Mobile Programming

More UIs

Agenda

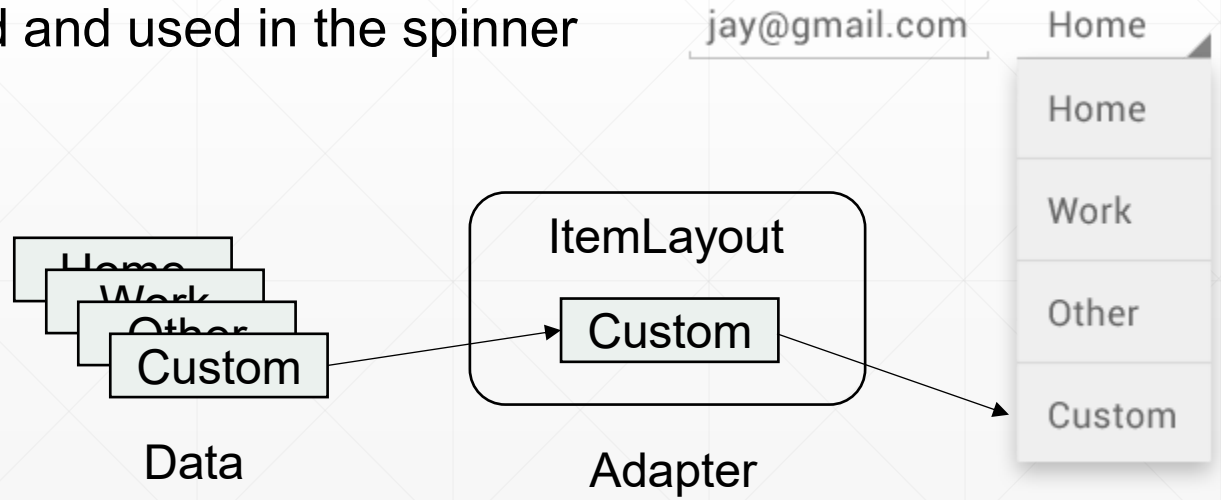
- Container
  - Spinner
  - Recyclerview
- Popup
  - Dialog
  - Snackbar & Toast

Container

- UI component to display data on the widget/layout dynamically
- Used to repetitively display data
  - RecyclerView
  - ListView (legacy)
  - GridView (legacy)
  - Spinner
  - …

Spinner (1/6)

- UI component which provides a quick way to select one value from a set
  - In the default state, a spinner shows its currently selected value
  - Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one.
- Adapter
  - Connects data to be displayed and a spinner UI
  - One layout for a single data is generated and used in the spinner
    - 1 line in the spinner == 1 layout for an item

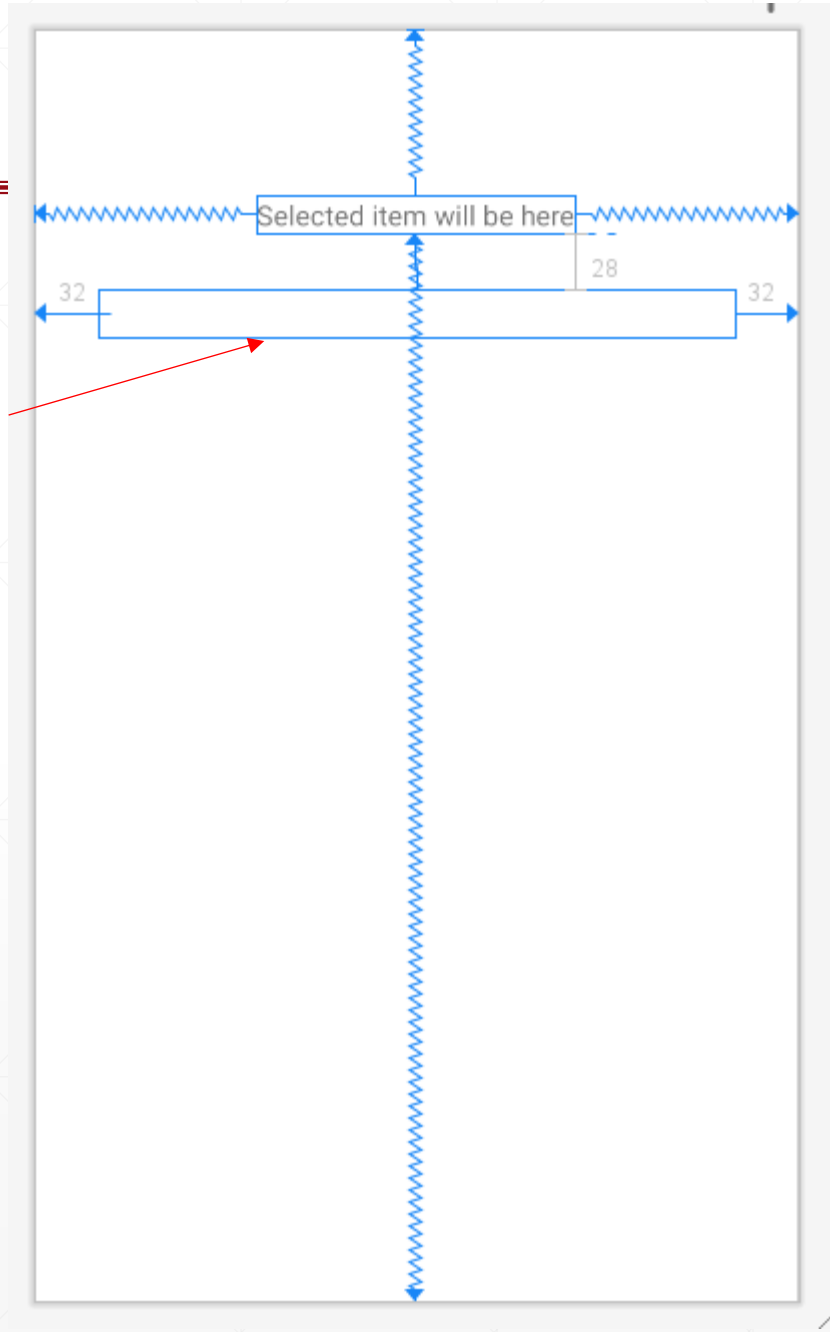


Spinner (2/6)

- TextView
  - Text: “Selected item will be here"
- Spinner
  - width: match constraint
  - topConstraint → bottom of textView
- Dataset

```
val myList = listOf("C","C++","C#","Java","Javascript","Kotlin","GoLang")
```

- ArrayAdapter
  - Provides views for an AdapterView
  - Returns a view for each object in a collection of data objects



Spinner (3/6)

- ArrayAdapter
  - Provides views for an AdapterView
  - Returns a view for each object in a collection of data objects

public ArrayAdapter (**Context** context, int resource, T[] objects)

- Context: the current context
- Resource: the resource ID for a layout file containing a TextView to use when instantiating views
- Objects: the data to represent

```
val myAdapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, myList)
```

Pre-built layout provided by Android!

