Photo Gallery

based on Chapter 25 of
Android Programming: A Big Nerd Ranch Guide (4th edition)

Photo Layout for Recycler View

- list_item_gallery
- placeholder.jpg

This layout will be used for items (images) in the recycler view

list_item_gallery

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="120dp"
    android:layout_gravity="center"
    android:scaleType="centerCrop"/>
```

Drag and drop a placeholder image (placeholder.jpg) into the drawables folder

PhotoGalleryFragment

- PhotoHolder
- PhotoAdapter

```
private inner class PhotoHolder(itemImageView: ImageView)
    : RecyclerView.ViewHolder(itemImageView) {
    val bindDrawable: (Drawable) -> Unit = itemImageView::setImageDrawable
}

private inner class PhotoAdapter(private val galleryItems: List<Gall
    : RecyclerView.Adapter<PhotoHolder>() {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoHolder {
        val view = layoutInflater.inflate(
            R.layout.list_item_gallery,
            parent,
            false
        ) as ImageView
        return PhotoHolder(view)
    }

    override fun getItemCount(): Int = galleryItems.size

    override fun onBindViewHolder(holder: PhotoHolder, position: Int) {
        val galleryItem = galleryItems[position]
        val placeholder: Drawable = ContextCompat.getDrawable(
            requireContext(),
            R.drawable.placeholder
        ) ?: ColorDrawable()
        holder.bindDrawable(placeholder)
        // thumbnailDownloader.queueThumbnail(holder, galleryItem.url)
    }
}
```

Replace the text-view code formerly in the holder with image-view code

```kotlin
private inner class PhotoHolder(itemImageView: ImageView)
    : RecyclerView.ViewHolder(itemImageView) {
    val bindDrawable: (Drawable) -> Unit = itemImageView::setImageDrawable
}

private inner class PhotoAdapter(private val galleryItems: List<GalleryItem>)
    : RecyclerView.Adapter<PhotoHolder>() {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoHolder {
        val view = layoutInflater.inflate(
            R.layout.list_item_gallery,
            parent,
            false
        ) as ImageView
        return PhotoHolder(view)
    }

    override fun getItemCount(): Int = galleryItems.size

    override fun onBindViewHolder(holder: PhotoHolder, position: Int) {
        val galleryItem = galleryItems[position]
        val placeholder: Drawable = ContextCompat.getDrawable(
            requireContext(),
            R.drawable.placeholder
        ) ?: ColorDrawable()
        holder.bindDrawable(placeholder)
        // thumbnailDownloader.queueThumbnail(holder, galleryItem.url)
    }
}
```

Instead of a text-view, create an image-view and pass that into the photo-holder's constructor

```
private inner class PhotoHolder(itemImageView: ImageView)
    : RecyclerView.ViewHolder(itemImageView) {
    val bindDrawable: (Drawable) -> Unit = itemImageView::setImageDrawable
}

private inner class PhotoAdapter(private val galleryItems: List<GalleryItem>)
    : RecyclerView.Adapter<PhotoHolder>() {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoHolder {
        val view = layoutInflater.inflate(
            R.layout.list_item_gallery,
            parent,
            false
        ) as ImageView
        return PhotoHolder(view)
    }

    override fun getItemCount(): Int = galleryItems.size

    override fun onBindViewHolder(holder: PhotoHolder, position: Int
        val galleryItem = galleryItems[position]
        val placeholder: Drawable = ContextCompat.getDrawable(
            requireContext(),
            R.drawable.placeholder
        ) ?: ColorDrawable()
        holder.bindDrawable(placeholder)
        // thumbnailDownloader.queueThumbnail(holder, galleryItem.url)
    }
}
```

Instead of binding text to the holder, create a drawable from your placeholder image and bind that

The placeholder binding is temporary. Eventually, we want to fetch an image from the network and bind that

For now, we keep the call to the network commented out

Flickr API

- FlickrApi

```
interface FlickrApi {
    @GET(
        "services/rest/?method=flickr.interestingness.getList" +
                "&api_key=${FlickrKey.key}" +
                "&format=json" +
                "&nojsoncallback=1" +
                "&extras=url_s"
    )
    fun fetchPhotos(): Call<FlickrResponse>

    @GET
    fun fetchUrlBytes(@Url url: String): Call<ResponseBody>
}
```

fetch-URL-bytes retrieves the bytes from the JPG image at the specified URL

@GET does not need a string because @Url tells Retrofit to use the url string

ResponseBody is a type from the OkHttp library that holds the body of a raw HTTP response

FlickrFetchr

- FlickrFetcher

Unlike in BNR, we're going to make flickr-fetchr an object (singleton)

If it was a class before, that will break things (like the call to fetchPhotos)

But that should be easy enough to fix

```
object FlickrFetchr {

    ...

