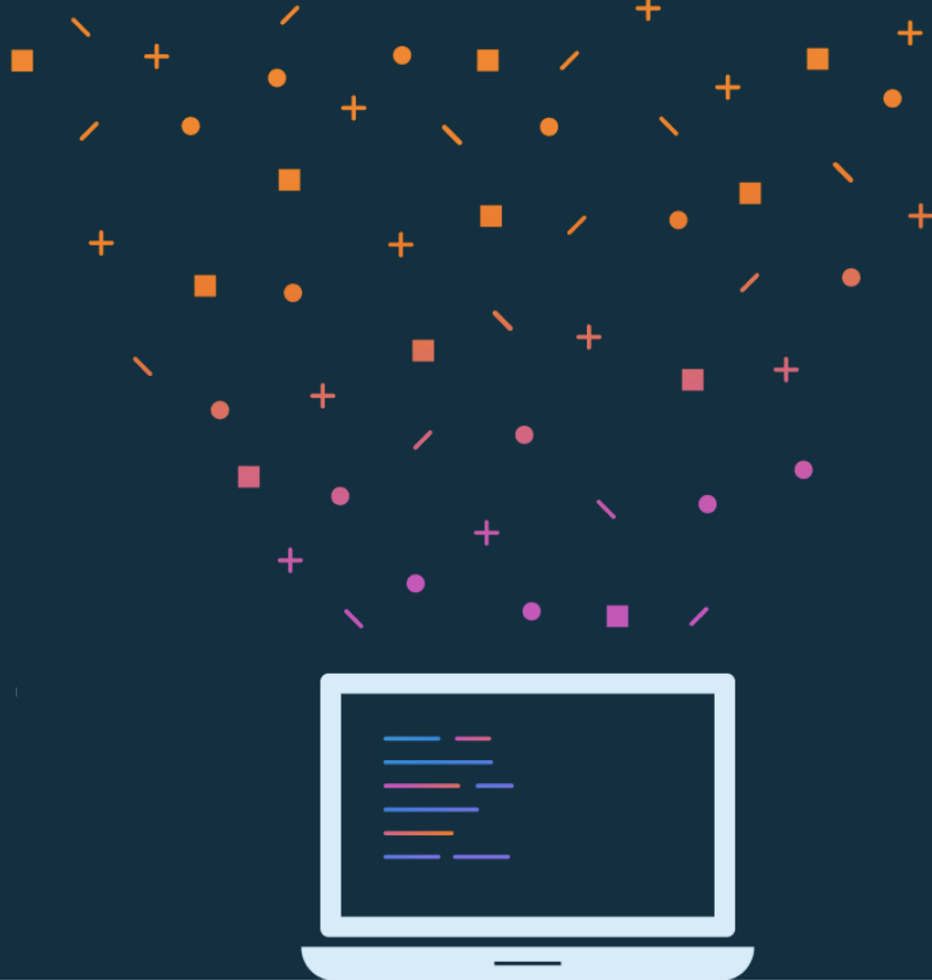




Lesson 10: Advanced RecyclerView use cases



About this lesson

Lesson 10: Advanced RecyclerView use cases

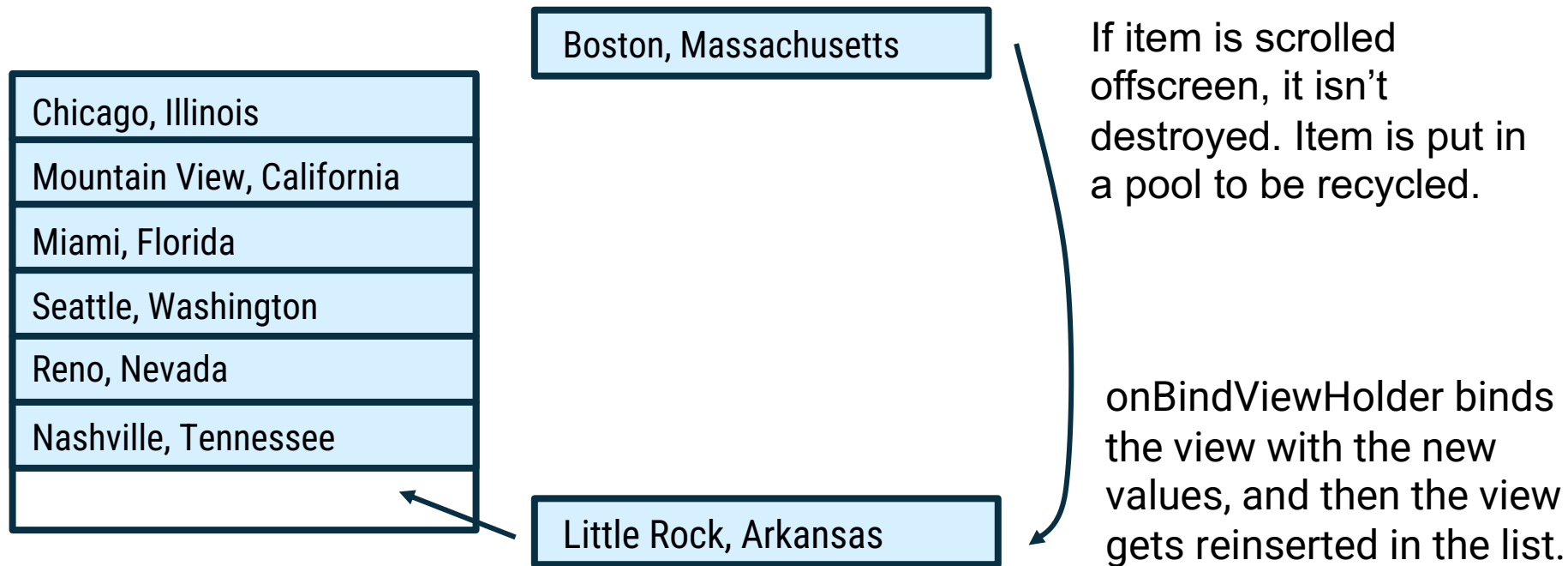
- [RecyclerView recap](#)
- [Advanced binding](#)
- [Multiple item view types](#)
- [Headers](#)
- [Grid layout](#)
- [Summary](#)

RecyclerView recap

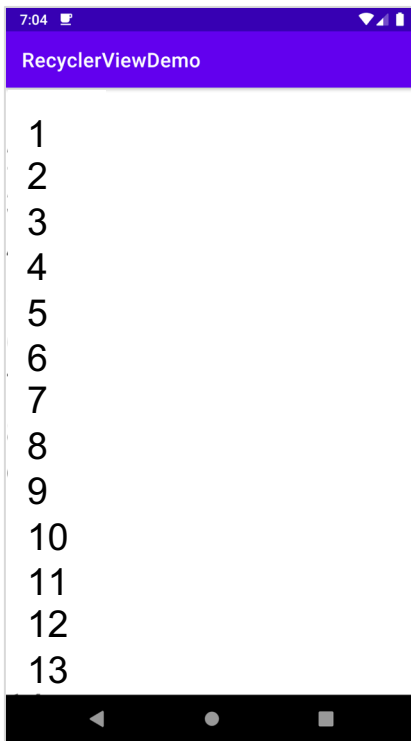
RecyclerView overview

- 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



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

Starting state

1
5
2
6
3
7
4
8

8 deletions

8 insertions

1
2
3
4
5
6
7
8

16 actions:
8 deletions
8 insertions

Ending state

1
2
3
4
5
6
7
8

Sort using ListAdapter

Starting state

1
5
2
6
3
7
4
8



3 insertions
3 deletions

1
5
2
6
3
7
4
5
6
7
8



6 actions:
3 insertions
3 deletions

Ending state

1
2
3
4
5
6
7
8

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 inflater = LayoutInflater.from(parent.context)  
            val binding = ItemViewBinding.inflate(inflater,  
                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

