UI
- Summary

Multiple activities and intents

Multiple screens in an app

Sometimes app functionality may be separated into multiple screens.

Examples:

- View details of a single item (for example, product in a shopping app)
- Create a new item (for example, new email)
- Show settings for an app
- Access services in other apps (for example, photo gallery or browse documents)

Intent

Requests an action from another app component, such as another Activity

- An `Intent` usually has two primary pieces of information:
  - Action to be performed (for example, `ACTION_VIEW`, `ACTION_EDIT`, `ACTION_MAIN`)
  - Data to operate on (for example, a person's record in the contacts database)
- Commonly used to specify a request to transition to another Activity

Explicit intent

- Fulfills a request **using a specific component**
- Navigates internally to an Activity in your app
- Navigates to a specific third-party app or another app you've written

Explicit intent examples

Navigate between activities in your app:

```
fun viewNoteDetail() {
    val intent = Intent(this, NoteDetailActivity::class.java)
    intent.putExtra(NOTE_ID, note.id)
    startActivity(intent)
}
```

Navigate to a specific external app:

```
fun openExternalApp() {
    val intent = Intent("com.example.workapp.FILE_OPEN")
    if (intent.resolveActivity(packageManager) != null) {
        startActivity(intent)
    }
}
```

Implicit intent

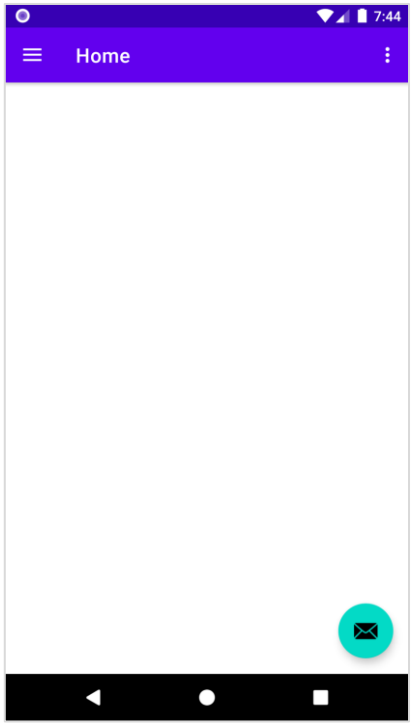
- Provides generic action the app can perform
- Resolved using mapping of the data type and action to known components
- Allows any app that matches the criteria to handle the request

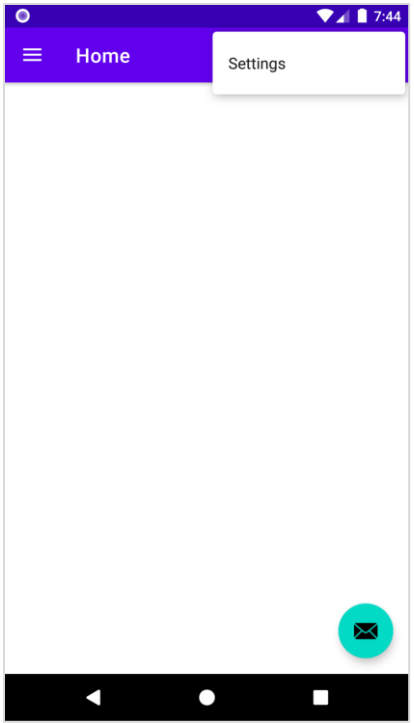
Implicit intent example

```
fun sendEmail() {
    val intent = Intent(Intent.ACTION_SEND)
    intent.type = "text/plain"
    intent.putExtra(Intent.EXTRA_EMAIL, emailAddresses)
    intent.putExtra(Intent.EXTRA_TEXT, "How are you?")

    if (intent.resolveActivity(packageManager) != null) {
        startActivity(intent)
    }
}
```

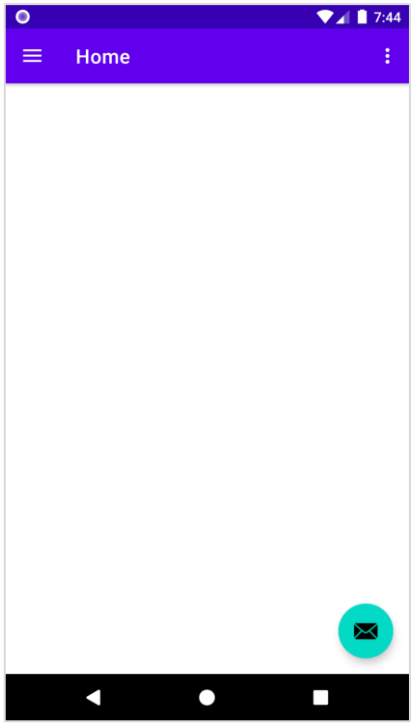
App bar, navigation drawer, and menus

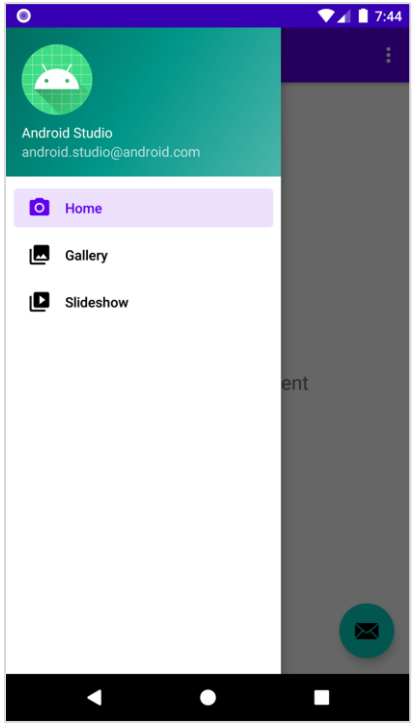
App bar





Navigation drawer





Menu

Define menu items in XML menu resource (located in `res/menu` folder)

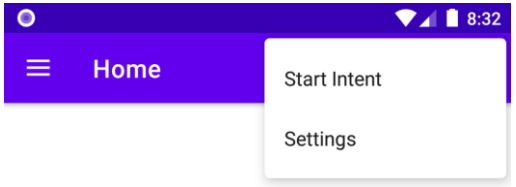
```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="@string/action_settings"
        app:showAsAction="never" />
</menu>
```

More menu options

```
<menu>
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_home"
            android:icon="@drawable/ic_menu_camera"
            android:title="@string/menu_home" />
        <item
            android:id="@+id/nav_gallery"
            android:icon="@drawable/ic_menu_gallery"
            android:title="@string/menu_gallery" />
        <item
            android:id="@+id/nav_slideshow"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="@string/menu_slideshow" />
    </group>
</menu>
```

Options menu example

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/action_intent"
        android:title="@string/action_intent" />

    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="@string/action_settings"
        app:showAsAction="never" />
</menu>
```



Inflate options menu

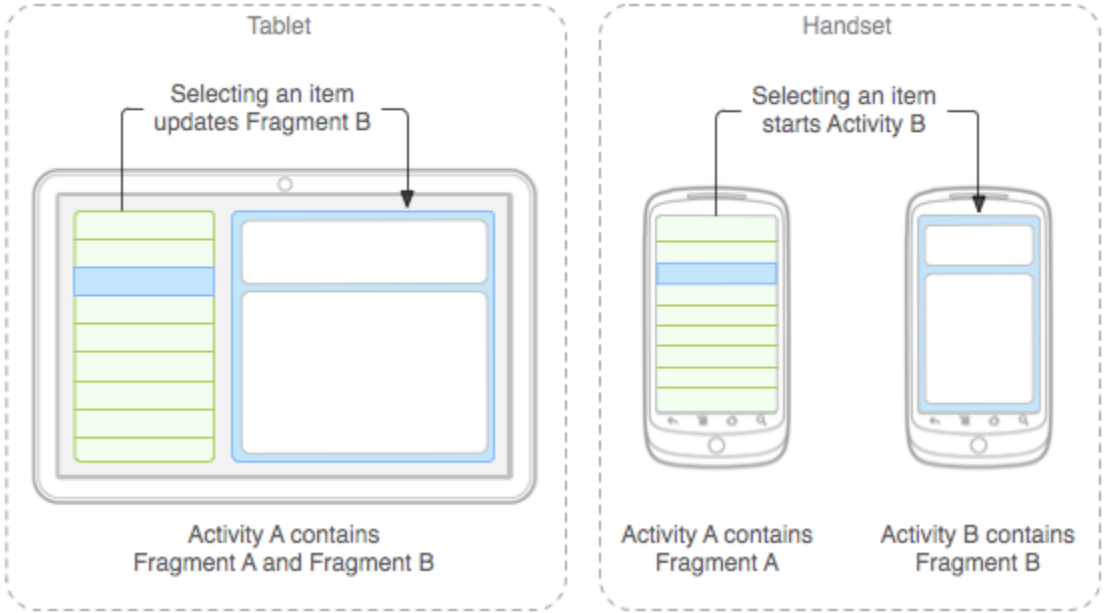
```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.main, menu)
    return true
}
```

Handle menu options selected

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {

    when (item.itemId) {
        R.id.action_intent -> {
            val intent = Intent(Intent.ACTION_WEB_SEARCH)
            intent.putExtra(SearchManager.QUERY, "pizza")
            if (intent.resolveActivity(packageManager) != null) {
                startActivity(intent)
            }
        }
        else -> Toast.makeText(this, item.title, Toast.LENGTH_LONG).show()
    ...
```

Fragments

Fragments for tablet layouts



Fragment

- Represents a behavior or portion of the UI in an activity ("microactivity")
- Must be hosted in an activity
- Lifecycle tied to host activity's lifecycle
- Can be added or removed at runtime

Note about fragments

Use the AndroidX version of the `Fragment` class. (`androidx.fragment.app.Fragment`).

Don't use the platform version of the `Fragment` class (`android.app.Fragment`), which was deprecated.

Navigation within an app

Navigation component

- Collection of libraries and tooling, including an integrated editor, for creating navigation paths through an app
- Assumes one `Activity` per graph with many `Fragment` destinations
- Consists of three major parts:
  - Navigation graph
  - Navigation Host (`NavHost`)
  - Navigation Controller (`NavController`)

Add dependencies

In `build.gradle`, under `dependencies`:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"

implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

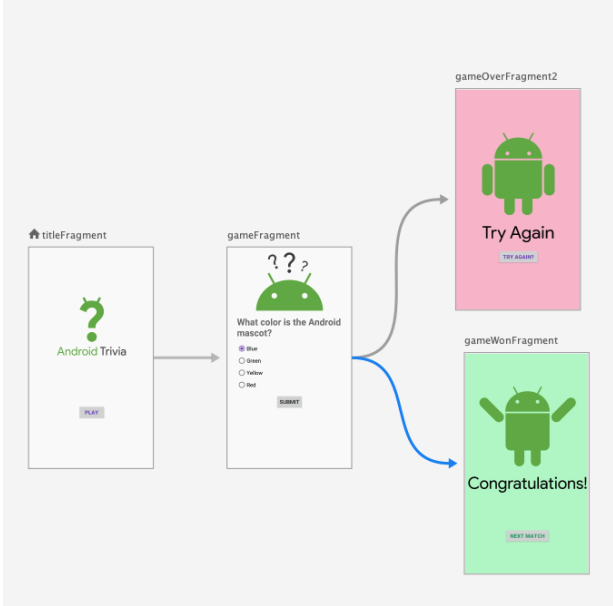
Navigation host (NavHost)

```
<fragment
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph_name"/>
```

