COMP-304 Winter 2023

Review of Lecture 6

- **Creating custom views:**
  - **Modifying** existing views
    - Extend an existing view
    - Override three constructors
    - Optionally override onDraw and onKeyDown methods
    - Add the **new attributes** by creating **getter** and **setter** methods
    - **Create a custom attribute**, generally in a res/values/attrs.xml file that contains one or more <declare-styleable> elements
- Creating **compound** views
  - Inherit the new view from a layout manager, for example LinearLayout
  - Override the three constructors
  - Add existing views to the layout
  - Create corresponding event handlers
  - Optionally, add new attributes
- **Build new Views:**
  - Inherit it from View class
  - Override onDraw method

Review of Lecture 6

- Drawing on screen
  - Use **Canvas** methods to draw
  - Use **Paint** object to set painting features(*color*, *style*,etc)
- **Paint Gradients**:
  - LinearGradient
  - RadialGradient
  - SweepGradient
  - set the Paint gradient using the setShader() method.
- **Working with Text:**
  - Create a **Typeface** object
  - Set *text size* and *typeface* using **Paint** methods
  - Draw text using Canvas method **drawText**.
- **Drawing Bitmap Graphics on a Canvas**
  - draw bitmaps within the **onDraw()** method of a View, using one of the **drawBitmap()** methods
- Transforming Bitmaps Using **Matrix** class
  - Matrix class contain methods to perform tasks such as **mirroring**, **scaling** and **rotating** graphics

Review of Lecture 6

- **Using Shapes**
  - Draw primitive shapes such as rectangles and ovals using the ShapeDrawable class in conjunction with a variety of specialized Shape classes
  - Define shape in xml
  - Draw it on ImageView using setImageResource method.
  - define ShapeDrawable instances programmatically
  - Draw shape on Imageview using setImageDrawable method
- **Drawing on ImageView**
  - Create a **BitMap** as content view for the image
  - Construct a **canvas** with the specified bitmap to draw into
  - Create a **Paint** object
  - Use **Canvas methods** (**drawLine**, etc.) to draw on the image

Review of Lecture 6

- Applying **tweened animation transformations** to View objects
  - define tweening transformations as **XML resource** files or **programmatically:**
    - **Transparency** changes (Alpha)
    - **Rotations** (Rotate)
    - **Scaling** (Scale)
    - **Movement** (Translate)
  - store animation sequences as specially formatted XML files within the **/res/anim/** resource directory
  - load an animation from xml file using AnimationUtils.**loadAnimation method**

Using Internet Resources

Objectives:

- **Apply Kotlin Coroutines** to download and process Internet resources on background threads
- Apply Flow to emit and collect data
- Parse internet resources (XML, JSON)

Connecting to an Internet Resource

- First, you need to add an INTERNET uses-permission node to your application manifest:

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

- Use URL and HttpURLConnection to **create a new HTTP URL connection**:

```
val url = URL(loginUrl)
(url.openConnection() as? HttpURLConnection)?.run {
        requestMethod = "POST"
        setRequestProperty("Content-Type", "application/json; utf-8")
        setRequestProperty("Accept", "application/json")
        doOutput = true
        outputStream.write(jsonBody.toByteArray())
       return Result.Success(responseParser.parse(inputStream))
    }
```

Connecting to an Internet Resource

- Time-intensive operations such as **networking should not block the main UI thread**.
- Retrieving larg amount of data and additional processing such as XML parsing, are **time-intensive operations and should be moved off of the main UI thread**.
- Android uses Kotlin **coroutines** to **manage long-running tasks** that might otherwise block the main thread and cause your app to become unresponsive.
- Over 50% of professional developers who use coroutines have reported seeing increased productivity.

Kotlin Coroutines

- A **coroutine** is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously.
  - Coroutines were added to Kotlin in version 1.3 and are based on established concepts from other languages.
- **Coroutines** is Google's recommended solution for asynchronous programming on Android.