    @WorkerThread
    fun fetchPhoto(url: String): Bitmap? {
        val response: Response<ResponseBody> = flickrApi.fetchUrlBytes(url).execute()
        val bitmap = response.body()?.byteStream()?.use(BitmapFactory::decodeStream)
        Log.i(TAG, "Decoded bitmap=$bitmap from Response=$response")
        return bitmap
    }
}
```

The @WorkerThread annotation says that this function should only be called on a background thread

It does *not* create a background thread for you – you have to do that yourself.

A worker thread and a background thread are the same thing -- any thread created separately from the main UI thread.

```
object FlickrFetchr {

    ...

    @WorkerThread
    fun fetchPhoto(url: String): Bitmap? {
        val response: Response<ResponseBody> = flickrApi.fetchUrlBytes(url).execute()
        val bitmap = response.body()?.byteStream()?.use(BitmapFactory::decodeStream)
        Log.i(TAG, "Decoded bitmap=$bitmap from Response=$response")
        return bitmap
    }
}
```

FetchPhoto (singular) takes a URL and returns a bitmap image

We use execute (synchronous) instead of enqueue here because we are already working on a background thread.

```
object FlickrFetchr {

    ...

    @WorkerThread
    fun fetchPhoto(url: String): Bitmap? {
        val response: Response<ResponseBody> = flickrApi.fetchUrlBytes(url).execute()
        val bitmap = response.body()?.byteStream()?.use(BitmapFactory::decodeStream)
        Log.i(TAG, "Decoded bitmap=$bitmap from Response=$response")
        return bitmap
    }
}
```

The type of fetchUrlBytes is Call<ResponseBody>

The type of execute is Response<T> where T is the type parameter of the Call object

Thumbnail Downloader

- Thumbnail Downloader

Message Handler Classes

- Handler Thread – starts a new thread with a looper
  – our handler thread with be ThumbnailDownloader
- Looper – runs the message loop for the thread
- MessageQueue – holds the list of messages
- Message – holds data to be sent to the handler
  – the fields of interest in our messages are what, obj, and target
- Handler – creates, schedules, and processes messages
  – Handler threads can have multiple handlers; ours only has one

Handler threads only have one Looper and one MessageQueue. In our code, we will not access these objects directly

Anatomy of a Message

- what – an identifier for the kind of message
  – in our case, it will always be MESSAGE_DOWNLOAD
- obj – user specified object, sent to the message
  – in our case, it will be a PhotoHolder object
- target – the handler that will handle the message
  – in our case, it will always be requestHandler

There are other fields, but these three are the ones we will use

```kotlin
private const val TAG = "ThumbnailDownloader"
private const val MESSAGE_DOWNLOAD = 0

class ThumbnailDownloader<in H: Any>(
    private val responseHandler: Handler,
    private val onThumbnailDownloaded: (H, Bitmap) -> Unit
) : HandlerThread(TAG) {

    ...
```

The TAG will serve as an identifier for the handler thread

MESSAGE_DOWNLOAD will be the "what" of all messages

H is the generic type that identifies each download

The use of "Any" here ensures that H is non-nullable

For PGF, this will be the PhotoHolder, because the image will be bound to the holder

```
private const val TAG =
private const val MESSAG

class ThumbnailDownloader<in H: Any>(
    private val responseHandler: Handler,
    private val onThumbnailDownloaded: (H, Bitmap) -> Unit
) : HandlerThread(TAG) {

    ...
```

TD takes a Handler. PGF will pass in a handler for the UI thread

TD also takes a function (onThumbnailDownloaded)

The handler thread constructor *requires* a name as an identifier

PGF will pass in a function that binds the bitmap image to the holder. It will be run on the UI thread