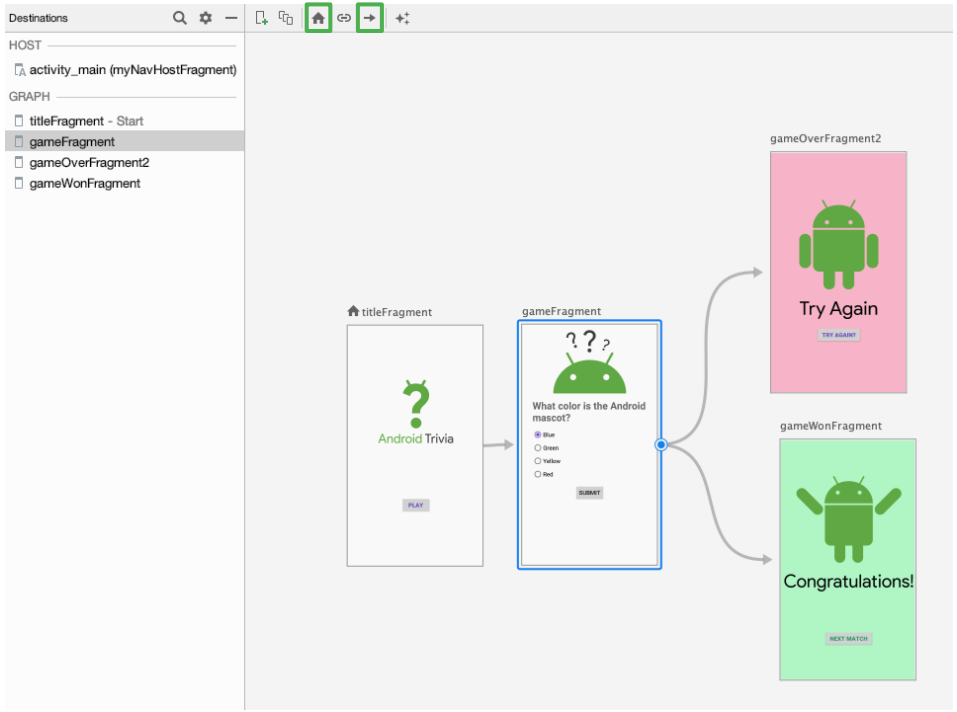
Navigation graph

New resource type located in `res/navigation` directory

- XML file containing all of your navigation destinations and actions
- Lists all the (Fragment/Activity) destinations that can be navigated to
- Lists the associated actions to traverse between them
- Optionally lists animations for entering or exiting



Navigation Editor in Android Studio



Creating a Fragment

- Extend `Fragment` class
- Override `onCreateView()`
- Inflate a layout for the Fragment that you have defined in XML

```
class DetailFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
            savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.detail_fragment, container, false)
    }
}
```

Specifying Fragment destinations

- Fragment destinations are denoted by the `action` tag in the navigation graph.
- Actions can be defined in XML directly or in the Navigation Editor by dragging from source to destination.
- Autogenerated action IDs take the form of `action_<sourceFragment>_to_<destinationFragment>`.

Example fragment destination

```
<fragment
    android:id="@+id/welcomeFragment"
    android:name="com.example.android.navigation.WelcomeFragment"
    android:label="fragment_welcome"
    tools:layout="@layout/fragment_welcome" >

    <action
        android:id="@+id/action_welcomeFragment_to_detailFragment"
        app:destination="@id/detailFragment" />

</fragment>
```

Navigation Controller (NavController)

`NavController` manages UI navigation in a navigation host.

- Specifying a destination path only names the action, but it doesn't execute it.
- To follow a path, use `NavController`.

Example NavController

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val navController = findNavController(R.id.myNavHostFragment)
    }

    fun navigateToDetail() {
        navController.navigate(R.id.action_welcomeFragment_to_detailFragment)
    }
}
```

More custom navigation behavior

Passing data between destinations

Using Safe Args:

- Ensures arguments have a valid type
- Lets you provide default values
- Generates a `<SourceDestination>Directions` class with methods for every action in that destination
- Generates a class to set arguments for every named action
- Generates a `<TargetDestination>Args` class providing access to the destination's arguments

Setting up Safe Args

In the project `build.gradle` file:

```
buildscript {
    repositories {
        google()
    }
    dependencies {
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

In the app's or module's `build.gradle` file:

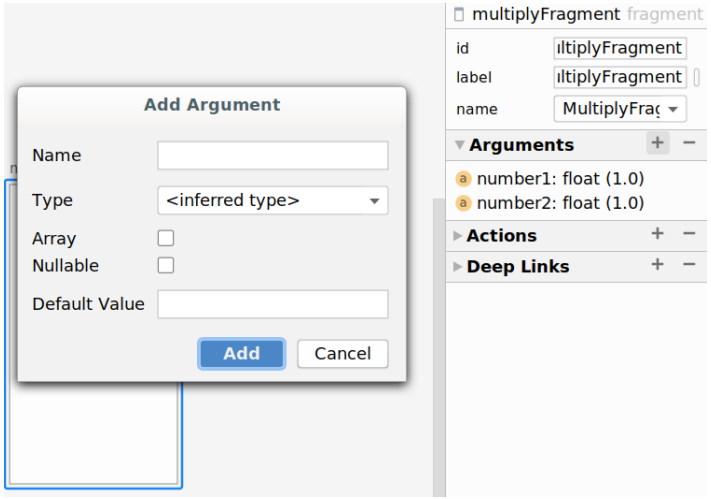
```
apply plugin: "androidx.navigation.safeargs.kotlin"
```

Sending data to a Fragment

1. Create arguments the destination fragment will expect.
2. Create action to link from source to destination.
3. Set the arguments in the action method on `<Source>FragmentDirections`.
4. Navigate according to that action using the Navigation Controller.
5. Retrieve the arguments in the destination fragment.

Destination arguments

```
<fragment
    android:id="@+id/multiplyFragment"
    android:name="com.example.arithmetic.MultiplyFragment"
    android:label="MultiplyFragment" >
    <argument
        android:name="number1"
        app:argType="float"
        android:defaultValue="1.0" />
    <argument
        android:name="number2"
        app:argType="float"
        android:defaultValue="1.0" />
</fragment>
```



Supported argument types

Type	**Type Syntax** `app:argType=<type>`	**Supports Default Values**	**Supports Null Values**
Integer	`"integer"`	Yes	No
Float	`"float"`	Yes	No
Long	`"long"`	Yes	No
Boolean	`"boolean"`	Yes (`"true"` or `"false"`)	No
String	`"string"`	Yes	Yes
Array	above type + `"[]"` (for example, `"string[]"` `"long[]"`)	Yes (only `"@null"`)	Yes
Enum	Fully qualified name of the enum	Yes	No
Resource reference	`"reference"`	Yes	No

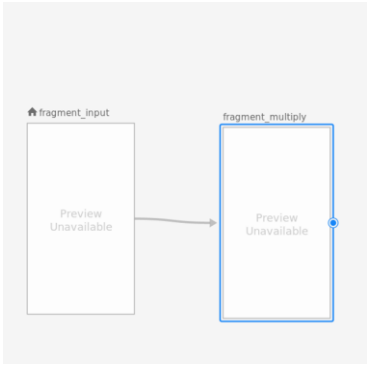
Supported argument types: Custom classes

Type	**Type Syntax** `app:argType=<type>`	**Supports Default Values**	**Supports Null Values**
Serializable	Fully qualified class name	Yes (only "`@null`")	Yes
Parcelable	Fully qualified class name	Yes (only "`@null`")	Yes

Create action from source to destination

In `nav_graph.xml`:

```
<fragment
    android:id="@+id/fragment_input"
    android:name="com.example.arithmetic.InputFragment">

    <action
        android:id="@+id/action_to_multiplyFragment"
        app:destination="@id/multiplyFragment" />

</fragment>
```



Navigating with actions

In `InputFragment.kt`:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding.button.setOnClickListener {
        val n1 = binding.number1.text.toString().toFloatOrNull() ?: 0.0
        val n2 = binding.number2.text.toString().toFloatOrNull() ?: 0.0

        val action = InputFragmentDirections.actionToMultiplyFragment(n1, n2)
        view.findNavController().navigate(action)
    }
}
```

Retrieving Fragment arguments

```
class MultiplyFragment : Fragment() {
    val args: MultiplyFragmentArgs by navArgs()
    lateinit var binding: FragmentMultiplyBinding
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        val number1 = args.number1
        val number2 = args.number2
        val result = number1 * number2
        binding.output.text = "${number1} * ${number2} = ${result}"
    }
}
```

Navigation UI

Menus revisited

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    val navController = findNavController(R.id.nav_host_fragment)
    return item.onNavDestinationSelected(navController) ||
            super.onOptionsItemSelected(item)
}
```

DrawerLayout for navigation drawer

```
<androidx.drawerlayout.widget.DrawerLayout
    android:id="@+id/drawer_layout" ...>

    <fragment
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:id="@+id/nav_host_fragment" ... />

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        app:menu="@menu/activity_main_drawer" ... />


</androidx.drawerlayout.widget.DrawerLayout>
```

Finish setting up navigation drawer

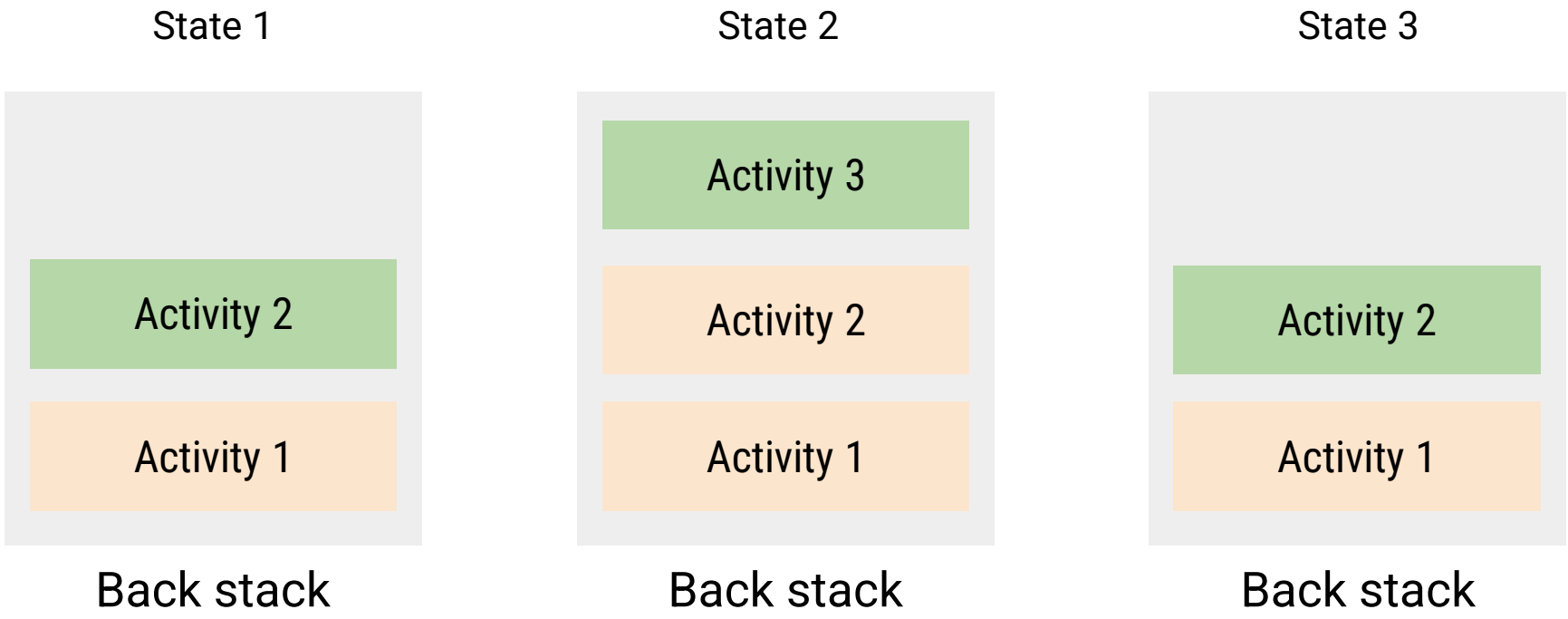
Connect `DrawerLayout` to navigation graph:

```
val appBarConfiguration = AppBarConfig(navController.graph, drawer)
```