Kotlin Coroutines

Advantages over previous solutions (Threading and AsyncTask):

- **Lightweight**: You can run many coroutines on a single thread due to support for suspension, which doesn't block the thread where the coroutine is running.
  - Suspending saves memory over blocking while supporting many concurrent operations.
- **Fewer memory leaks**: Use structured concurrency to run operations within a scope.
- **Built-in cancellation support**: Cancellation is propagated automatically through the running coroutine hierarchy.
- **Jetpack integration**: Many Jetpack libraries include extensions that provide full coroutines support.
  - Some libraries also provide their own coroutine scope that you can use for structured concurrency.

Kotlin Coroutines

- **SimpleCoroutinesExample**
- In **onCreate** method:

```
// Launch a new coroutine without blocking the current thread.
// Returns a reference to the coroutine as a Job.
// The coroutine is cancelled when the resulting job is cancelled.
CoroutineScope(Dispatchers.IO).launch {
    makeApiCalls()
}
private suspend fun makeApiCalls() {
        val result1 = fetchDataFromFirstApi()
        println("Result $result1")
        val result2 = fetchDataFromSecondApi()
        println("Result $result2")
}
private suspend fun fetchDataFromFirstApi(): String {
        delay(1000)
        return "Result 1"
}
```

Flow

- A flow is a type that **can emit multiple values sequentially**, as opposed to suspend functions that return only a single value.
- A flow is conceptually a stream of data that can be computed **asynchronously**.
- Flows are **built on top of coroutines** and can provide multiple values.
- A flow for example can be used in an app that needs to receive live updates from a database.
- The emitted values must be of the same type - for example, a flow<Int> is a flow that emits integer values.
- A flow can safely make a network request to produce the next value without blocking the main thread.

Flow

❑ KotlinFlowExample

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        //
        val textView = findViewById(R.id.text_view_id) as TextView