- Connect your adapter to the Spinner

```
binding.spinner.adapter = myAdapter
```

Note! Here, binding variable is declared through a property initializer like
val binding by *lazy* { ActivityMainBinding.inflate(*layoutInflater*) } **(e.g., find Lecture Video 8:17~20)**

Spinner (4/6)

- Responding to user selections
  - When the user selects an item from the drop-down, the Spinner object receives an on-item-selected event
  - To define the selection event handler for a spinner, we need to implement the AdapterView.OnItemSelectedListener interface
    - Interface definition for a callback to be invoked when an item in this view has been selected
    - Requires the onItemSelected() and onNothingSelected() callback methods

abstract void	onItemSelected(AdapterView<?> parent, View view, int position, long id) Callback method to be invoked when an item in this view has been selected.
abstract void	onNothingSelected(AdapterView<?> parent) Callback method to be invoked when the selection disappears from this view.

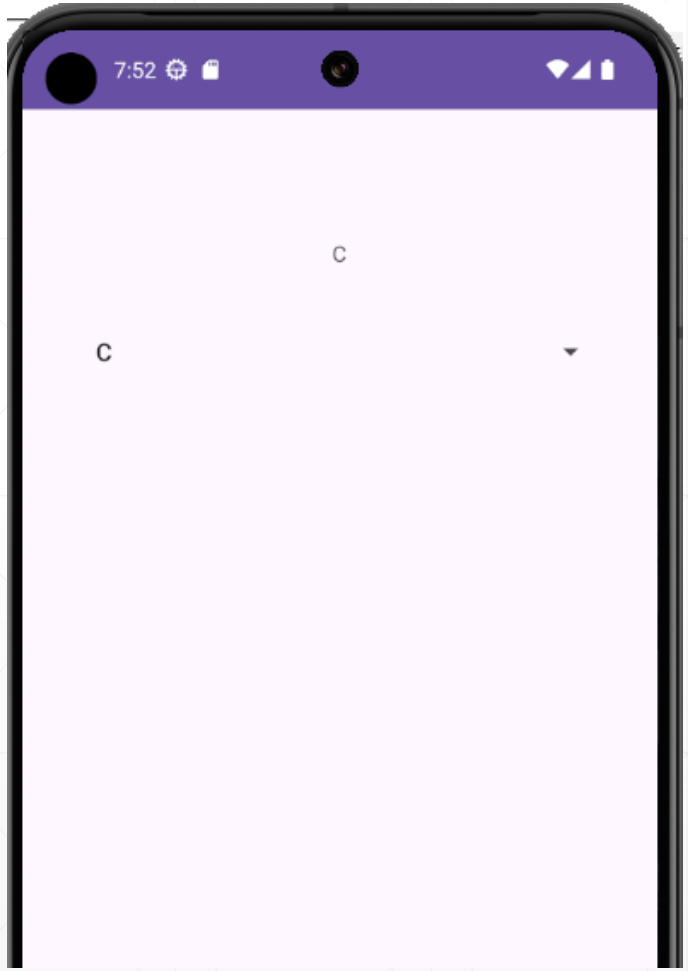
Spinner (5/6)

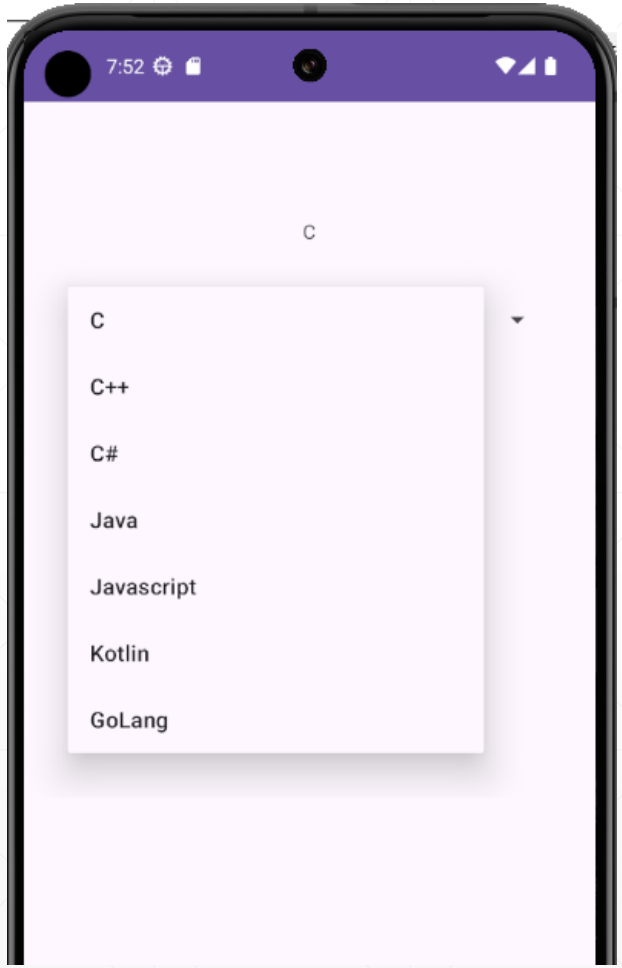
- Responding to user selections
  - To define the selection event handler for a spinner, we need to implement the AdapterView.OnItemSelectedListener interface
    - Interface definition for a callback to be invoked when an item in this view has been selected
    - Requires the onItemSelected() and onNothingSelected() callback methods