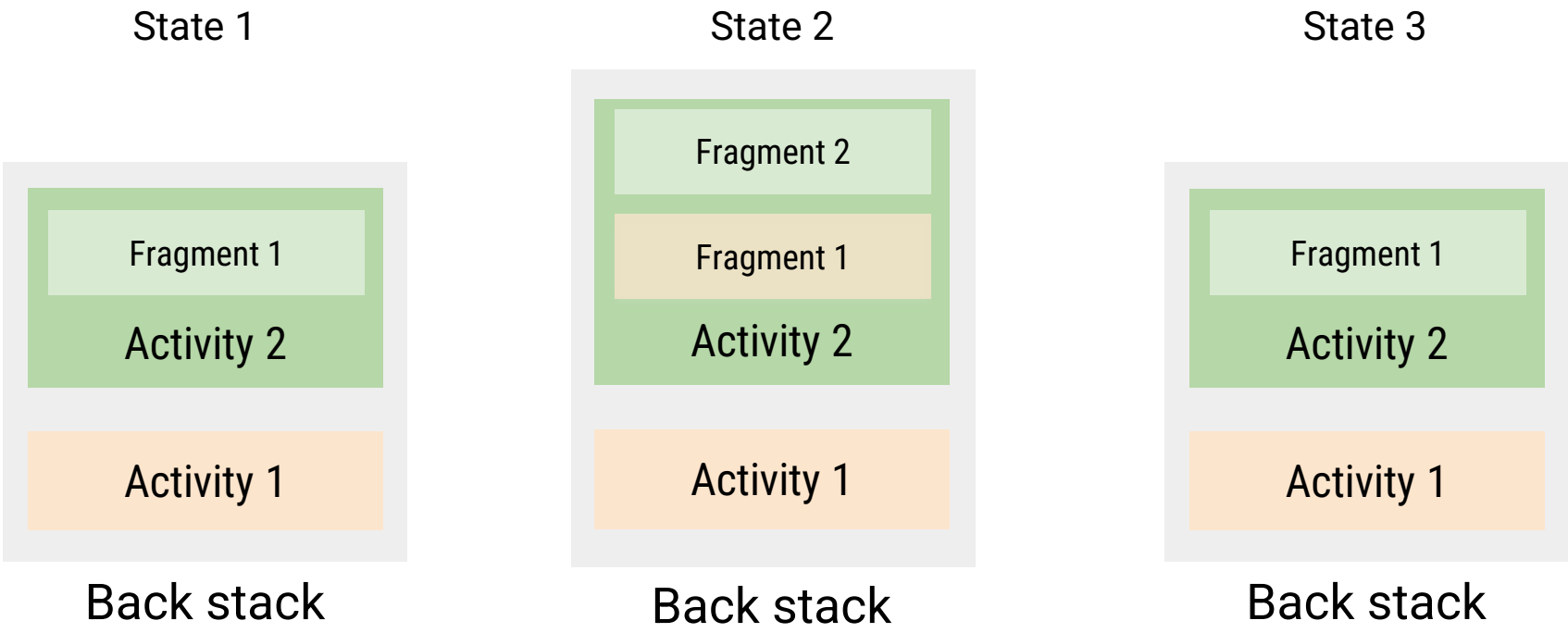
Set up `NavigationView` for use with a `NavController`:

```
val navView = findViewById<NavigationView>(R.id.nav_view)
navView.setupWithNavController(navController)
```

Understanding the back stack



Fragments and the back stack



Summary

Summary

In Lesson 6, you learned how to:

- Use explicit and implicit intents to navigate between Activities
- Structure apps using fragments instead of putting all UI code in the Activity
- Handle navigation with NavGraph, NavHost, and NavController
- Use Safe Args to pass data between fragment destinations
- Use NavigationUI to hook up top app bar, navigation drawer, and bottom navigation
- Android keeps a back stack of all the destinations you've visited, with each new destination being pushed onto the stack.

Learn more

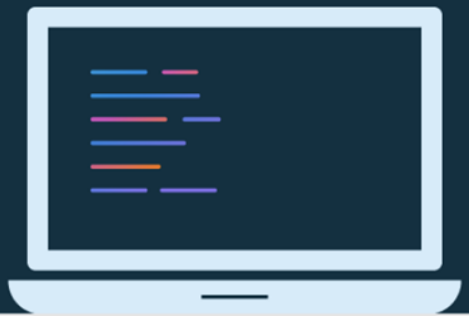
- Principles of navigation
- Navigation component
- Pass data between destinations
- NavigationUI

Pathway

Practice what you've learned by completing the pathway:

Lesson 6: App navigation

Lesson 7: Activity and fragment lifecycles



About this lesson

Lesson 7: Activity and fragment lifecycles

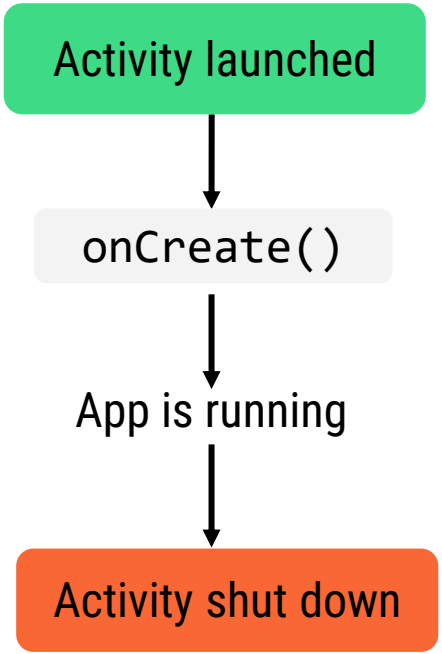
- Activity lifecycle
- Logging
- Fragment lifecycle
- Lifecycle-aware components
- Tasks and back stack
- Summary

Activity lifecycle

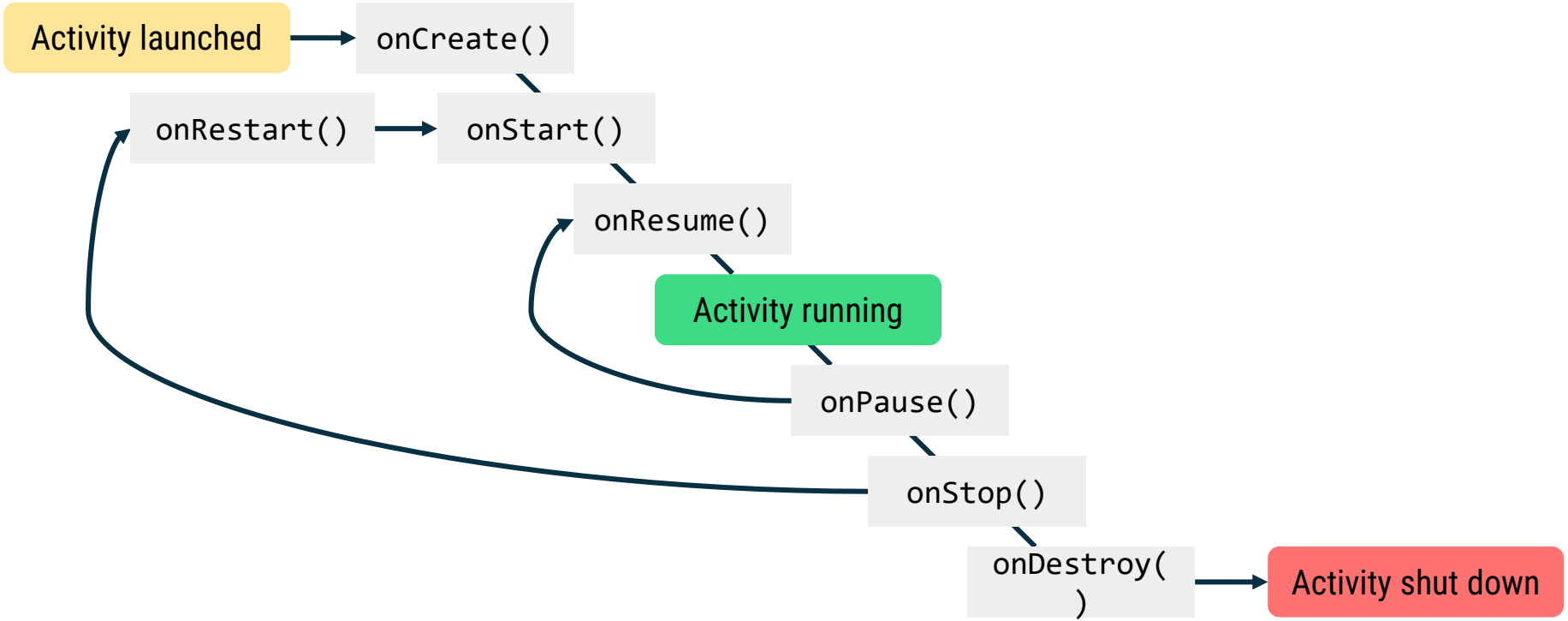
Why it matters

- Preserve user data and state if:
  - User temporarily leaves app and then returns
  - User is interrupted (for example, a phone call)
  - User rotates device
- Avoid memory leaks and app crashes.

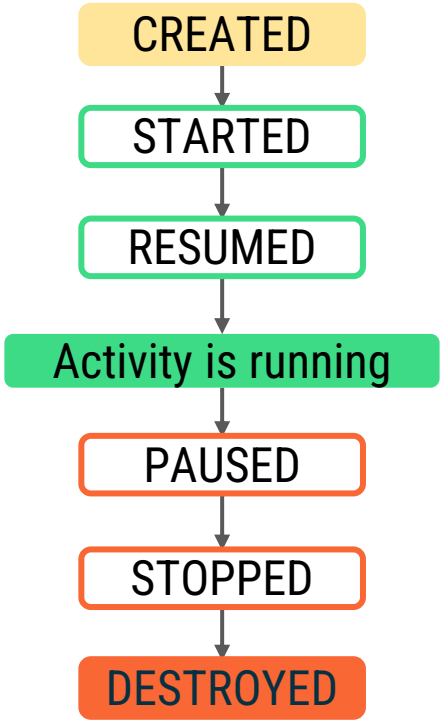
Simplified activity lifecycle



Activity lifecycle



Activity states



onCreate()

- Activity is created and other initialization work occurs
- You must implement this callback
- Inflate activity UI and perform other app startup logic

onStart()

- Activity becomes visible to the user
- Called after activity:
  - `onCreate()`

    or
  - `onRestart()` if activity was previously stopped

onResume()

- Activity gains input focus:
  - User can interact with the activity
- Activity stays in resumed state until system triggers activity to be paused

onPause()

- Activity has lost focus (not in foreground)
- Activity is still visible, but user is not actively interacting with it
- Counterpart to `onResume()`

onStop()

- Activity is no longer visible to the user
- Release resources that aren't needed anymore
- Save any persistent state that the user is in the process of editing so they don't lose their work

onDestroy()

- Activity is about to be destroyed, which can be caused by:
  - Activity has finished or been dismissed by the user
  - Configuration change
- Perform any final cleanup of resources.
- Don’t rely on this method to save user data (do that earlier)

Summary of activity states

State	Callbacks	Description
Created	`onCreate()`	Activity is being initialized.
Started	`onStart()`	Activity is visible to the user.
Resumed	`onResume()`	Activity has input focus.
Paused	`onPause()`	Activity does not have input focus.
Stopped	`onStop()`	Activity is no longer visible.
Destroyed	`onDestroy()`	Activity is destroyed.

Save state

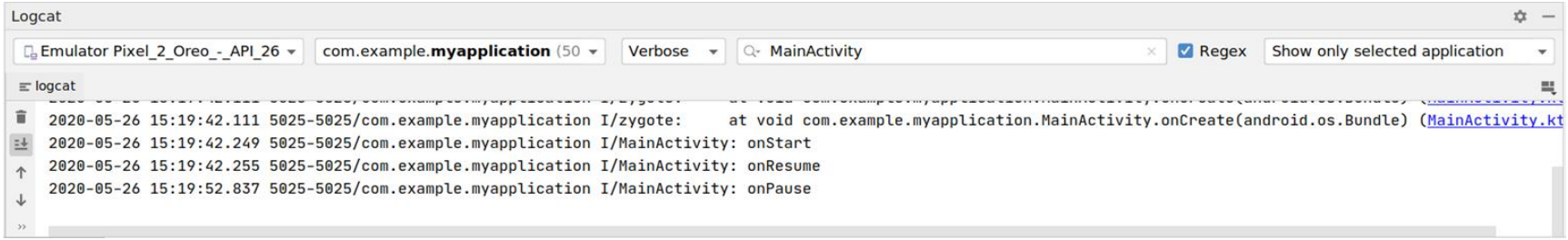
User expects UI state to stay the same after a config change or if the app is terminated when in the background.

- Activity is destroyed and restarted, or app is terminated and activity is started.
- Store user data needed to reconstruct app and activity Lifecycle changes:
  - Use `Bundle` provided by `onSaveInstanceState()`.
  - `onCreate()` receives the `Bundle` as an argument when activity is created again.