        // Creates a cold flow from the given suspendable block.
        // The flow being cold means that the block is called every time
        // a terminal operator is applied to the resulting flow.
        val flow = flow<Int> {
            for (i in 1..10) {
                emit(i) // collects the value emitted by the upstream
                delay(1000L)
            }
        }
```

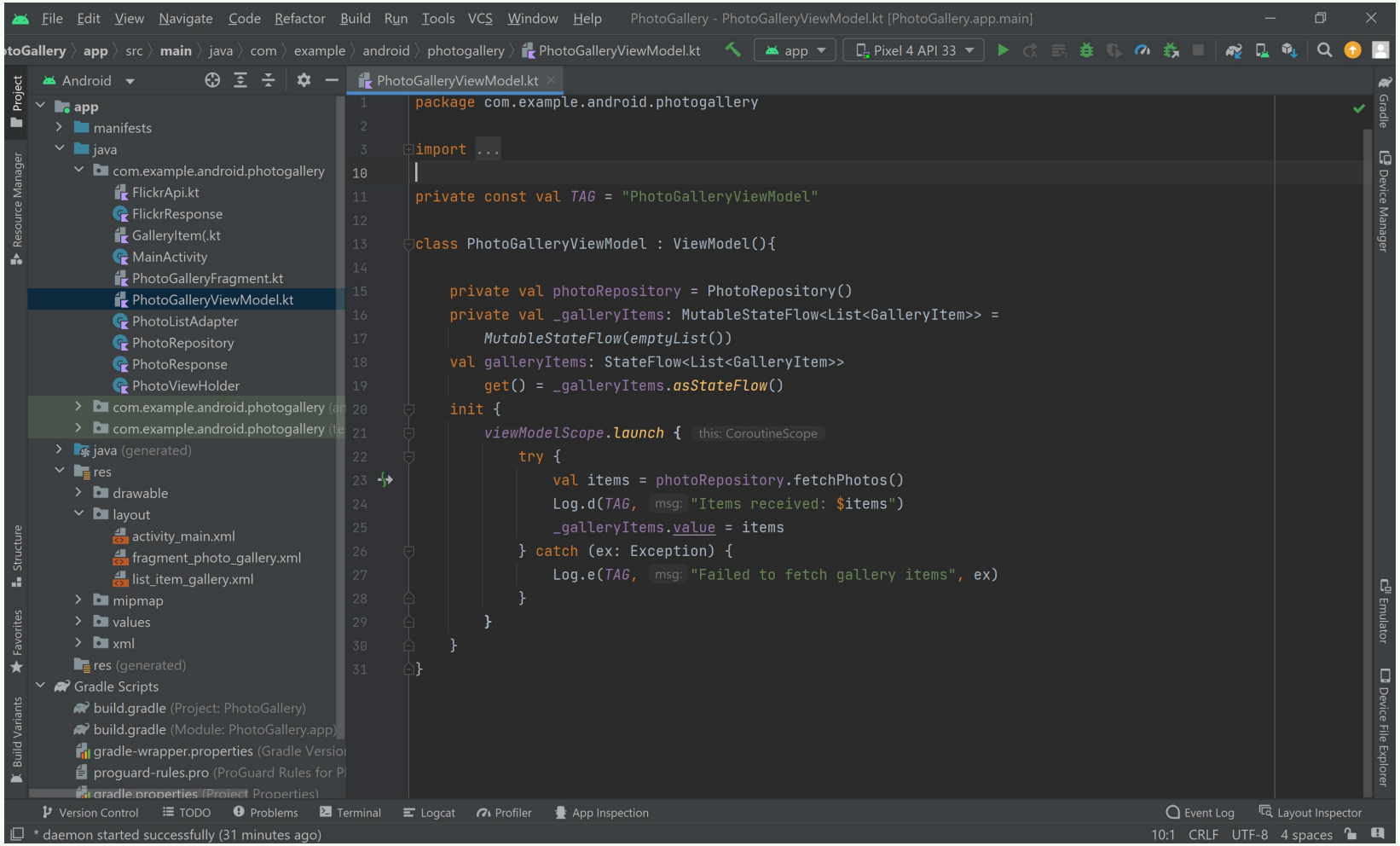
Flow

```
        // Launches a new coroutine without blocking the current thread
        // and returns a reference to the coroutine as a Job
        lifecycleScope.launch {
            flow.buffer().filter {
                it % 2 == 0
            }.map {
                it * it
            }.collect {
                println("debug: $it")
                textView.text = textView.text.toString() + "\n" + it.toString() + "\n"
                delay(2000L)
            }
        }
    }
}
```

Making Network Requests

- **Retrofit** is the de facto **official way to communicate with an HTTP API on Android**.
  - Retrofit is an open-source library created and maintained by Square (square.github.io/retrofit).
  - It is highly configurable and extendable, allowing you to easily and safely communicate with a remote web server.
  - It is organized into components that serve a specific purpose, and you can swap out individual components as you need.
- Retrofit is meant to **define the contracts for many different types of network requests**.
  - Similar to using the Room database library, you **write an interface with annotated instance methods**, and Retrofit creates the implementation.
  - Under the hood, Retrofit's implementation **uses OkHttp**, another library by Square, **to handle making an HTTP request and parsing the HTTP response**.

PhotoGallery Example



PhotoGallery Example

- We will use libraries and tools like the **Fragment** class, the **ViewModel** class, the **RecyclerView** component, and View Binding.
- Enable viewBinding:

```
buildFeatures {
        viewBinding true
    }
```

- Add the related dependencies:

```
implementation 'androidx.fragment:fragment-ktx:1.5.2'
implementation 'androidx.recyclerview:recyclerview:1.2.1'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1'
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.5.1'
```

PhotoGallery Example

- PhotoGallery will display its results in a RecyclerView, using the built-in GridLayoutManager to arrange the items in a grid.
- The Kotlin class will be named PhotoGalleryFragment.

PhotoGallery Example