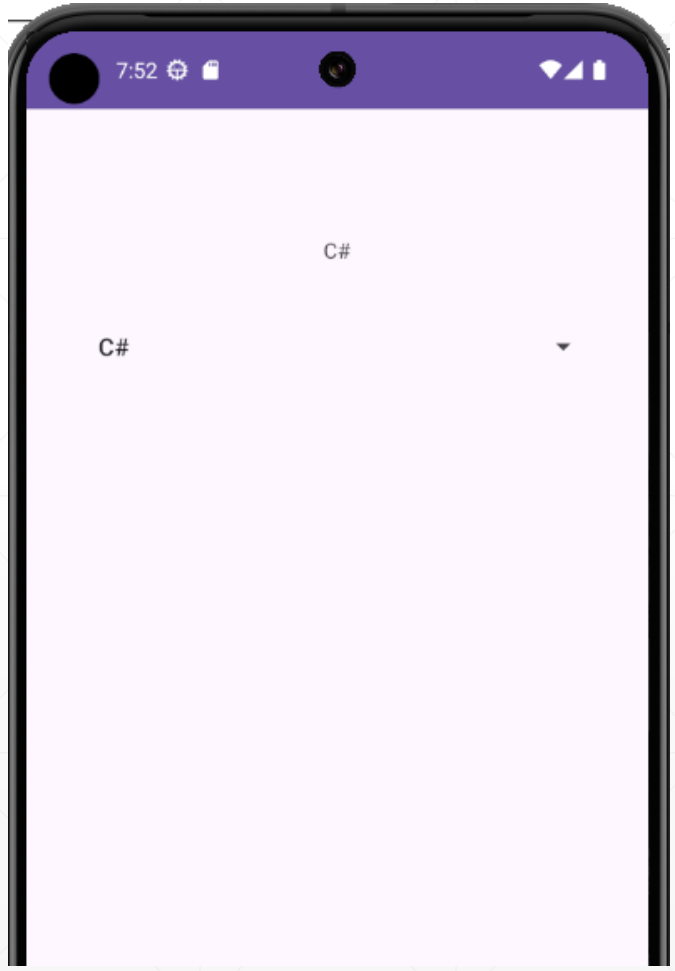
```
binding.spinner.onItemSelectedListener = object: AdapterView.OnItemSelectedListener{
    override fun onItemSelected(p0: AdapterView<*>?, p1: View?, p2: Int, p3: Long) {
        binding.tView.text = myList.get(p2)
    }
    override fun onNothingSelected(p0: AdapterView<*>?) { }
}
```

Spinner (6/6)

- Running example!







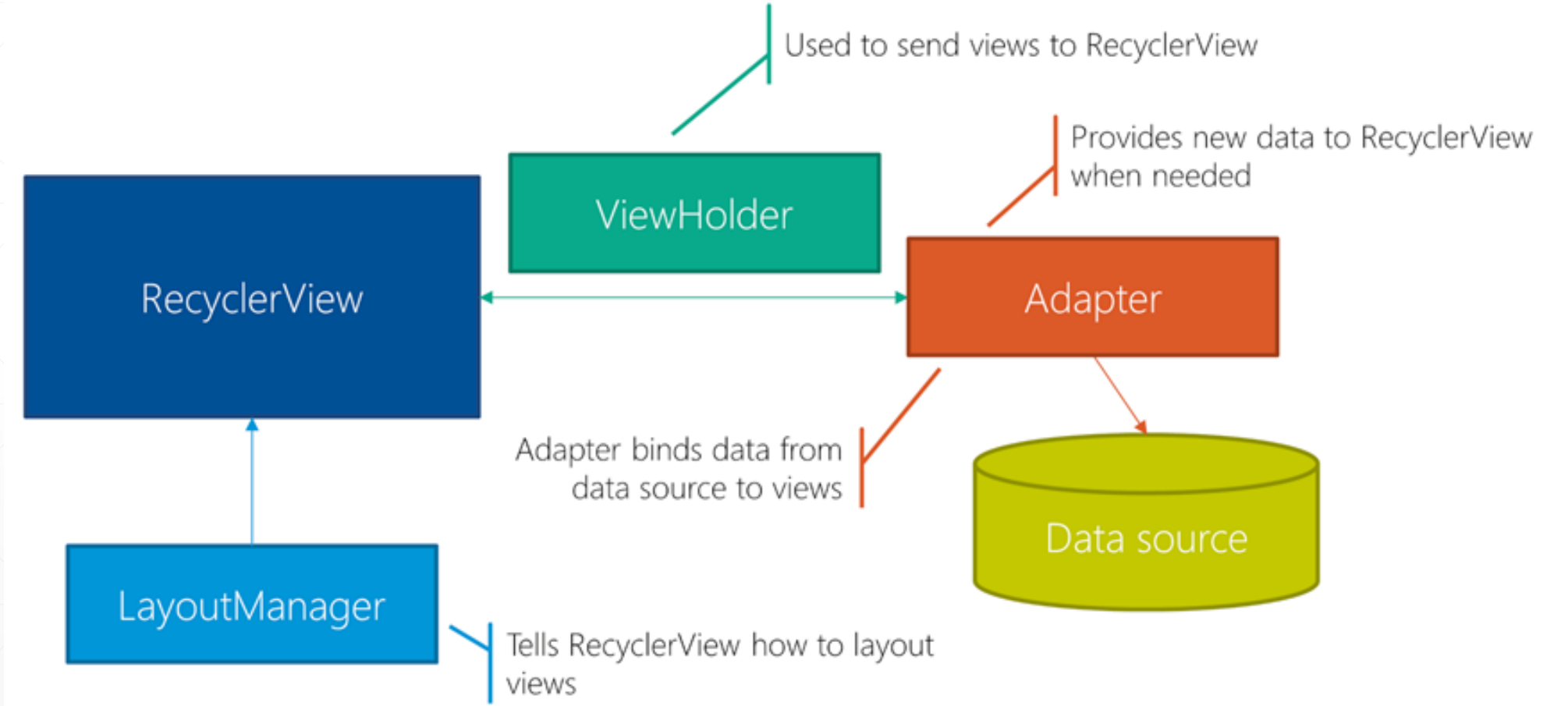
RecyclerView (1/13)

- RecyclerView makes it easy to efficiently display large sets of data
- Supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they are needed!

- Why “recycler” view?
  - RecyclerView *recycles* those individual elements
  - When an item scrolls off the screen, RecyclerView does not destroy its view
  - Instead, RecyclerView ***reuses*** the view for new items that have scrolled onscreen!
  - This reuse vastly improves performance, improving your app's responsiveness and reducing power consumption

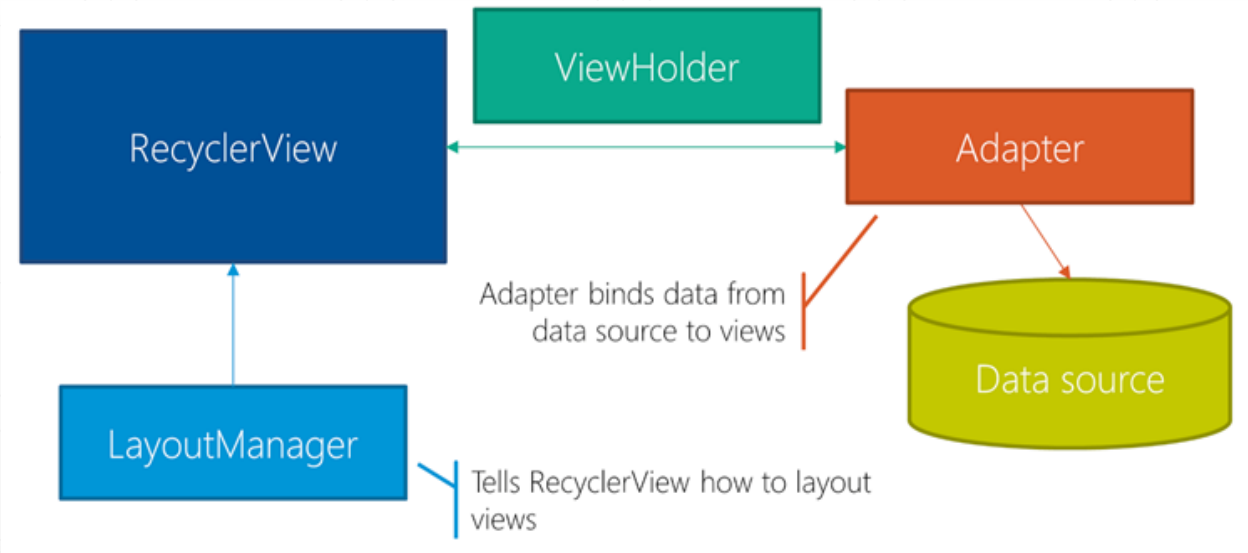
RecyclerView (2/13)