Logging

Logging in Android

- Monitor the flow of events or state of your app.
- Use the built-in `Log` class or third-party library.
- Example `Log` method call: `Log.d(TAG, "Message")`

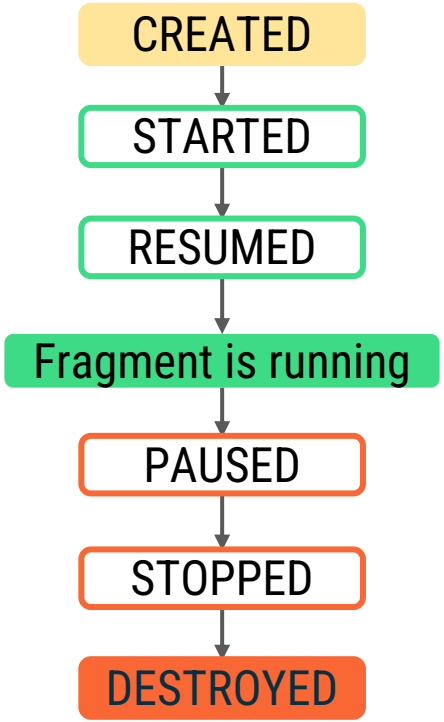


Write logs

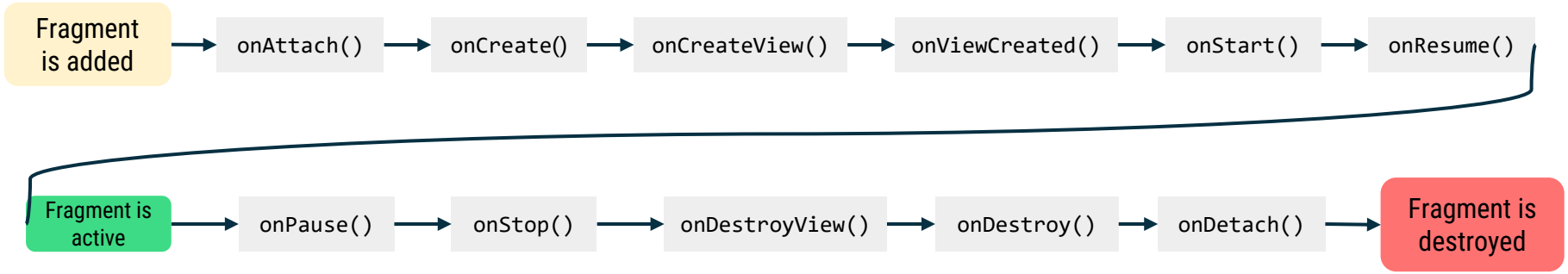
Priority level	Log method
Verbose	`Log.v(String, String)`
Debug	`Log.d(String, String)`
Info	`Log.i(String, String)`
Warning	`Log.w(String, String)`
Error	`Log.e(String, String)`

Fragment lifecycle

Fragment states



Fragment lifecycle diagram



onAttach()

- Called when a fragment is attached to a context
- Immediately precedes `onCreate()`

onCreateView()

- Called to create the view hierarchy associated with the fragment
- Inflate the fragment layout here and return the root view

onViewCreated()

- Called when view hierarchy has already been created
- Perform any remaining initialization here (for example, restore state from `Bundle`)

onDestroyView() and onDetach()

- `onDestroyView()` is called when view hierarchy of fragment is removed.
- `onDetach()` is called when fragment is no longer attached to the host.

Summary of fragment states

State	Callbacks	Description
Initialized	`onAttach()`	Fragment is attached to host.
Created	`onCreate()`, `onCreateView()`, `onViewCreated()`	Fragment is created and layout is being initialized.
Started	`onStart()`	Fragment is started and visible.
Resumed	`onResume()`	Fragment has input focus.
Paused	`onPause()`	Fragment no longer has input focus.
Stopped	`onStop()`	Fragment is not visible.
Destroyed	`onDestroyView()`, `onDestroy()`, `onDetach()`	Fragment is removed from host.

Save fragment state across config changes

Preserve UI state in fragments by storing state in `Bundle`:

- `onSaveInstanceState(outState: Bundle)`

Retrieve that data by receiving the Bundle in these fragment callbacks:

- `onCreate()`
- `onCreateView()`
- `onViewCreated()`

Lifecycle-aware components

Lifecycle-aware components

Adjust their behavior based on activity or fragment lifecycle

- Use the `androidx.lifecycle` library
- `Lifecycle` tracks the lifecycle state of an activity or fragment
  - Holds current lifecycle state
  - Dispatches lifecycle events (when there are state changes)

LifecycleOwner

- Interface that says this class has a lifecycle
- Implementers must implement `getLifecycle()` method

  Examples: `Fragment` and `AppCompatActivity` are implementations of `LifecycleOwner`

LifecycleObserver

Implement `LifecycleObserver` interface:

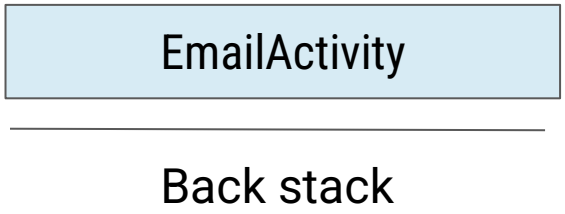
```
class MyObserver : LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    fun connectListener() {
        ...
    }
```

Add the observer to the lifecycle:

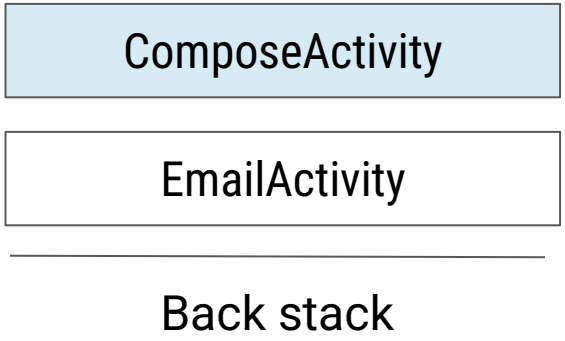
```
myLifecycleOwner.getLifecycle().addObserver(MyObserver())
```