```
private const val TAG = "PhotoGalleryFragment"
//
class PhotoGalleryFragment : Fragment() {
    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cannot access binding because it is null. Is the view visible?"
        }
    private val photoGalleryViewModel: PhotoGalleryViewModel by viewModels()

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding =
            FragmentPhotoGalleryBinding.inflate(inflater, container, false)
        binding.photoGrid.layoutManager = GridLayoutManager(context, 3)
        return binding.root
    }
```

PhotoGallery Example

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            try {
                val response = PhotoRepository().fetchPhotos()
                Log.d(TAG, "Response received: $response")
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
            //
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.galleryItems.collect { items ->
                    //Log.d(TAG, "Response received: $items")
                    binding.photoGrid.adapter = PhotoListAdapter(items)
                }
            }
            //
        }
    }
```

PhotoGallery Example

```
//
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

- Add Retrofit dependencies

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.okhttp3:okhttp:4.9.3'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.1'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.1'
implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'
```

PhotoGallery Example

- Create an API key with **flickr**.
- Defining an API interface:

```
private const val API_KEY = “abd21511931b1d2288218074d515a4c1"
//
interface FlickrApi {
    @GET(
        "services/rest/?method=flickr.interestingness.getList" +
                "&api_key=$API_KEY" +
                "&format=json" +
                "&nojsoncallback=1" +
                "&extras=url_s"
    )
    suspend fun fetchPhotos(): FlickrResponse

}
```

PhotoGallery Example

- Making a network request:

```
viewLifecycleOwner.lifecycleScope.launch {
        try {
            val response = PhotoRepository().fetchPhotos()
            Log.d(TAG, "Response received: $response")
        } catch (ex: Exception) {
            Log.e(TAG, "Failed to fetch gallery items", ex)
        }
```

- When you call **fetchPhotos()**, Retrofit automatically **executes the request on a background thread**.
  - Retrofit manages the background thread for you, so you do not have to worry about it.
  - When it receives a response, thanks to coroutines, it will pass the result back on the thread where it was first invoked, which in this case is the UI thread.

PhotoGallery Example

- Retrofit makes it easy to respect the two most important Android threading rules:
  1. Execute long-running operations only on a background thread, never on the main thread.
  2. Update the UI only from the main thread, never from a background thread.
- Asking permission to network:
  - add the following permission to AndroidManifest.xml file:

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

PhotoGallery Example

- Put Retrofit configuration code and API direct access in a separate class:

```
class PhotoRepository {

    private val flickrApi: FlickrApi
    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
        flickrApi = retrofit.create()
    }
    suspend fun fetchPhotos(): List<GalleryItem> =
        flickrApi.fetchPhotos().photos.galleryItems
}
```

PhotoGallery Example

- Retrofit.Builder() is a fluent interface that makes it easy to configure and build your Retrofit instance.
- You provide a base URL for your endpoint using the baseUrl(…) function. Here, you provide the Flickr home page: "https://www.flickr.com/".
- Calling build() returns a Retrofit instance, configured based on the settings you specified using the builder object.
- Once you have a Retrofit object, you use it to create an instance of your API interface.
- Retrofit does not generate any code at compile time – instead, it does all the work at runtime.
- When you call retrofit.create(), Retrofit uses the information in the API interface you specify, along with the information you specified when building the Retrofit instance, to create and instantiate an anonymous class that implements the interface on the fly.