- RecyclerView



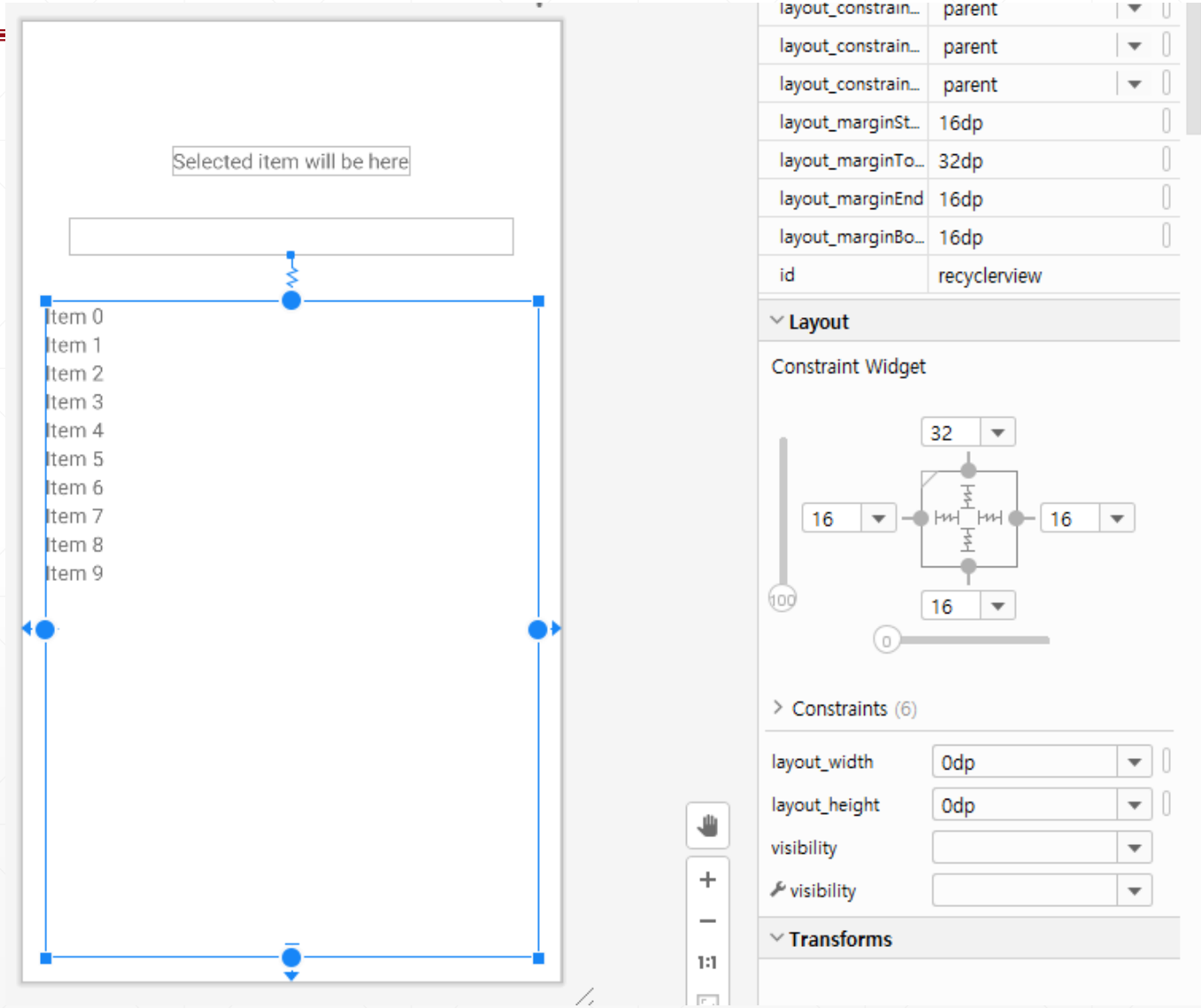
RecyclerView (3/13)

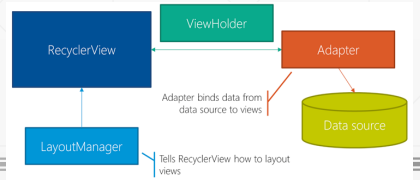
- Key classses
  - RecyclerView
    - ViewGroup that contains the views corresponding to your data
    - It's a view itself, so you add RecyclerView into your layout
  - ViewHolder
    - Each individual element in the list is defined by a view holder object
  - Adapter
    - Binds the views to their data
  - LayoutManager
    - Arranges the individual elements in your list



RecyclerView (4/13)

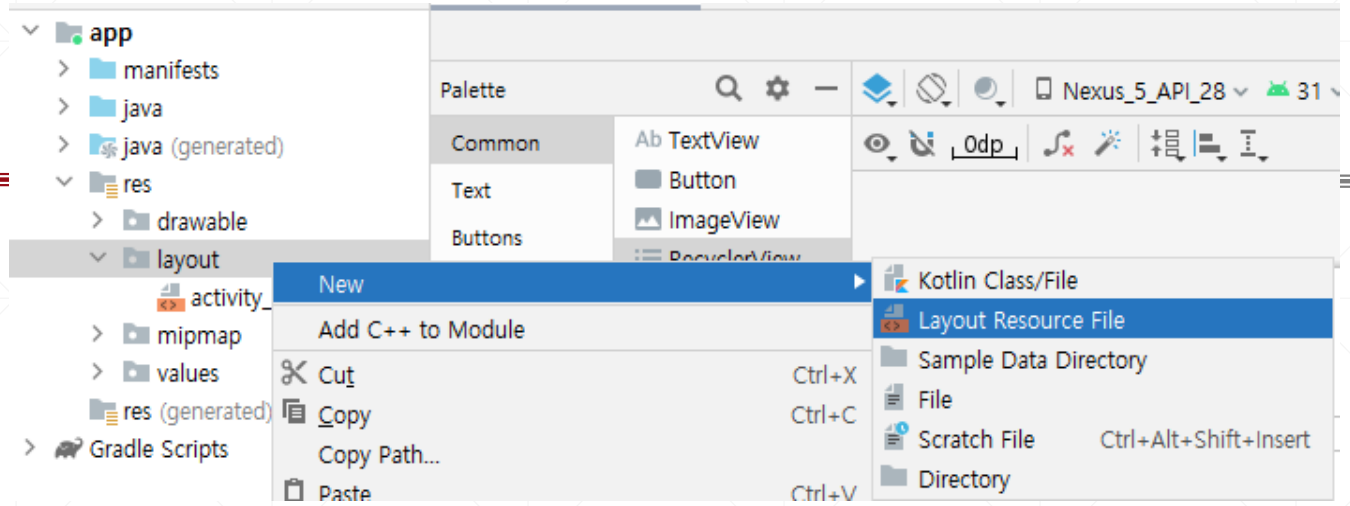
- Add RecyclerView into your activity
  - Set contstraints
  - Set margins
  - Set ID