Tasks and back stack

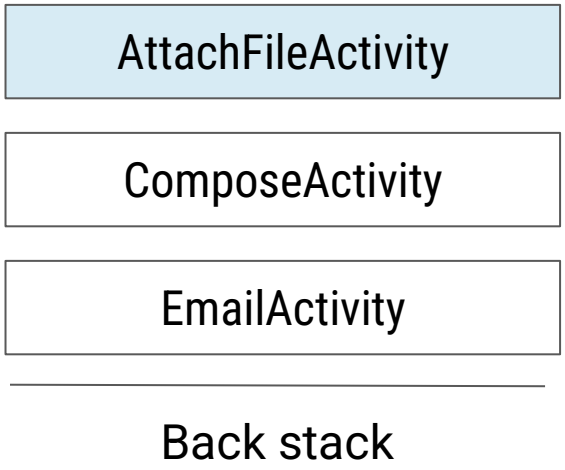
Back stack of activities



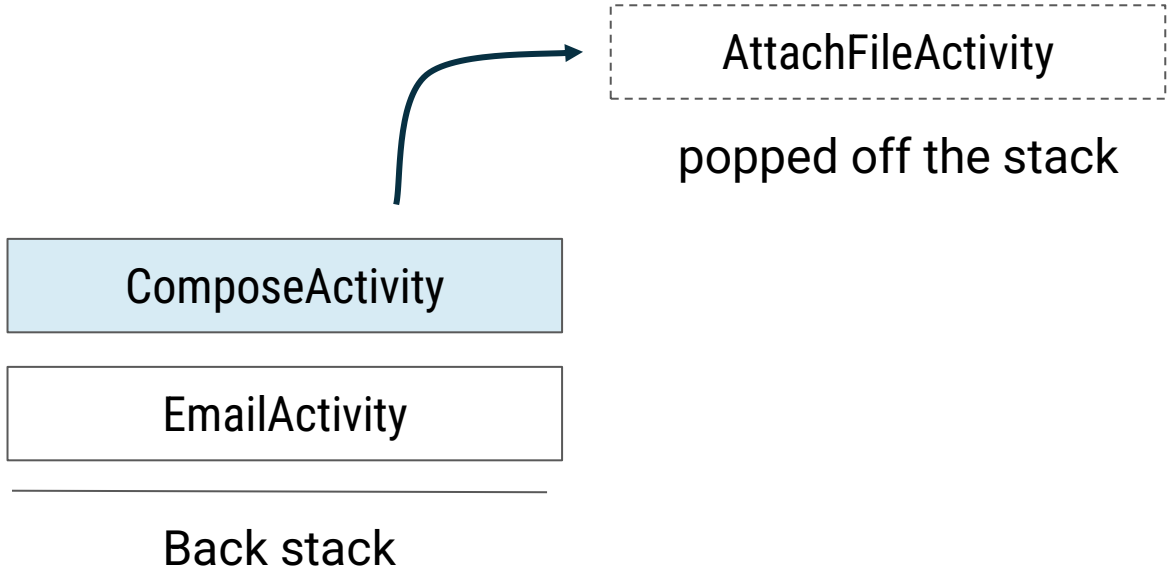
Add to the back stack



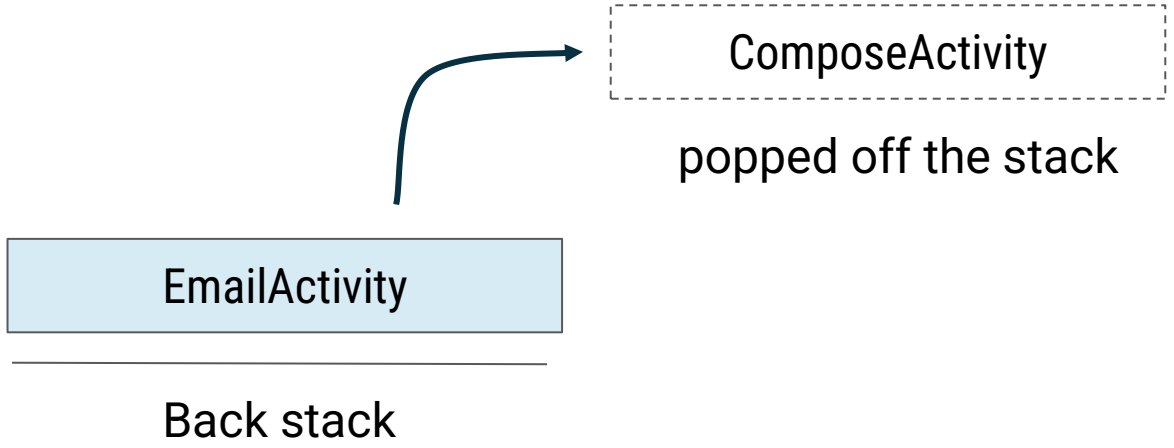
Add to the back stack again



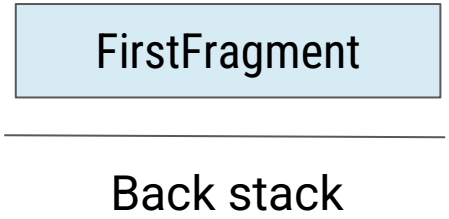
Tap Back button



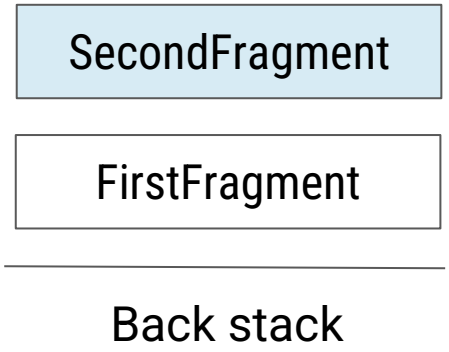
Tap Back button again



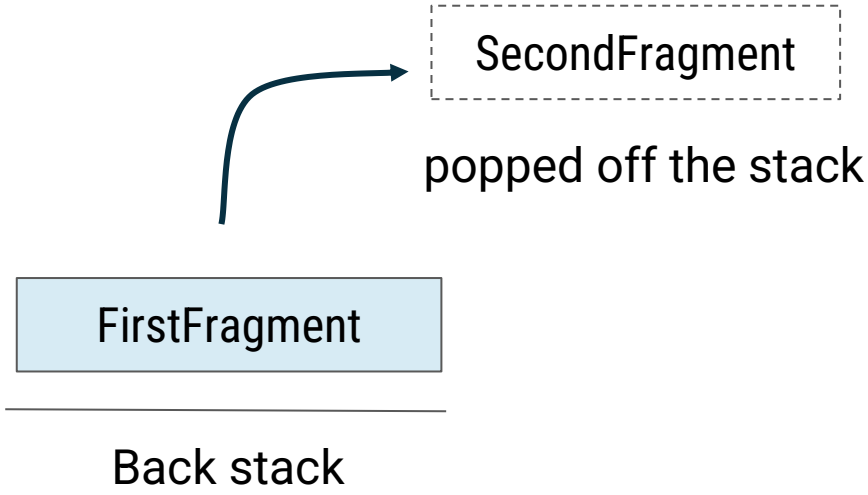
First destination in the back stack



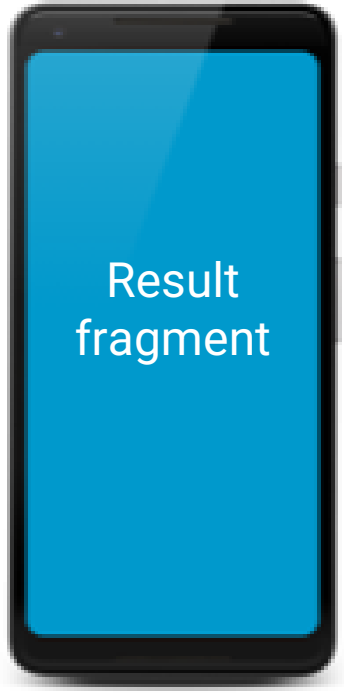
Add a destination to the back stack

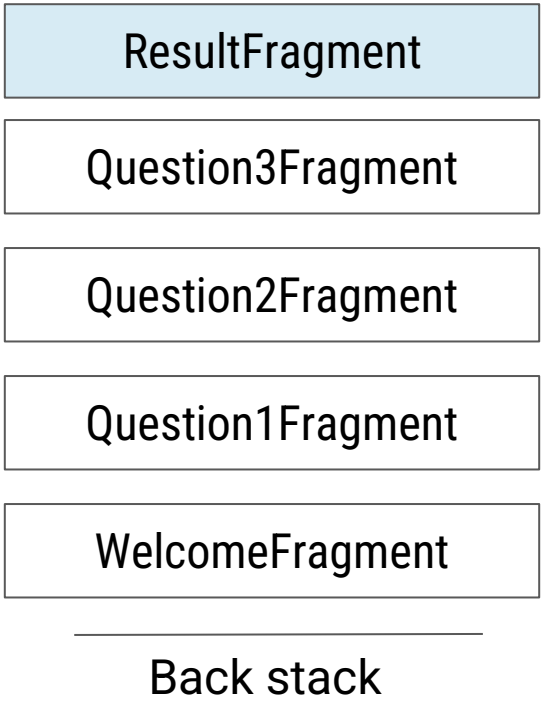


Tap Back button



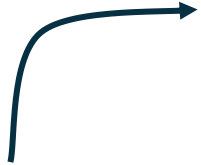
Another back stack example





Modify Back button behavior

pop additional destinations
off the back stack



Summary

Summary

In Lesson 7, you learned how to:

- Understand how an activity instance transitions through different lifecycle states as the user interacts with or leaves your app
- Reserve UI state across configuration changes using a `Bundle`
- Fragment lifecycle callback methods similar to activity, but with additions
- Use lifecycle-aware components help organize your app code
- Use default or custom back stack behavior
- Use logging to help debug and track the state of the app

Learn more

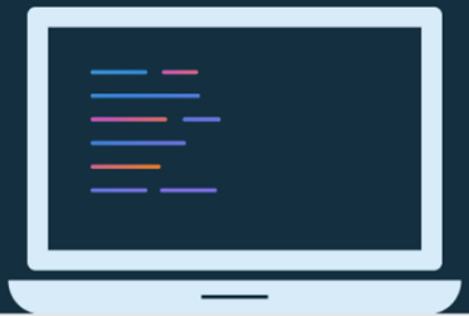
- Understand the Activity Lifecycle
- Activity class
- Fragments guide and lifecycle
- Fragment class

Pathway

Practice what you've learned by completing the pathway:

Lesson 7: Activity and Fragment Lifecycles

Lesson 8: App architecture (UI layer)



About this lesson

Lesson 8: App architecture (UI layer)

- Android app architecture
- ViewModel
- Data binding
- LiveData
- Transform LiveData
- Summary

Android app architecture

Avoid short-term hacks

- External factors, such as tight deadlines, can lead to poor decisions about app design and structure.
- Decisions have consequences for future work (app can be harder to maintain long-term).
- Need to balance on-time delivery and future maintenance burden.

Examples of short-term hacks

- Tailoring your app to a specific device
- Blindly copying and pasting code into your files
- Placing all business logic in activity file
- Hardcoding user-facing strings in your code

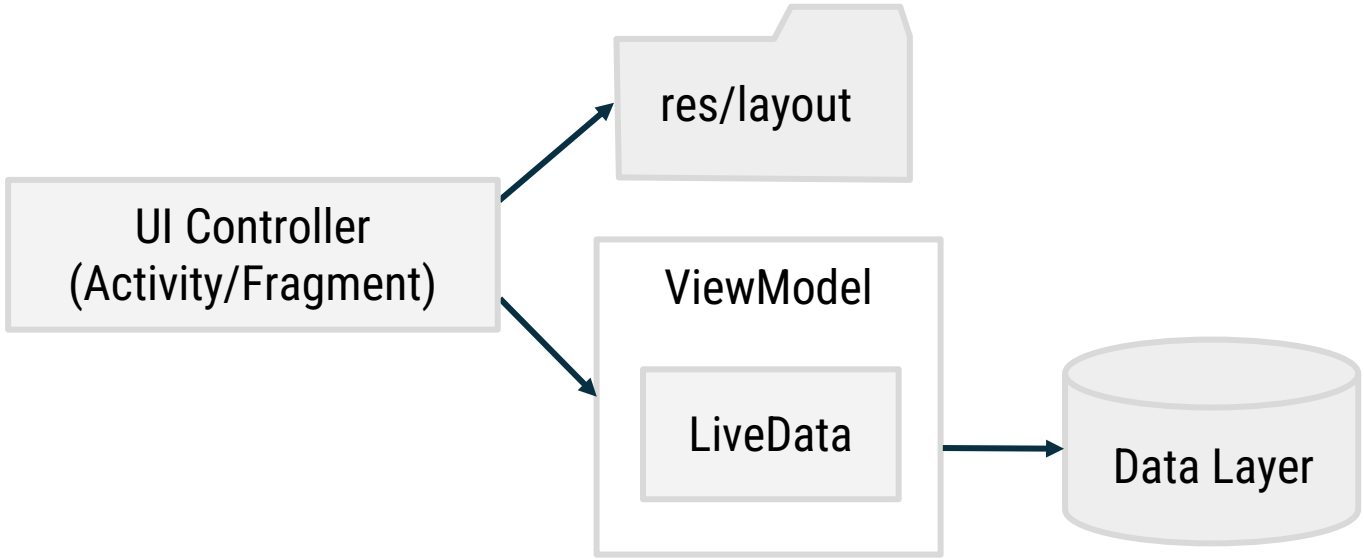
Why you need good app architecture

- Clearly defines where specific business logic belongs
- Makes it easier for developers to collaborate
- Makes your code easier to test
- Lets you benefit from already-solved problems
- Saves time and reduces technical debt as you extend your app

Android Jetpack

- Android libraries that incorporate best practices and provide backward compatibility in your apps
- Jetpack comprises the `androidx.*` package libraries

Separation of concerns



Architecture components

- Architecture design patterns, like MVVM and MVI, describe a loose template for what the structure of your app should be.
- Jetpack architecture components help you design robust, testable, and maintainable apps.

ViewModel

Gradle: lifecycle extensions

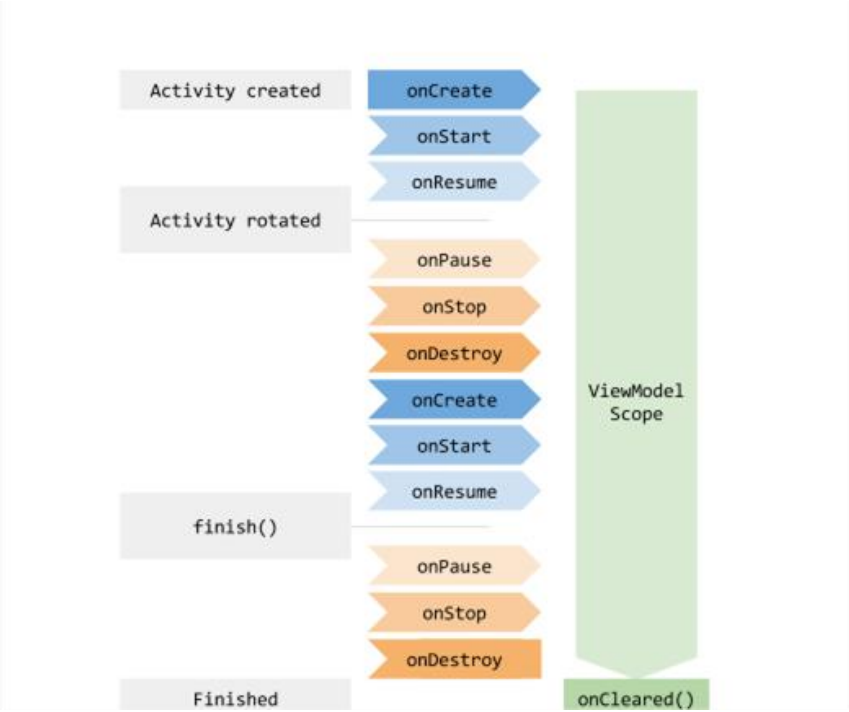
In `app/build.gradle` file:

```
dependencies {
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.activity:activity-ktx:$activity_version"
}
```

ViewModel

- Prepares data for the UI
- Must not reference activity, fragment, or views in view hierarchy
- Scoped to a lifecycle (which activity and fragment have)
- Enables data to survive configuration changes
- Survives as long as the scope is alive

Lifetime of a ViewModel



Kabaddi Kounter

ViewModel class

`abstract class ViewModel`

Summary

Public constructors	
`<init>()`	ViewModel is a class that is responsible for preparing and managing the data for an `Activity` or a `Fragment`.

Protected methods	
open Unit	`onCleared()` This method will be called when this ViewModel is no longer used and will be destroyed.

Extension properties	
From androidx.lifecycle	
CoroutineScope	`viewModelScope` CoroutineScope tied to this ViewModel.

Implement a ViewModel

```
class ScoreViewModel : ViewModel() {
    var scoreA : Int = 0
    var scoreB : Int = 0
    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            scoreA++
        }
        else {
            scoreB++
        }
    }
}
```

Load and use a ViewModel

```
class MainActivity : AppCompatActivity() {
    // Delegate provided by androidx.activity.viewModels
    val viewModel: ScoreViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val scoreViewA: TextView = findViewById(R.id.scoreA)
        scoreViewA.text = viewModel.scoreA.toString()
    }
```

Using a ViewModel

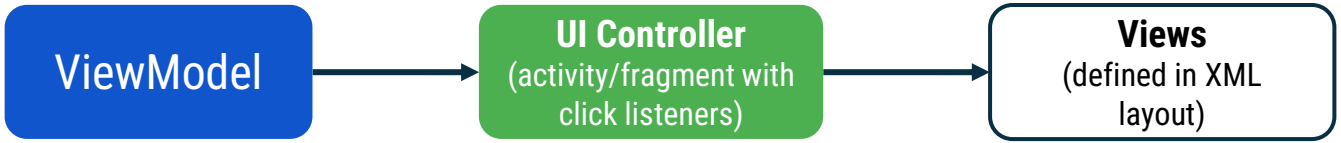
Within `MainActivity onCreate()`:

```
val scoreViewA: TextView = findViewById(R.id.scoreA)
val plusOneButtonA: Button = findViewById(R.id.plusOne_teamA)

plusOneButtonA.setOnClickListener {
    viewModel.incrementScore(true)
    scoreViewA.text = viewModel.scoreA.toString()
}
```

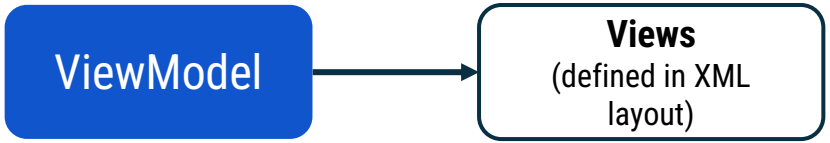
Data binding

ViewModels and data binding

- App architecture without data binding



- ViewModels can work in concert with data binding



Data binding in XML revisited

Specify ViewModels in the `data` tag of a binding.

```
<layout>
    <data>
        <variable>
            name="viewModel"
            type="com.example.kabaddikounter.ScoreViewModel" />
    </data>
    <ConstraintLayout ../>
</layout>
```

Attaching a ViewModel to a data binding

```
class MainActivity : AppCompatActivity() {

    val viewModel: ScoreViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding: ActivityMainBinding = DataBindingUtil.setContentView(this,
            R.layout.activity_main)

        binding.viewModel = viewModel

        ...
```

Using a ViewModel from a data binding

In `activity_main.xml`:

```
<TextView
    android:id="@+id/scoreViewA"
    android:text="@{viewModel.scoreA.toString()}" />
        ...
```