PhotoGallery Example

- **Adding a String converter:**
- To get Retrofit to deserialize the response into strings instead, you will **specify a converter** when building your Retrofit object.
- A converter knows how **to decode a ResponseBody** object into some other object type.
- Square created an open-source converter, called the **scalars converter**, that can **convert the response into a string**.
- You will use it to deserialize Flickr responses into string object
- Add the scalars converter dependency (app/build.gradle):

```
implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'
```

PhotoGallery Example

- Fetching JSON from Flickr:
- You can make a GET request to the Flickr web service:
- https://api.flickr.com/services/rest/?method=flickr.interestingness.getList&api_key=yourApiKeyHere&format=json&nojsoncallback=1&extras=url_s
- See the Kotlin code in FlickrApi interface that parses the results to JSON:

```
@GET(
        "services/rest/?method=flickr.interestingness.getList" +
                "&api_key=$API_KEY" +
                "&format=json" +
                "&nojsoncallback=1" +
                "&extras=url_s"
    )
```

PhotoGallery Example

- Create a model class called **GalleryItem** to hold meta information for a single photo, including the **title**, the **ID**, and the **URL** to download the image from:

```
@JsonClass(generateAdapter = true)
data class GalleryItem(
    val title: String,
    val id: String,
    @Json(name = "url_s") val url: String,
)
```

- Android includes the standard org.json package, which has classes that provide access to creating and parsing JSON text (such as **JSONObject** and **JSONArray**).
- Alternatively, use **Moshi** (github.com/square/moshi) another library from Square.
  - Moshi **maps JSON data to Kotlin objects** automatically.

PhotoGallery Example

- To configure Moshi to do all those things for you, first enable the **kapt** plugin:
- It is defined at the project level, so add the following line to the build.gradle file labeled (Project: PhotoGallery):

```
id 'org.jetbrains.kotlin.kapt' version '1.6.10' apply false
```

- Once you have enabled the plugin, apply it to your app's build process in app/build.gradle.

```
id 'org.jetbrains.kotlin.kapt'
```

- Include the core library as well as the library that performs the code generation in your dependencies (build.gradle(app)):

```
implementation 'com.squareup.moshi:moshi:1.13.0'
kapt 'com.squareup.moshi:moshi-kotlin-codegen:1.13.0'
implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
```

PhotoGallery Example

- Create a PhotoResponse class to map to the "photos" object in the JSON data.
- Place the new class in the api package as well.
- Include a property called **galleryItems** to store a list of gallery items and annotate it with **@Json(name = "photo")**.
- Moshi will automatically create a list and populate it with gallery item objects based on the JSON array named "photo".

```
@JsonClass(generateAdapter = true)
data class PhotoResponse(
    @Json(name = "photo") val galleryItems: List<GalleryItem>
)
```