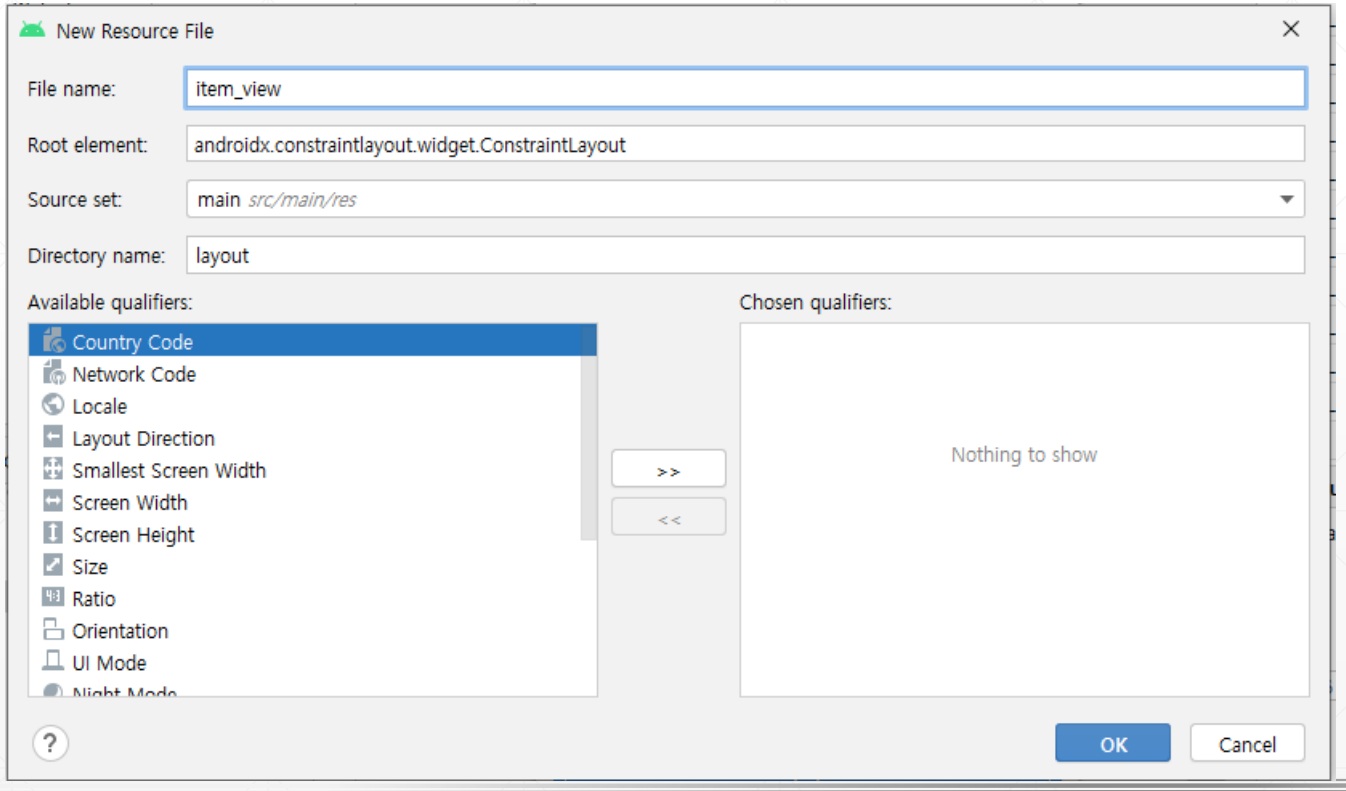


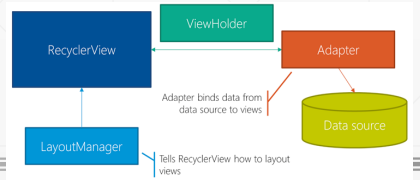


RecyclerView (5/13)

- Layout for individual items
  - Layout→new→layout resource file
  - Set your filename

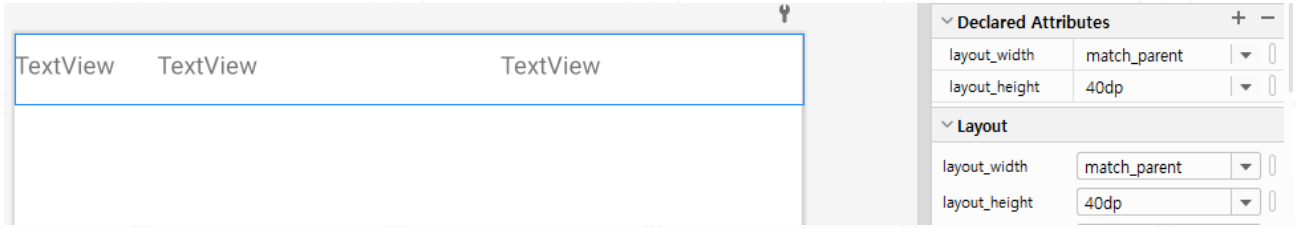




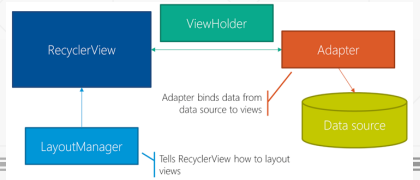


RecyclerView (6/13)

- Layout for individual items
  - Layout→new→layout resource file
  - Set your filename
  - Set your layout for an individual item