ViewModels and data binding

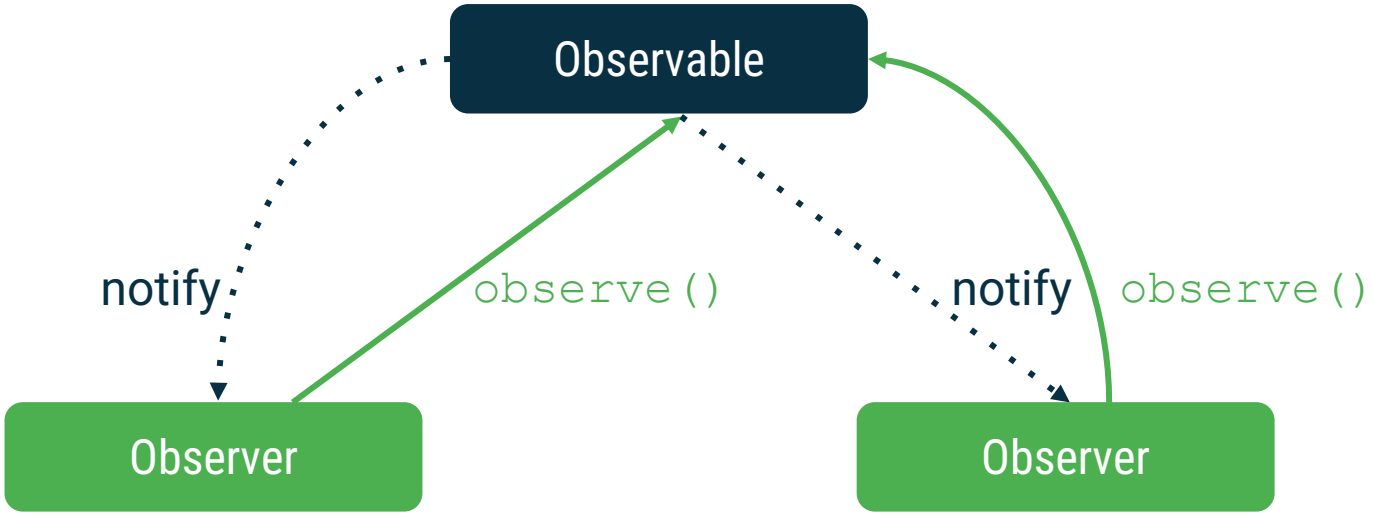
```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val binding: ActivityMainBinding = DataBindingUtil.setContentView(this,
        R.layout.activity_main)

    binding.plusOneButtonA.setOnClickListener {
        viewModel.incrementScore(true)
        binding.scoreViewA.text = viewModel.scoreA.toString()
    }
}
```

LiveData

Observer design pattern

- Subject maintains list of observers to notify when state changes.
- Observers receive state changes from subject and execute appropriate code.
- Observers can be added or removed at any time.

Observer design pattern diagram



LiveData

- A lifecycle-aware data holder that can be observed
- Wrapper that can be used with any data including lists (for example, `LiveData<Int>` holds an `Int`)
- Often used by ViewModels to hold individual data fields
- Observers (activity or fragment) can be added or removed
  - `observe(owner: LifecycleOwner, observer: Observer)`
    `removeObserver(observer: Observer)`

LiveData versus MutableLiveData

`LiveData<T>`	`MutableLiveData<T>`
• `getValue()`	• `getValue()` • `postValue(value: T)` • `setValue(value: T)`

`T` is the type of data that's stored in `LiveData` or `MutableLiveData`.

Use LiveData in ViewModel

```
class ScoreViewModel : ViewModel() {

    private val _scoreA = MutableLiveData<Int>(0)
    val scoreA: LiveData<Int>
        get() = _scoreA

    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            _scoreA.value = _scoreA.value!! + 1
        }
        ...
```

Add an observer on LiveData

Set up click listener to increment `ViewModel` score:

```
binding.plusOneButtonA.setOnClickListener {
    viewModel.incrementScore(true)
}
```

Create observer to update team A score on screen:

```
val scoreA_Observer = Observer<Int> { newValue ->
    binding.scoreViewA.text = newValue.toString()
}
```

Add the observer onto `scoreA LiveData` in `ViewModel`:

```
viewModel.scoreA.observe(this, scoreA_Observer)
```

Two-way data binding

- We already have two-way binding with `ViewModel` and `LiveData`.
- Binding to `LiveData` in XML eliminates need for an observer in code.

Example layout XML

```
<layout>
    <data>
        <variable>
            name="viewModel"
            type="com.example.kabaddikounter.ScoreViewModel" />
    </data>

    <ConstraintLayout ..>
        <TextView ...
            android:id="@+id/scoreViewA"
            android:text="@{viewModel.scoreA.toString()}" />
        ...
    </ConstraintLayout>
</layout>
```

Example Activity

```
class MainActivity : AppCompatActivity() {
    val viewModel: ScoreViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding: ActivityMainBinding = DataBindingUtil
             .setContentView(this, R.layout.activity_main)

        binding.viewModel = viewModel
        binding.lifecycleOwner = this

        binding.plusOneButtonA.setOnClickListener {
            viewModel.incrementScore(true)
        }
        ...
```

Example ViewModel

```
class ScoreViewModel : ViewModel() {
    private val _scoreA = MutableLiveData<Int>(0)
    val scoreA : LiveData<Int>
        get() = _scoreA
    private val _scoreB = MutableLiveData<Int>(0)
    val scoreB : LiveData<Int>
        get() = _scoreB
    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            _scoreA.value = _scoreA.value!! + 1
        } else {
            _scoreB.value = _scoreB.value!! + 1
        }
    }
}
```

Transform LiveData

Manipulating LiveData with transformations

`LiveData` can be transformed into a new `LiveData` object.

- `map()`
- `switchMap()`

Example LiveData with transformations

```
val result: LiveData<String> = Transformations.map(viewModel.scoreA) {
    x -> if (x > 10) "A Wins" else ""
}
```

Summary

Summary

In Lesson 8, you learned how to:

- Follow good app architecture design, and the separation-of-concerns principle to make apps more maintainable and reduce technical debt
- Create a `ViewModel` to hold data separately from a UI controller
- Use `ViewModel` with data binding to make a responsive UI with less code
- Use observers to automatically get updates from `LiveData`

Learn More

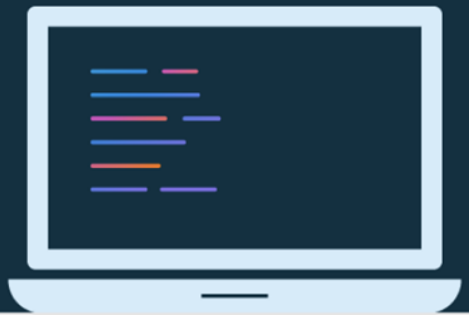
- Guide to app architecture
- Android Jetpack
- ViewModel Overview
- Android architecture sample app
- ViewModelProvider
- Lifecycle Aware Data Loading with Architecture Components
- ViewModels and LiveData: Patterns + AntiPatterns

Pathway

Practice what you've learned by completing the pathway:

Lesson 8: App architecture (UI layer)

Lesson 10: Advanced RecyclerView use cases



About this lesson

Lesson 10: Advanced `RecyclerView` use cases

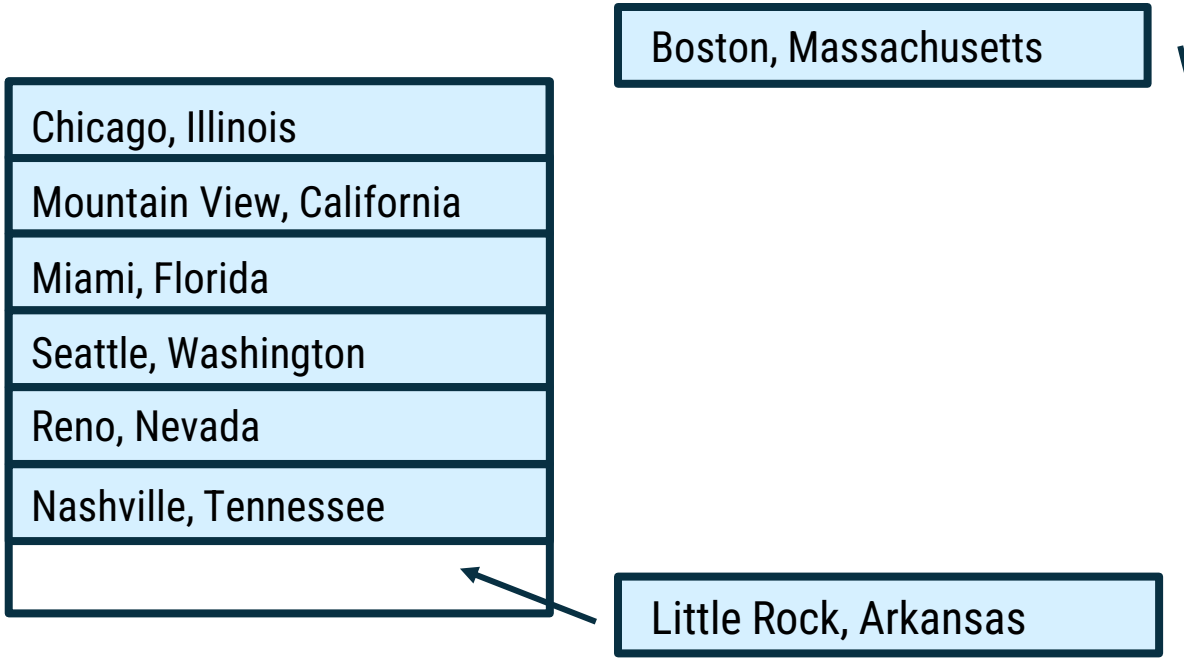
- RecyclerView recap
- Advanced binding
- Multiple item view types
- Headers
- Grid layout
- Summary

RecyclerView recap

RecyclerView overview

- Widget for displaying lists of data
- "Recycles" (reuses) item views to make scrolling more performant
- Can specify a list item layout for each item in the dataset
- Supports animations and transitions

View recycling in RecyclerView



If item is scrolled offscreen, it isn't destroyed. Item is put in a pool to be recycled.

onBindViewHolder binds the view with the new values, and then the view gets reinserted in the list.

RecyclerViewDemo app

Adapter for RecyclerViewDemo

```
class NumberListAdapter(var data: List<Int>):
        RecyclerView.Adapter<NumberListAdapter.IntViewHolder>() {
    class IntViewHolder(val row: View): RecyclerView.ViewHolder(row) {
        val textView = row.findViewById<TextView>(R.id.number)
    }
```

Functions for RecyclerViewDemo

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        IntViewHolder {
    val layout = LayoutInflater.from(parent.context)
        .inflate(R.layout.item_view, parent, false)
    return IntViewHolder(layout)
}

override fun onBindViewHolder(holder: IntViewHolder, position: Int) {
    holder.textView.text = data.get(position).toString()
}
```

Set the adapter onto the RecyclerView

In `MainActivity.kt`:

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

    val rv: RecyclerView = findViewById(R.id.rv)
    rv.layoutManager = LinearLayoutManager(this)

    rv.adapter = NumberListAdapter(IntRange(0,100).toList())
}
```