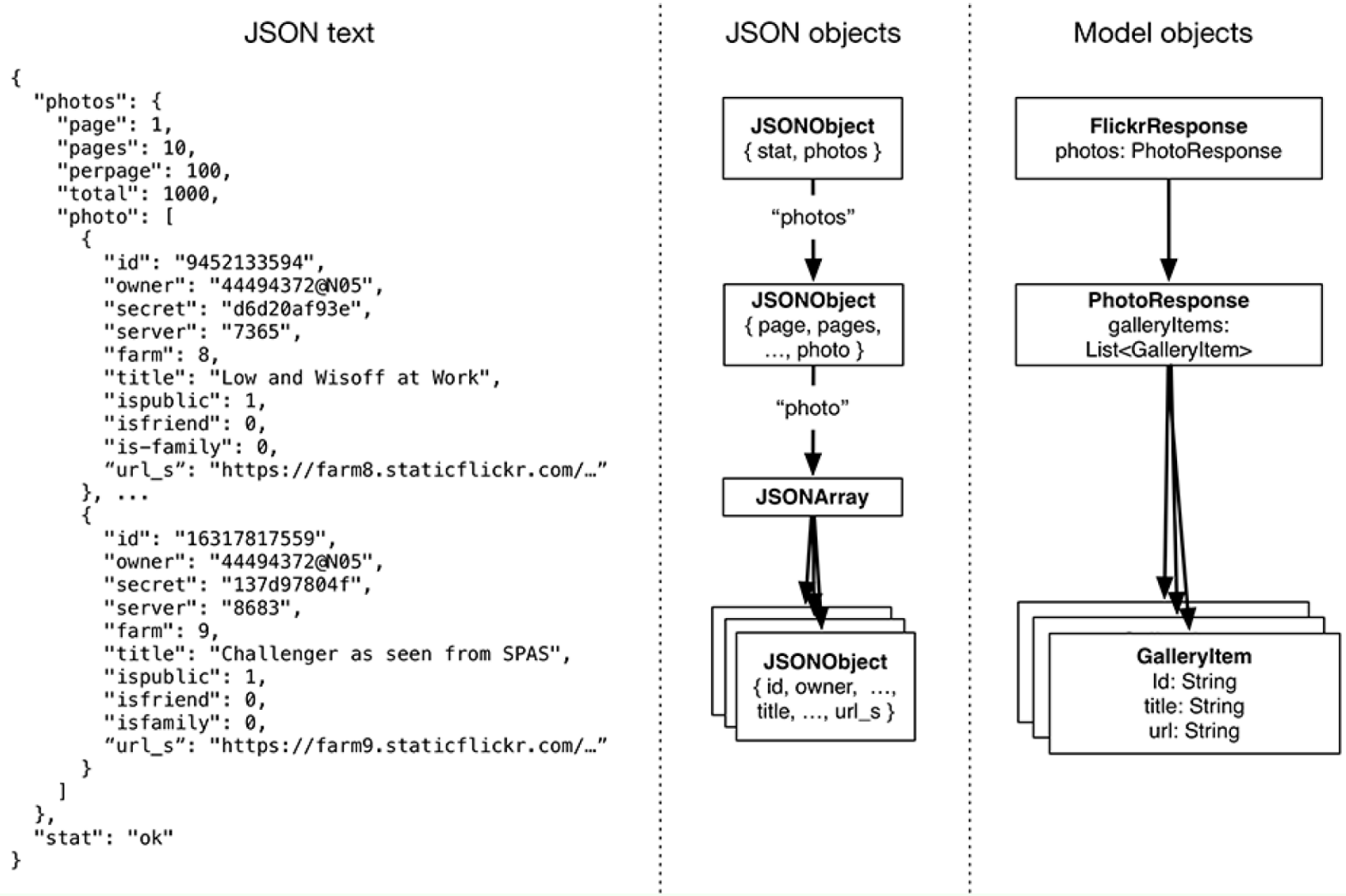
PhotoGallery Example

- Finally, add a class named **FlickrResponse** to the api package.
- This class will map to the outermost object in the JSON data (the one at the top of the JSON object hierarchy, denoted by the outermost { }).
- Add a property to map to the "photos" field.

```
@JsonClass(generateAdapter = true)
data class FlickrResponse(
    val photos: PhotoResponse
)
```

PhotoGallery Example

Figure 20.8 PhotoGallery data and model objects



PhotoGallery Example

- Create a ViewModel class named **PhotoGalleryViewModel:**

```
private const val TAG = "PhotoGalleryViewModel"
class PhotoGalleryViewModel : ViewModel(){
    private val photoRepository = PhotoRepository()
    private val _galleryItems: MutableStateFlow<List<GalleryItem>> =
        MutableStateFlow(emptyList())
    val galleryItems: StateFlow<List<GalleryItem>>
        get() = _galleryItems.asStateFlow()
    init {
        viewModelScope.launch {
            try {
                val items = photoRepository.fetchPhotos()
                Log.d(TAG, "Items received: $items")
                _galleryItems.value = items
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
        }
    }
}
```

PhotoGallery Example

- Use a **StateFlow** to expose a list of gallery items to the fragment.
- Start a web request to fetch photo data when the ViewModel is first initialized, and stash the resulting data in the property you created.
- Use a **try/catch** block to handle any errors.
- Recall that the first time a ViewModel is requested for a given lifecycle owner, a new instance of the ViewModel is created.
- Successive requests for the ViewModel return the same instance that was originally created.
- When the user rotates the device or some other configuration change occurs, the **ViewModel will remain in memory**, and the re-created version of the fragment will be able to access the results of the original request through the ViewModel.
- Thanks to coroutines, when the **viewModelScope** is canceled, your network request will also be canceled

PhotoGallery Example

- Displaying Results in RecyclerView:
- Get PhotoGalleryFragment's RecyclerView to display some images in ImageView.
- We need to create two Kotlin classes: one that will extend RecyclerView.**ViewHolder** and another that will extend RecyclerView.**Adapter**.