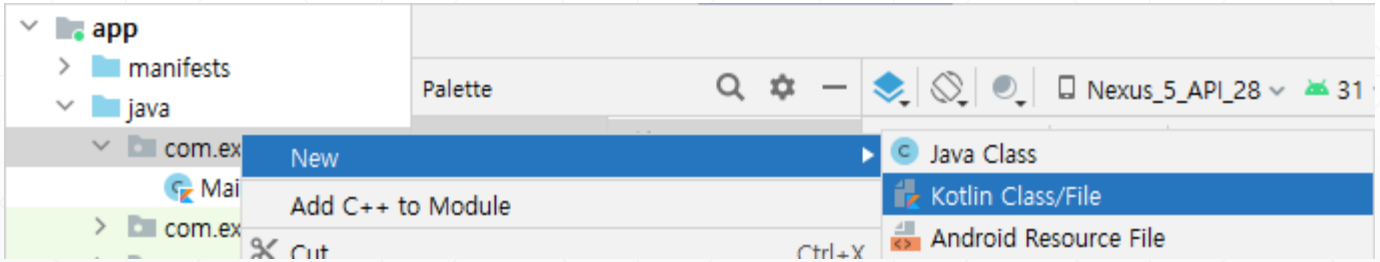


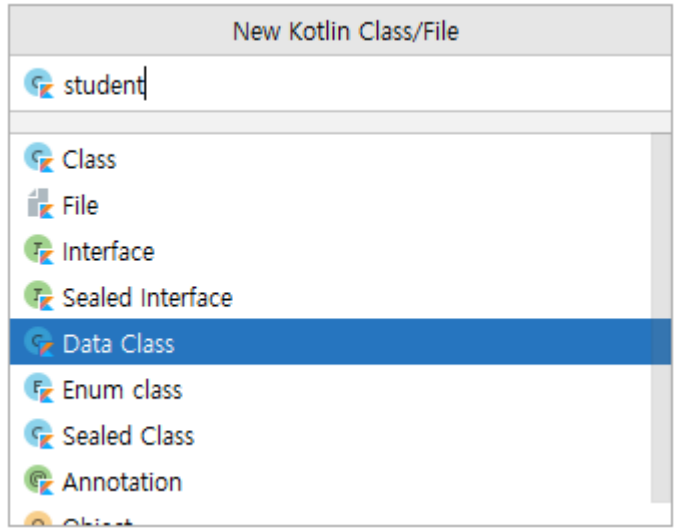

```
<TextView
    android:id="@+id/id"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:text="TextView"
    app:layout_constraintEnd_toStartOf="@+id/name"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintHorizontal_chainStyle="spread_inside"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/name"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="24dp"
    android:layout_marginEnd="24dp"
    android:text="TextView"
    app:layout_constraintBaseline_toBaselineOf="@+id/id"
    app:layout_constraintEnd_toStartOf="@+id/grade"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/id" />

<TextView
    android:id="@+id/grade"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="TextView"
    app:layout_constraintBaseline_toBaselineOf="@+id/name"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/name" />
```

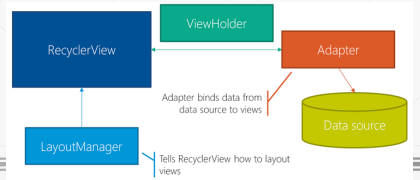


RecyclerView (7/13)

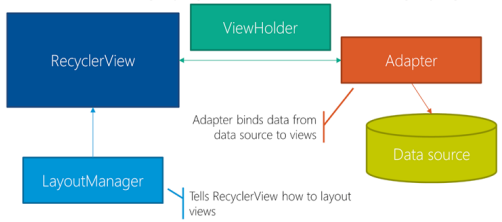
- Data
  - We will define a class to store the following data
    - id: int
    - name: String
    - grade: String
  - Package name →new → kotlin class







RecyclerView (8/13)



- Data
  - Data class definition

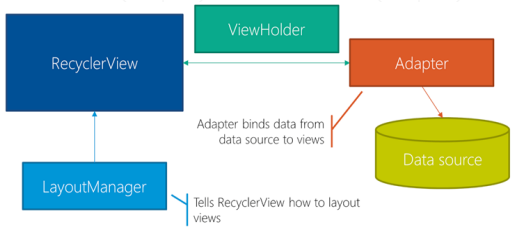
```
package com.example.uiapp

data class Student(var id:Int, var name:String, var grade:String)
```

  - Data preparation
    - Example data

```
val myStudents = mutableListOf<Student>()
for (no in 0 .. 100){
    val item = Student(no, "student"+no, "A+")
    myStudents.add(item)
}
```

    - Any data can be used!

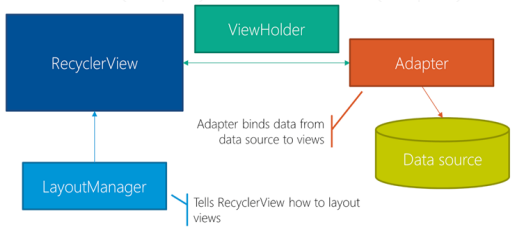
RecyclerView (9/13)



- Adapter & ViewHolder
  - These two classes work together to define how your data is displayed
  - ViewHolder is a wrapper around a View that contains the layout for an individual item
  - Adapter creates ViewHolder objects as needed, and also sets the data for those views

- Adapter structure
  - onCreateViewHolder()
    - RecyclerView calls this method whenever it needs to create a new ViewHolder
  - onBindViewHolder()
    - RecyclerView calls this method to associate a ViewHolder with data
  - getItemCount()
    - RecyclerView calls this method to get the size of the data set

RecyclerView (10/13)



Adapter & ViewHolder

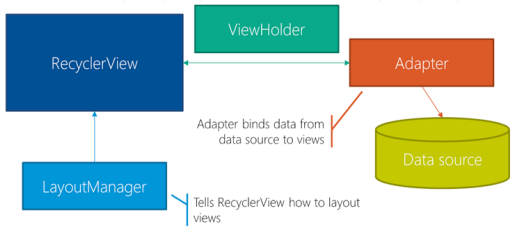
```
...

class StudentAdapter: RecyclerView.Adapter<StudentAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        TODO("Not yet implemented")
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        TODO("Not yet implemented")
    }

    override fun getItemCount(): Int {
        TODO("Not yet implemented")
    }

    class ViewHolder(val binding:ItemViewBinding): RecyclerView.ViewHolder(binding.root) {

    }
}
```

Called to create a new ViewHolder

Associates data with a ViewHolder

Individual layout

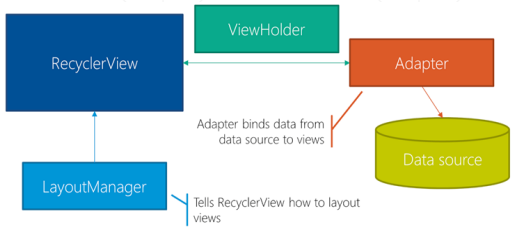
RecyclerView (11/13)