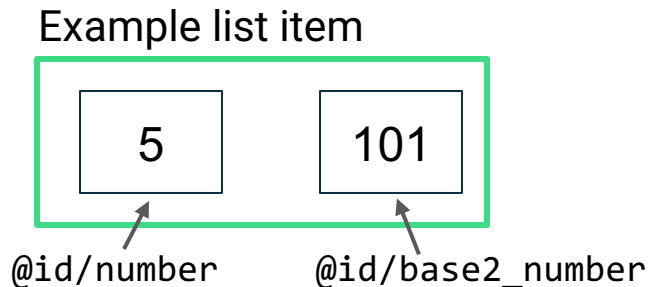
- Create your own custom attributes:

`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

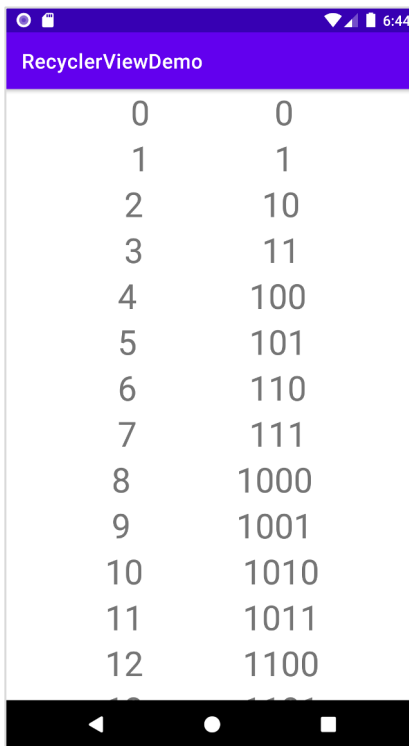
Declare binding adapter:

```
@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



A screenshot of an Android application titled "RecyclerViewDemo". The app displays a list of 13 items, each consisting of an index (0-12) and a corresponding binary value. The values are: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, and 1100. The app is running on a device with a purple status bar and a black navigation bar.

0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100

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 inflater = LayoutInflater.from(parent.context)  
            val binding = ColorItemViewBinding.inflate(inflater,  
                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

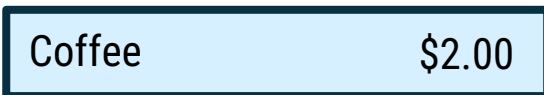
 Entrees	
Burger	\$5.00
Salad	\$3.00
Sandwich	\$4.00
 Drinks	
Coffee	\$2.00
Soda	\$1.00

- 2 item view types:

- header item

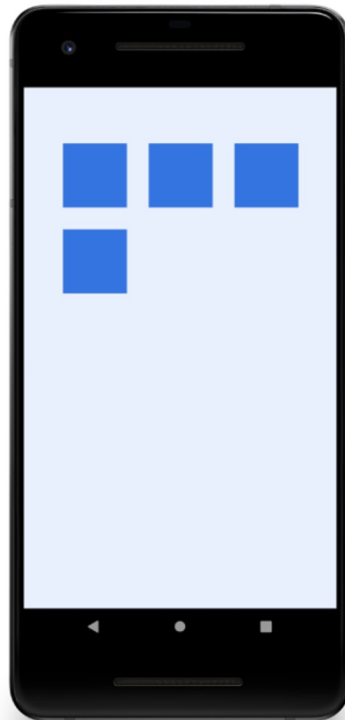


- food menu item



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](#)
- [LayoutManager](#)
- [DiffUtil](#) and [ItemCallback](#)

Pathway

Practice what you've learned by completing the pathway:

[Lesson 10: Advanced RecyclerView use cases](#)

