Extracted from:

Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Kotlin and Android Development *featuring Jetpack*

Build Better, Safer Android Apps

Michael Fazio

edited by Michael Swaine

Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

Michael Fazio

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-815-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Create a Custom ListAdapter

The PlayerSummaryAdapter class is responsible for managing all the PlayerSummary items in our list and handling how they're displayed. We use a custom RecyclerView.ViewHolder inner class (meaning it lives inside PlayerSummaryAdapter) to bind a PlayerSummary item to the layout, then the RecyclerView library handles the rest. All we need to do in PlayerSummaryAdapter is tell the RecyclerView what to do when creating and binding a new ViewHolder plus how to tell the difference between PlayerSummary items in the list.

After creating PlayerSummaryAdapter in the adapters package, first up is the PlayerSummaryViewHolder inner class. The PlayerSummaryAdapter class both contains and depends on this class, so we'll create it first then wrap PlayerSummaryAdapter around it. The PlayerSummaryViewHolder class inherits from RecyclerView.ViewHolder and has a single function, bind(), which takes in a PlayerSummary object.

The bind() function doesn't do much other than assign binding.playerSummary to the item value. The binding value is an instance of PlayerSummaryListItemBinding, which was generated by the Data Binding library when we added the generic <layout> tag to the player_summary_list_item.xml file. The item value, then, is the PlayerSummary object coming into the method. Once that assignment is complete, bind() then ensures bindings are executed so the data shows up properly with the executePendingBindings() function.

```kotlin
inner class PlayerSummaryViewHolder(
  private val binding: PlayerSummaryListItemBinding
) :
  RecyclerView.ViewHolder(binding.root) {

  fun bind(item: PlayerSummary) {
    binding.apply {
      playerSummary = item
      executePendingBindings()
    }
  }
}
```

The PlayerSummaryAdapter class around this inner class inherits from ListAdapter, which takes two type parameters and a DiffUtil.ItemCallback instance. The type parameters are the type of item in the list (PlayerSummary) and the type of ViewHolder for those items (PlayerSummaryAdapter.PlayerSummaryViewHolder). The callback piece is a new private class at the end of the file called (uncreatively) PlayerSummaryDiffCallback. That class looks like this:

```kotlin
private class PlayerSummaryDiffCallback :
  DiffUtil.ItemCallback<PlayerSummary>() {
```

```
  override fun areItemsTheSame(
    oldItem: PlayerSummary,
    newItem: PlayerSummary
  ): Boolean = oldItem.id == newItem.id

  override fun areContentsTheSame(
    oldItem: PlayerSummary,
    newItem: PlayerSummary
  ): Boolean = oldItem == newItem
}
```

With both PlayerSummaryDiffCallback and PlayerSummaryViewHolder ready, we can get PlayerSummaryAdapter created. This class, which inherits from ListAdapter, will also contain a few overridden functions that we'll create in a bit. The class declaration plus the other class and function from before together look like this:

```
class PlayerSummaryAdapter :
  ListAdapter<PlayerSummary, PlayerSummaryAdapter.PlayerSummaryViewHolder>(
    PlayerSummaryDiffCallback()
  ) {

    //Overridden functions will go here in a bit.

  inner class PlayerSummaryViewHolder(
    private val binding: PlayerSummaryListItemBinding
  ) :
    RecyclerView.ViewHolder(binding.root) {

    fun bind(item: PlayerSummary) {
      binding.apply {
        playerSummary = item
        executePendingBindings()
      }
    }
  }
}

private class PlayerSummaryDiffCallback :
  DiffUtil.ItemCallback<PlayerSummary>() {

  override fun areItemsTheSame(
    oldItem: PlayerSummary,
    newItem: PlayerSummary
  ): Boolean =
    oldItem.id == newItem.id

  override fun areContentsTheSame(
    oldItem: PlayerSummary,
    newItem: PlayerSummary
  ): Boolean =
    oldItem == newItem
}
```

An error should be there with the PlayerSummaryAdapter as written since we've yet to implement the two abstract functions from ListAdapter: onCreateViewHolder() and onBindViewHolder(). Both functions are effectively one step, so we can get them done pretty quickly.

onCreateViewHolder() needs to know how to build instances of PlayerSummaryViewHolder. That means we're inflating our layout using the DataBindingUtil class as we have done a few times in this book, sending that into a new PlayerSummaryViewHolder instance, and returning that from the function.

```
override fun onCreateViewHolder(
  parent: ViewGroup,
  viewType: Int
): PlayerSummaryViewHolder =
  PlayerSummaryViewHolder(
    DataBindingUtil.inflate(
      LayoutInflater.from(parent.context),
      R.layout.player_summary_list_item,
      parent,
      false
    )
  )
```

onBindViewHolder() is even more straightforward, as it uses a PlayerSummaryViewHolder instance from onCreateViewHolder(), then sends a PlayerSummary item into the bind() function. We use the getItem() function from the ListAdapter class to get the correct PlayerSummary based on where we are in the list. This is a major advantage of inheriting from the ListAdapter class—it does almost all the work for us as far as handling the items and retrieving the correct one.

```
override fun onBindViewHolder(
  viewHolder: PlayerSummaryViewHolder,
  position: Int
) {
  viewHolder.bind(getItem(position))
}
```

The PlayerSummaryAdapter is now ready for use, so we can head over to the RankingsFragment class to get everything connected.

Connect Adapter to RecyclerView

Here, we're expanding on what we set up earlier with RankingsFragment. Inside the onCreateView() function, we instantiate a PlayerSummaryAdapter object, then assign that to the RecyclerView. Retrieving that RecyclerView object turns out to be easier than previous times we've gotten view components because the entire view we inflated earlier is a <RecyclerView>. As a result, we can convert the view value into a RecyclerView instance, then assign the adapter property. We're also going to add an ItemDecoration to the RecyclerView, which adds light gray lines between each row.

```
override fun onCreateView(
  inflater: LayoutInflater,
  container: ViewGroup?,
  savedInstanceState: Bundle?
): View? {
  val view =
    inflater.inflate(R.layout.fragment_rankings, container, false)
  val playerSummaryAdapter = PlayerSummaryAdapter()

  if (view is RecyclerView) {
    with(view) {
      adapter = playerSummaryAdapter

      addItemDecoration(
        DividerItemDecoration(
          context,
          LinearLayoutManager.VERTICAL
        )
      )
    }
  }

  return view
}
```

This is another great example of smart casting in Kotlin that we first saw in Update roll() and pass() Functions, on page ?. Since we checked that view is an instance of RecyclerView, view is treated in that entire block as a RecyclerView instance without having to create a new value.

Also, we normally would have assigned a value to the layoutManager property on RecyclerView like this:

```
layoutManager = LinearLayoutManager(context)
```

However, it wasn't required since we already handled setting a LayoutManager in the <RecyclerView> tag inside fragment_rankings.xml.

The RecyclerView is now complete and has an assigned adapter to handle all its data. The last piece we need to cover here is how to get that data from the database into the PlayerSummaryAdapter. To do that, we're going to create RankingsViewModel and *observe* a LiveData value from there.