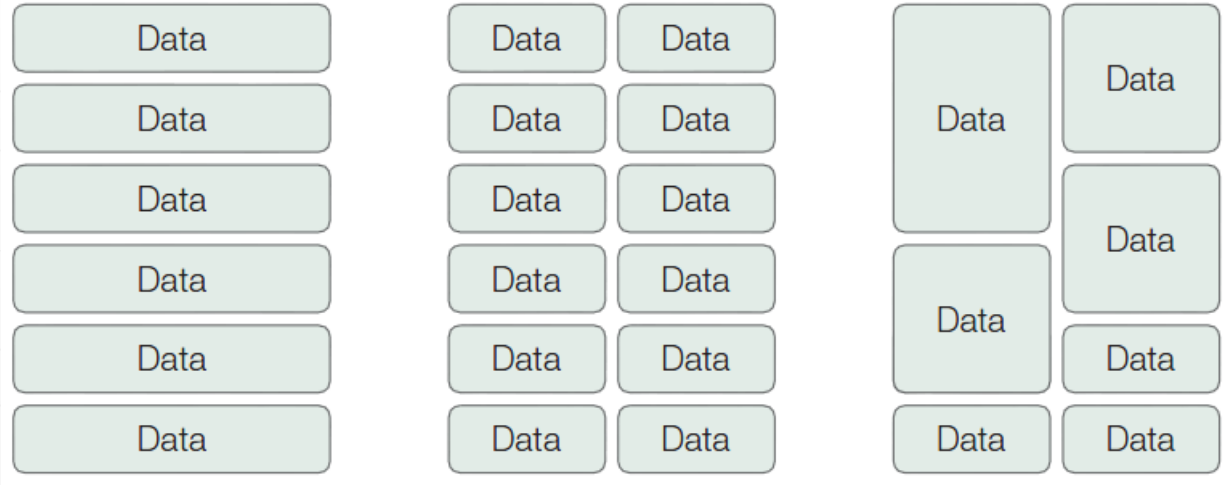
- Adapter & ViewHolder

```
class StudentAdapter(val myStudents:MutableList<Student>): RecyclerView.Adapter<StudentAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val binding = ItemViewBinding.inflate(LayoutInflater.from(parent.context), parent, false)
        return ViewHolder(binding)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val student = myStudents.get(position)
        holder.bind(student)
    }

    override fun getItemCount(): Int {
        return myStudents.size
    }

    class ViewHolder(val binding:ItemViewBinding): RecyclerView.ViewHolder(binding.root) {
        fun bind(student:Student){
            binding.grade.text=student.grade
            binding.name.text=student.name
            binding.id.text="${student.id}"
        }
    }
}
```

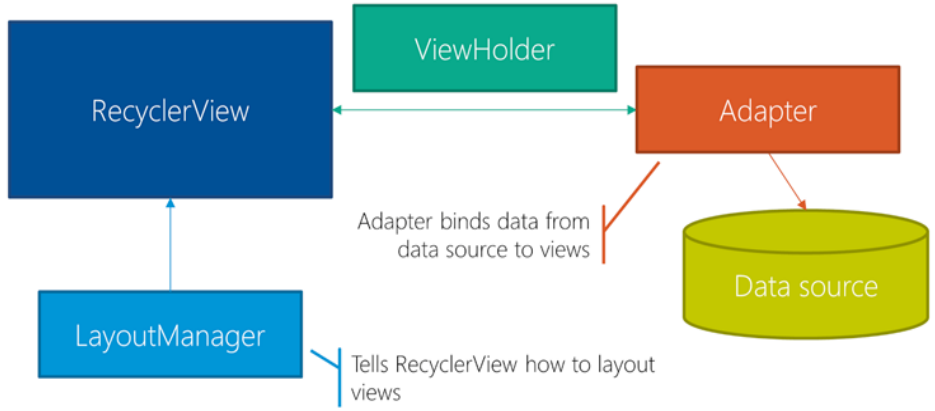
RecyclerView (12/13)



- LayoutManager
  - The items in your RecyclerView are arranged by a LayoutManager class
  - RecyclerView library provides three layout managers, which handle the most common layout situations
    - *LinearLayoutManager* arranges the items in a one-dimensional list
    - *GridLayoutManager* arranges all items in a two-dimensional grid
    - *StaggeredGridLayoutManager*



RecyclerView (13/13)



- Setup Adapter & LayoutManager

```
…

val stuAdapter = StudentAdapter(myStudents)
binding.recyclerview.adapter = stuAdapter
binding.recyclerview.layoutManager = LinearLayoutManager(this)
```

  - You can also use GridLayoutManager or something else!

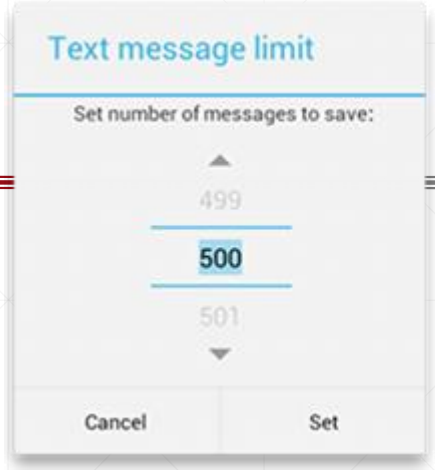
- Click event for each item
  - Set onClickListener() for a ViewHolder

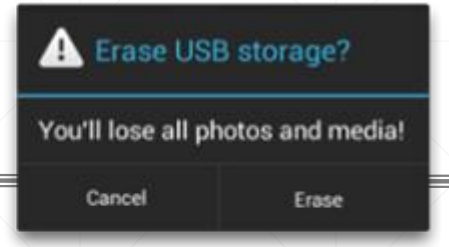
```
class ViewHolder(val binding: ItemViewBinding) : RecyclerView.ViewHolder(binding.root) {

    init {
        binding.root.setOnClickListener {
            Toast.makeText(binding.root.context,"Success! ${binding.name.text}'s grade is ${binding.grade.text}!!!",Toast.LENGTH_SHORT).show()
        }
    }
…
```

Popup

- UI components to interact with a user
  - Show messages
  - Take simple responses
  - Perform simple actions
  - …
- Example
  - AlertDialog
  - Toast/SnackBar
  - …

Dialog





- Dialog
  - Small window that prompts the user to make a decision or enter additional information
  - Does not fill the screen and is normally used for modal events that require users to take an action before they can proceed
- AlertDialog
  - A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout
- DatePickerDialog/TimePickerDialog
  - A dialog with a pre-defined UI that allows the user to select a date or time

AlertDialog (1/5)

- AlertDialog
  - Title
    - Optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout
  - Content area
    - This can display a message, a list, or other custom layout
  - Action buttons
    - There should be no more than three action buttons in a dialog

- AlertDialog.Builder class provides APIs that allow you to create an AlertDialog!
  - https://developer.android.com/reference/kotlin/androidx/appcompat/app/AlertDialog.Builder