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {

    fun bind(galleryItem: GalleryItem) {
        // TODO
        binding.itemImageView.load(galleryItem.url)
    }
}
```

PhotoGallery Example

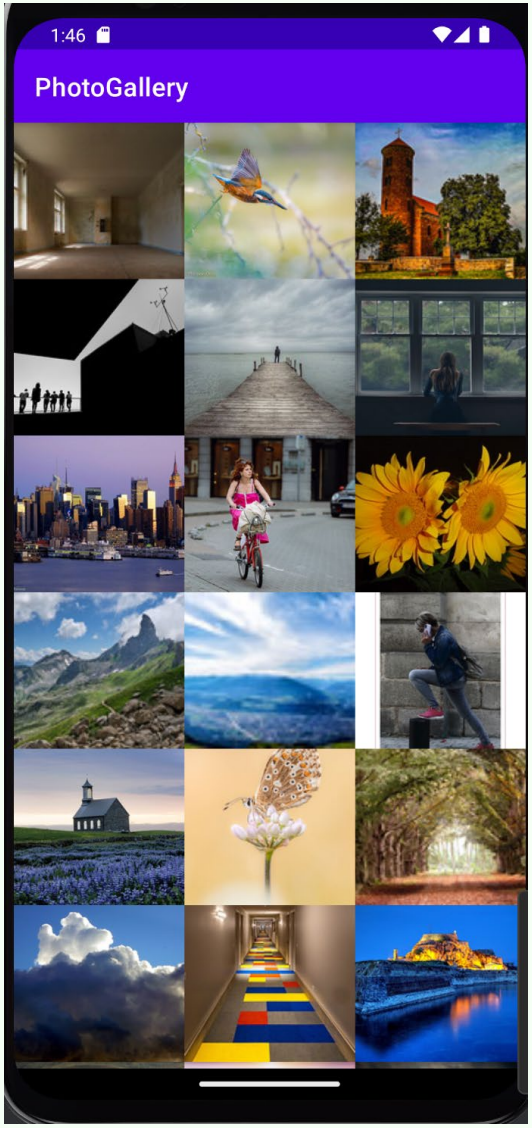
```
class PhotoListAdapter(
    private val galleryItems: List<GalleryItem>
) : RecyclerView.Adapter<PhotoViewHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemGalleryBinding.inflate(inflater, parent, false)
        return PhotoViewHolder(binding)
    }
    override fun onBindViewHolder(holder: PhotoViewHolder, position: Int) {
        val item = galleryItems[position]
        holder.bind(item)
    }
    override fun getItemCount() = galleryItems.size
}
```

PhotoGallery Example

- Displaying images
- Efficient image loading is a hard problem:
  - need to worry about network connections, juggling images across threads, caching images, resizing images to fit their containers, canceling requests when images are no longer needed, and much more.
- We will use **Coil**, originally developed at **Instacart**.
  - Coil leverages all the convenient features of the modern Kotlin language and integrates seamlessly with coroutines to manage performing work in the background.
  - Add the dependency:

    implementation 'io.coil-kt:**coil**:2.0.0-rc02'

PhotoGallery Example



References

- Textbook, Chapter 20
- https://kotlinlang.org/docs/async-programming.html#coroutines
- https://developer.android.com/codelabs/kotlin-coroutines#1
- https://developer.android.com/codelabs/advanced-kotlin-coroutines#1
- https://developer.android.com/kotlin/flow
- https://developer.android.com/topic/libraries/architecture/livedata#java
- https://developer.android.com/courses/android-development-with-kotlin/course