Make items in the list clickable

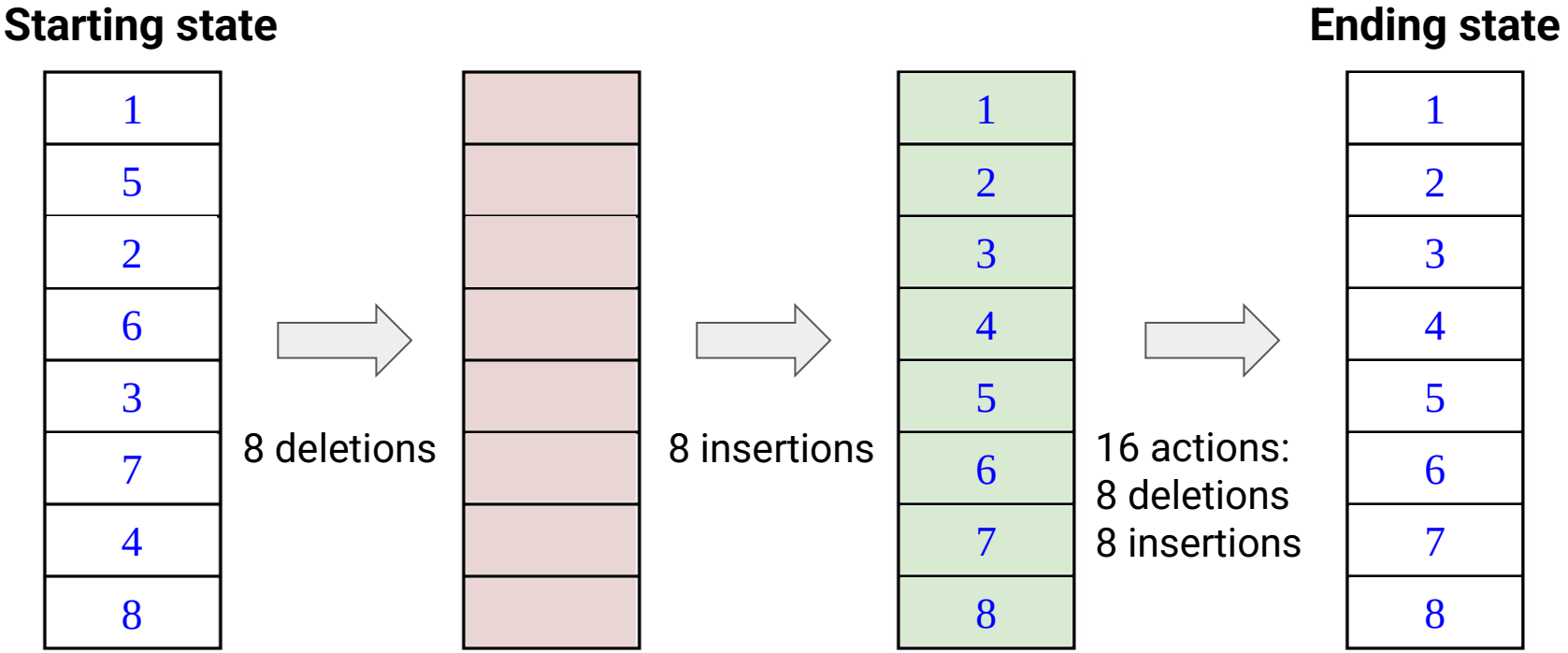
In `NumberListAdapter.kt`:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): IntViewHolder{
    val layout = LayoutInflater.from(parent.context).inflate(R.layout.item_view,
         parent, false)
    val holder = IntViewHolder(layout)
    holder.row.setOnClickListener {
        // Do something on click
    }
    return holder
}
```

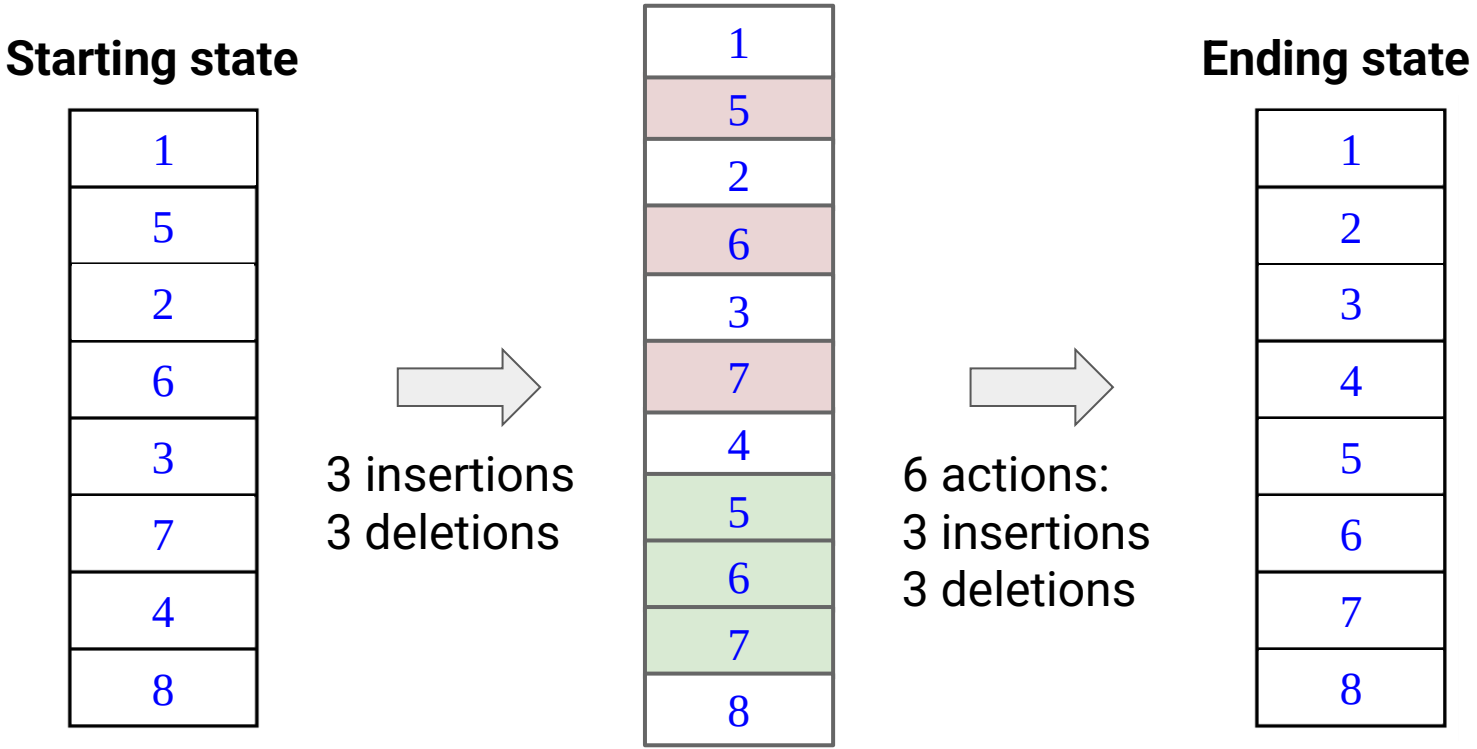
ListAdapter

- `RecyclerView.Adapter`
  - Disposes UI data on every update
  - Can be costly and wasteful
- `ListAdapter`
  - Computes the difference between what is currently shown and what needs to be shown
  - Changes are calculated on a background thread

Sort using RecyclerView.Adapter



Sort using ListAdapter



ListAdapter example

```
class NumberListAdapter: ListAdapter<Int,
        NumberListAdapter.IntViewHolder>(RowItemDiffCallback()) {

    class IntViewHolder(val row: View):RecyclerView.ViewHolder(row) {
        val textView = row.findViewById<TextView>(R.id.number)
    }

    ...
```

DiffUtil.ItemCallback

Determines the transformations needed to translate one list into another

- `areContentsTheSame(oldItem: T, newItem: T): Boolean`
- `areItemsTheSame(oldItem: T, newItem: T): Boolean`

DiffUtil.ItemCallback example

```
class RowItemDiffCallback : DiffUtil.ItemCallback<Int>() {

    override fun areItemsTheSame(oldItem: Int, newItem: Int): Boolean {
        return oldItem == newItem
    }

    override fun areContentsTheSame(oldItem: Int, newItem: Int): Boolean {
        return oldItem == newItem
    }
}
```

Advanced binding

ViewHolders and data binding

```
class IntViewHolder private constructor(val binding: ItemViewBinding):
        RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun from(parent: ViewGroup): IntViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val binding = ItemViewBinding.inflate(layoutInflater,
                parent, false)
            return IntViewHolder(binding)
        }
    }
}
```

Using the ViewHolder in a ListAdapter

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        IntViewHolder {
    return IntViewHolder.from(parent)
}

override fun onBindViewHolder(holder: NumberListAdapter.IntViewHolder,
        position: Int) {
    holder.binding.num = getItem(position)
}
```

Binding adapters

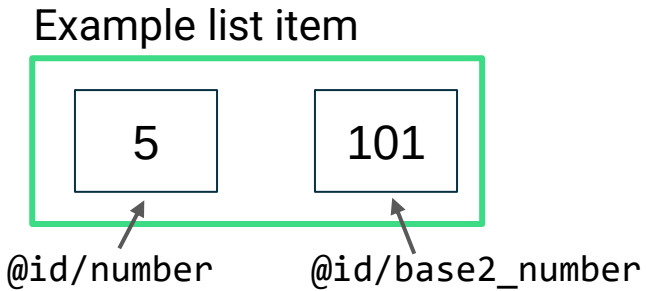
Let you map a function to an attribute in your XML

- Override existing framework behavior:
  `android:text = "foo"` → `TextView.setText("foo")` is called
- Create your own custom attributes:
  `app:base2Number = "5"` → `TextView.setBase2Number("5")` is called

Custom attribute

Add another `TextView` in the list item layout that uses a custom attribute:

```
<TextView
    android:id="@+id/base2_number"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="24sp"
    app:base2Number="@{num}"/>
```



Add a binding adapter

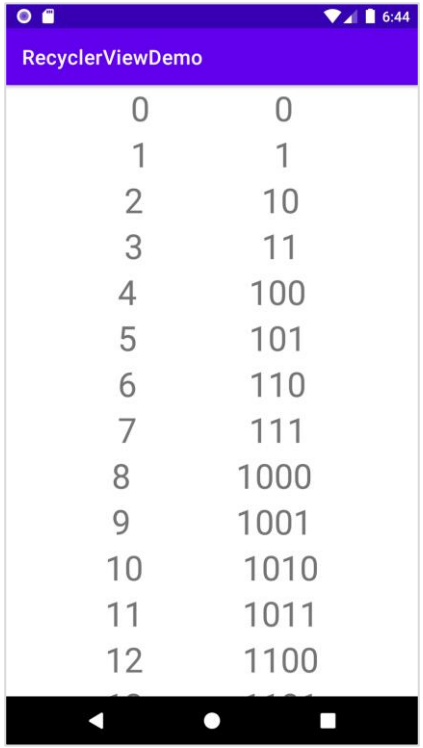
Declare binding adapter:

```
@BindingAdapter("base2Number")
fun TextView.setBase2Number(item: Int) {
    text = Integer.toBinaryString(item)
}
```

In `NumberListAdapter.kt`:

```
override fun onBindViewHolder(holder: NumberListAdapter.IntViewHolder,
        position: Int) {
    holder.binding.num = getItem(position)
    holder.binding.executePendingBindings()
}
```

Updated RecyclerViewDemo app



Multiple item view types

Add a new item view type

1. Create a new list item layout XML file.
2. Modify underlying adapter to hold the new type.
3. Override `getItemViewType` in adapter.
4. Create a new `ViewHolder` class.
5. Add conditional code in `onCreateViewHolder` and `onBindViewHolder` to handle the new type.

Declare new color item layout

```
<layout ...>
    <data>
        <variable
            name="color"
            type="android.graphics.Color" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout ...>
        <TextView
            ...
            android:backgroundColor="@{color.toArgb()}" />
        <TextView
            ...
            android:text="@{color.toString()}" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

New view type

- Adapter should know about two item view types:
  - Item that displays a number
  - Item that displays a color

```
enum class ITEM_VIEW_TYPE { NUMBER, COLOR }
```

- Modify `getItemViewType()` to return the appropriate type (as `Int`):

```
override fun getItemViewType(position: Int): Int
```

Override getItemViewType

In `NumberListAdapter.kt`:

```
override fun getItemViewType(position: Int): Int {
    return when(getItem(position)) {
        is Int -> ITEM_VIEW_TYPE.NUMBER.ordinal
        else -> ITEM_VIEW_TYPE.COLOR.ordinal
    }
}
```

Define new ViewHolder

```
class ColorViewHolder private constructor(val binding: ColorItemViewBinding):
      RecyclerView.ViewHolder(binding.root) {

    companion object {
        fun from(parent: ViewGroup): ColorViewHolder {
            val layoutInflater = LayoutInflater.from(parent.context)
            val binding = ColorItemViewBinding.inflate(layoutInflater,
                parent, false)
            return ColorViewHolder(binding)
        }
    }
}
```

Update onCreateViewHolder()

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        RecyclerView.ViewHolder {

    return when(viewType) {
        ITEM_VIEW_TYPE.NUMBER.ordinal -> IntViewHolder.from(parent)
        else -> ColorViewHolder.from(parent)
    }
}
```

Update onBindViewHolder()