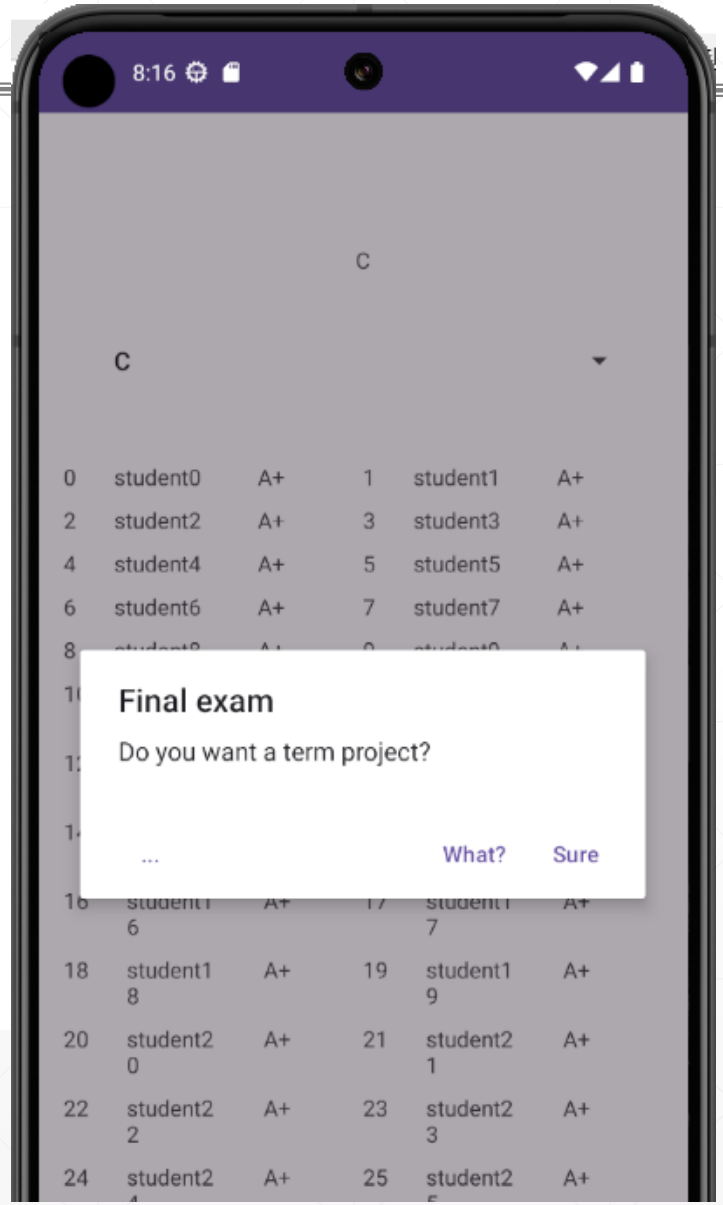
AlertDialog (2/5)

- AlertDialog.builder(context)
  - SetMessage(): Set the message to display
  - SetTitle(): Set the title displayed in the Dialog
  - Buttons
    - Require a title for the and a DialogInterface.OnClickListener that defines the action to take when the user presses the button
    - Positive button: use this to accept and continue with the action (the "OK" action)
      - setPositiveButton()
    - Negative button: use this to cancel the action
      - setNegativeButton()
    - Neutral button: use this when the user may not want to proceed with the action, but does not necessarily want to cancel
      - setNeutralButton()
  - Show(): Creates an AlertDialog with the arguments and immediately displays the dialog

AlertDialog (3/5)

- Example)

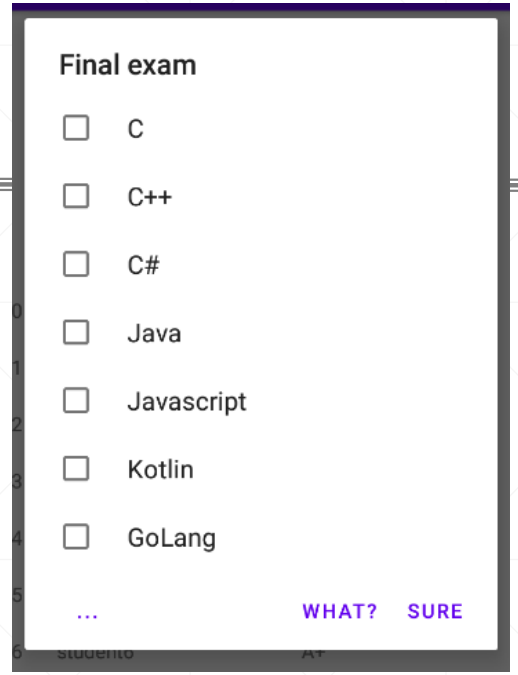
```
binding.tView.setOnClickListener {
    AlertDialog.Builder(this)
        .setTitle("Final exam")
        .setMessage("Do you want a term project?")
        .setPositiveButton("Sure", DialogInterface.OnClickListener { dialog, which ->
            Toast.makeText(this, "I Love it!", Toast.LENGTH_SHORT).show()
        })
        .setNegativeButton("What?", DialogInterface.OnClickListener { dialog, which ->
            Toast.makeText(this, "What?????", Toast.LENGTH_SHORT).show()
        })
        .setNeutralButton("...",DialogInterface.OnClickListener { dialog, which ->
            Toast.makeText(this, "...", Toast.LENGTH_SHORT).show()
        })
        .show()
}
```



AlertDialog (4/5)

- Adding list/multiple choices
  - A traditional single-choice list
    - setItems(array, listener)

```
.setItems(myList.toTypedArray(), DialogInterface.OnClickListener { dialog, which ->
    Toast.makeText(this, "I will use ${myList[which]} ", Toast.LENGTH_SHORT).show() })
```



  - A persistent single-choice list (radio buttons) / multiple-choice list (checkboxes)
    - setMultiChoiceItems()/setSingleChoiceItems()

```
val checkedItems = mutableListOf<String>()
…
.setMultiChoiceItems(myList.toTypedArray(),null,DialogInterface.OnMultiChoiceClickListener { dialog, which, checked ->
    if(checked){
        checkedItems.add(myList[which])
    }
    else{
        checkedItems.remove(myList[which])
    }
})
```

AlertDialog (5/5)

- Adding list/multiple choices
  - Response with Snackbar
    - Very similar to Toast!

```
.setNeutralButton("...",DialogInterface.OnClickListener { dialog, which ->
    //Toast.makeText(this, "...", Toast.LENGTH_SHORT).show()
    val mySnackbar = Snackbar.make(binding.root,checkedItems.toString(),Snackbar.LENGTH_LONG)
    mySnackbar.setAction("Drop"){binding.tView.text="No Coding, No Pain"}
    mySnackbar.show()
})
```