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
    when (holder) {
        is IntViewHolder -> {
            holder.binding.num = getItem(position) as Int
            holder.binding.executePendingBindings()
        }
        is ColorViewHolder -> {
            holder.binding.color = getItem(position) as Color
            holder.binding.executePendingBindings()
        }
    }
}
```

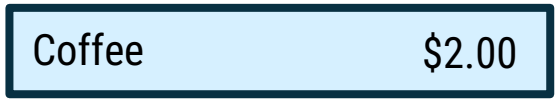
Headers

Headers Example

- 2 item view types:
  - header item

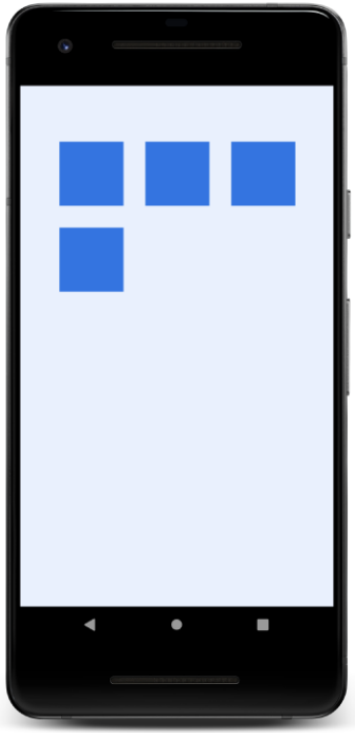
    

  - food menu item

Grid layout

List versus grid



Specifying a LayoutManager

In `MainActivity onCreate()`, once you have a reference to the `RecyclerView`

- Display a list with `LinearLayoutManager`:

```
recyclerView.layoutManager = LinearLayoutManager(this)
```

- Display a grid with `GridLayoutManager`:

```
recyclerView.layoutManager = GridLayoutManager(this, 2)
```

- Use a different layout manager (or create your own)

GridLayoutManager

- Arranges items in a grid as a table of rows and columns.
- Orientation can be vertically or horizontally scrollable.
- By default, each item occupies 1 span.
- You can vary the number of spans an item takes up (span size).

Set span size for an item

Create `SpanSizeLookup` instance and override `getSpanSize(position)`:

```
val manager = GridLayoutManager(this, 2)
manager.spanSizeLookup = object : GridLayoutManager.SpanSizeLookup() {
    override fun getSpanSize(position: Int): Int {
        return when (position) {
            0,1,2 -> 2
            else -> 1
        }
    }
}
```

Summary

Summary

In Lesson 10, you learned how to:

- Use `ListAdapter` to make `RecyclerView` more efficient at updating lists
- Create a binding adapter with custom logic to set View values from an XML attribute
- Handle multiple `ViewHolders` in the same `RecyclerView` to show multiple item types
- Use `GridLayoutManager` to display items as a grid
- Specify span size for an item in a grid with `SpanSizeLookup`

Learn More

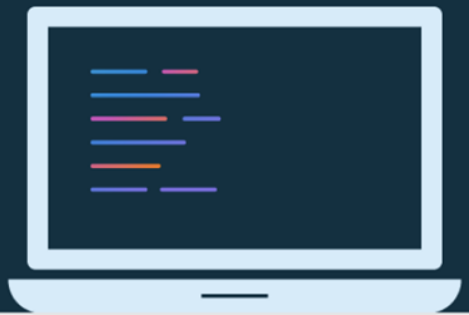
- Create a List with RecyclerView
- RecyclerView
- ListAdapter
- Binding adapters
- GridLayoutManager
- DiffUtil and ItemCallback

Pathway

Practice what you've learned by completing the pathway:

Lesson 10: Advanced RecyclerView use cases

Lesson 11: Connect to the internet



About this lesson

Lesson 11: Connect to the internet

- Android permissions
- Connect to, and use, network resources
- Connect to a web service
- Display images
- Summary

Android permissions

Permissions

- Protect the privacy of an Android user
- Declared with the `<uses-permission>` tag in the `AndroidManifest.xml`

Permissions granted to your app

- Permissions can be granted during installation or runtime, depending on protection level.
- Each permission has a protection level: normal, signature, or dangerous.
- For permissions granted during runtime, prompt users to explicitly grant or deny access to your app.

Permission protection levels

Protection Level	Granted when?	Must prompt before use?	Examples
Normal	Install time	No	`ACCESS_WIFI_STATE`, `BLUETOOTH`, `VIBRATE`, `INTERNET`
Signature	Install time	No	N/A
Dangerous	Runtime	Yes	`GET_ACCOUNTS`, `CAMERA`, `CALL_PHONE`

Add permissions to the manifest

In `AndroidManifest.xml`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.sampleapp">

    <uses-permission android:name="android.permission.USE_BIOMETRIC" />

    <application>
        <activity
            android:name=".MainActivity" ... >
            ...
        </activity>
    </application>
</manifest>
```

Internet access permissions

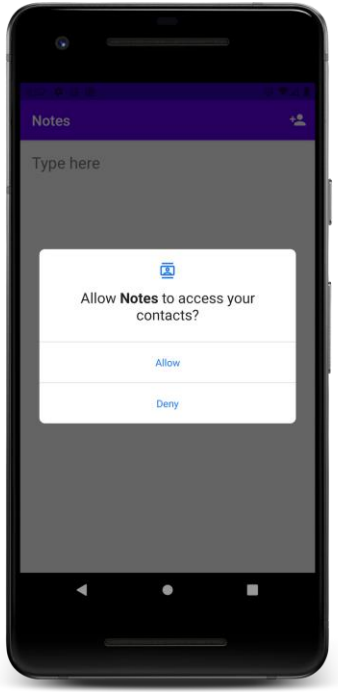
In `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Request dangerous permissions

- Prompt the user to grant the permission when they try to access functionality that requires a dangerous permission.
- Explain to the user why the permission is needed.
- Fall back gracefully if the user denies the permission (app should still function).

Prompt for dangerous permission



App permissions best practices

- Only use the permissions necessary for your app to work.
- Pay attention to permissions required by libraries.
- Be transparent.
- Make system accesses explicit.

Connect to, and use, network resources

Retrofit

- Networking library that turns your HTTP API into a Kotlin and Java interface
- Enables processing of requests and responses into objects for use by your apps
  - Provides base support for parsing common response types, such as XML and JSON
  - Can be extended to support other response types

Why use Retrofit?

- Builds on industry standard libraries, like OkHttp, that provide:
  - HTTP/2 support
  - Connection pooling
  - Response caching and enhanced security
- Frees the developer from the scaffolding setup needed to run a request

Add Gradle dependencies

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-moshi:2.9.0"

implementation "com.squareup.moshi:moshi:$moshi_version"
implementation "com.squareup.moshi:moshi-kotlin:$moshi_version"
kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
```

Connect to a web service

HTTP methods

- GET
- POST
- PUT
- DELETE

Example web service API

URL	DESCRIPTION	METHOD
`example.com/posts`	Get a list of all posts	`GET`
`example.com/posts/username`	Get a list of posts by user	`GET`
`example.com/posts/search?filter=queryterm`	Search posts using a filter	`GET`
`example.com/posts/new`	Create a new post	`POST`

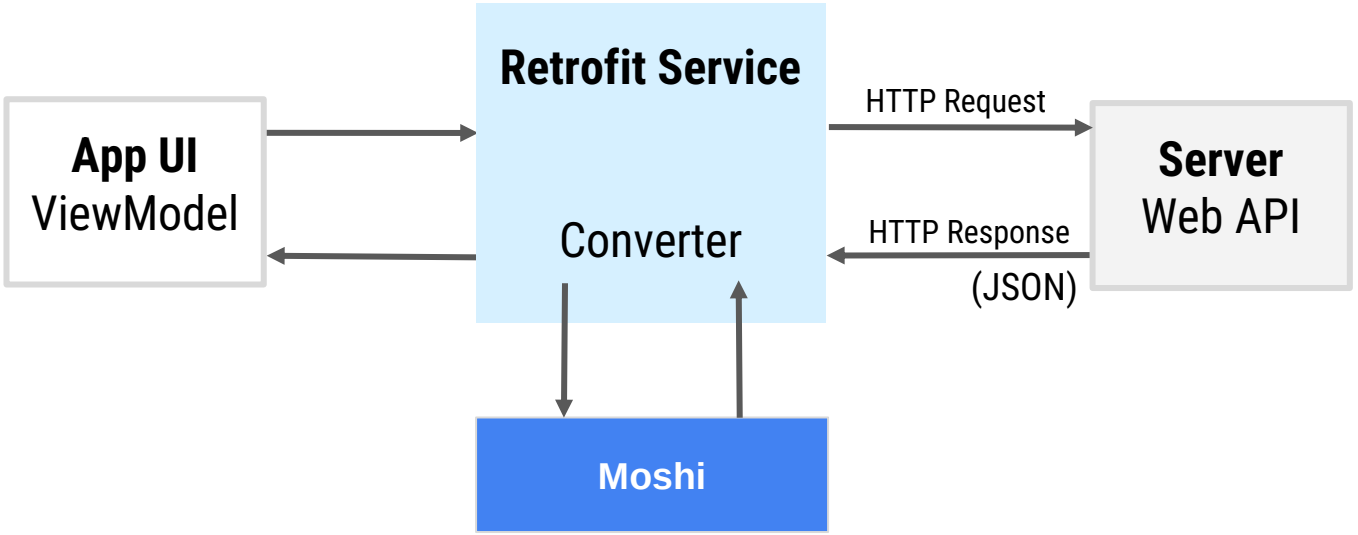
Define a Retrofit service

```
interface SimpleService {

    @GET("posts")
    suspend fun listPosts(): List<Post>

    @GET("posts/{userId}")
    suspend fun listByUser(@Path("userId") userId:String): List<Post>

    @GET("posts/search")  // becomes post/search?filter=query
    suspend fun search(@Query("filter") search: String): List<Post>

    @POST("posts/new")
    suspend fun create(@Body post : Post): Post
}
```

Create a Retrofit object for network access

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://example.com")
    .addConverterFactory(...)
    .build()

val service = retrofit.create(SimpleService::class.java)
```

End-to-end diagram



Converter.Factory

Helps convert from a response type into class objects

- JSON (Gson or Moshi)
- XML (Jackson, SimpleXML, JAXB)
- Protocol buffers
- Scalars (primitives, boxed, and Strings)

Moshi

- JSON library for parsing JSON into objects and back
- Add Moshi library dependencies to your app's Gradle file.
- Configure your Moshi builder to use with Retrofit.

Moshi JSON encoding

```
@JsonClass(generateAdapter = true)
data class Post (
    val title: String,
    val description: String,
    val url: String,
    val updated: String,
    val thumbnail: String,
    val closedCaptions: String?)
```

JSON code


```
{
    "title":"Android Jetpack: EmojiCompat",
    "description":"Android Jetpack: EmojiCompat",
    "url":"https://www.youtube.com/watch?v=sYGKUtM2ga8",
    "updated":"2018-06-07T17:09:43+00:00",
    "thumbnail":"https://i4.ytimg.com/vi/sYGKUtM2ga8/hqdefault.jpg"
}
```

Set up Retrofit and Moshi

```
private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()

val retrofit = Retrofit.Builder()
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .baseUrl(BASE_URL)
        .build()

object API {
    val retrofitService : SimpleService by lazy {
        retrofit.create(SimpleService::class.java)
    }
}
```

Use Retrofit with coroutines

Launch a new coroutine in the view model:

```
viewModelScope.launch {
    Log.d("posts", API.retrofitService.searchPosts("query"))
}
```

Display images

Glide

- Third-party image-loading library in Android
- Focused on performance for smoother scrolling
- Supports images, video stills, and animated GIFs

Add Gradle dependency

```
implementation "com.github.bumptech.glide:glide:$glide_version"
```

Load an image

```
Glide.with(fragment)
     .load(url)
     .into(imageView);
```

Customize a request with RequestOptions

- Apply a crop to an image
- Apply transitions
- Set options for placeholder image or error image
- Set caching policies

RequestOptions example

```
@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {

    imgUrl?.let {
        val imgUri = imgUrl.toUri().buildUpon().scheme("https").build()

        Glide.with(imgView)
            .load(imgUri)
            .apply(RequestOptions()
                .placeholder(R.drawable.loading_animation)
                .error(R.drawable.ic_broken_image))
            .into(imgView)
    }
}
```

Summary

Summary

In Lesson 11, you learned how to:

- Declare permissions your app needs in `AndroidManifest.xml`
- Use the three protection levels for permissions: normal, signature, and dangerous (prompt the user at runtime for dangerous permissions)
- Use the Retrofit library to make web service API calls from your app
- Use the Moshi library to parse JSON response into class objects
- Load and display images from the internet using the `Glide` library

Learn More

- App permissions best practices
- Retrofit
- Moshi
- Glide

Pathway

Practice what you've learned by completing the pathway:

Lesson 11: Connect to the internet