```
val fragmentLifecycleObserver: DefaultLifecycleObserver =
    object : DefaultLifecycleObserver {

    override fun onCreate(owner: LifecycleOwner) {
        Log.i(TAG, "Starting background thread")
        start()
        looper
    }

    override fun onDestroy(owner: LifecycleOwner) {
        Log.i(TAG, "Destroying background thread")
        quit()
    }
}

val vi                                  oserver =
    ob

    ov                                  wner) {
                                        om queue")
                                        E_DOWNLOAD)

    }
}
```

The use of annotation @onLifecycleEvent (as in BNR) is deprecated. Use DefaultLifecycleObserver instead (as shown here)

When the fragment's onCreate and onDestroy methods are called, these are also called, respectively

PGF adds this observer during onCreate, and removes it during onDestroy

`start` starts the background thread,
`looper` starts the looper, and
`quit` quits the background thread

```
val fragmentLifecycleObserver: DefaultLifecycleObserver =
    object : DefaultLifecycleObserver {

    override fun onCreate(owner: LifecycleOwner) {
        Log.i(TAG, "Starting background thread")
        start()
        looper
    }

    override fun onDestroy(owner: LifecycleOwner) {
        Log.i(TAG, "Destroying background thread")
        quit()
    }
}

val viewLifecycleObserver: DefaultLifecycleObserver =
    object : DefaultLifecycleObserver {

    override fun onDestroy(owner: LifecycleOwner) {
        Log.i(TAG, "Clearing all requests from queue")
        requestHandler.removeMessages(MESSAGE_DOWNLOAD)
        requestMap.clear()
    }
}
```

This property will be added to the *view* lifecycle

Therefore, when the fragment's *onDestroyView* is called, this is also called

When the view is destroyed (ex: on rotation) messages are removed from the handler and the request map is cleared

```kotlin
private var hasQuit = false
private lateinit var requestHandler: Handler
private val requestMap = ConcurrentHashMap<H, String>()

@Suppress("UNCHECKED_CAST")
@SuppressLint("HandlerLeak")
override fun onLooperPrepared() {
    requestHandler = object : Handler(Looper.myLooper()!!) {
        override fun handleMessage(msg: Message) {
            if (msg.what == MESSAGE_DOWNLOAD) {
                val holder = msg.obj as H
                Log.i(TAG, "Got a request for URL: ${requestMap[holder]}")
                handleRequest(holder)
            }
        }
    }
}

override fun quit(): Boolean {
    hasQuit = true
    return super.quit()
}
```

The requestHandler is the only handler for this thread

The requestMap holds the URL's associated with a specific photo holder onLooperPrepared is called once before the looper starts, so it's a good place to setup the request handler

```
private var hasQuit = false
private lateinit var requestHandler: Handler
private val requestMap = ConcurrentHashMap<H, String>()

@Suppress("UNCHECKED_CAST")
@SuppressLint("HandlerLeak")
override fun onLooperPrepared() {
    requestHandler = object : Handler(Looper.myLooper()!!) {
        override fun handleMessage(msg: Message) {
            if (msg.what == MESSAGE_DOWNLOAD) {
                val holder = msg.obj as H
                Log.i(TAG, "Got a request for URL: ${requestMap[holder]}")
                handleRequest(holder)
            }
        }
    }
}

ove

}
```

handleMessage gets the photo holder and calls handleRequest

The requestHandler runs on the background thread and handles messages (requests) sent from PGF

The responseHandler runs on the UI thread and handles tasks requested by the requestHandler

```kotlin
private var hasQuit = false
private lateinit var requestHandler: Handler
private val requestMap = ConcurrentHashMap<H, String>()

@Suppress("UNCHECKED_CAST")
@SuppressLint("HandlerLeak")
override fun onLooperPrepared() {
    requestHandler = object : Handler(Looper.myLooper()!!) {
        override fun handleMessage(msg: Message) {
            if (msg.what == MESSAGE_DOWNLOAD) {
                val holder = msg.obj as H
                Log.i(TAG, "Got a request for URL: ${requestMap[holder]}")
                handleRequest(holder)
            }
        }
    }
}

override fun quit(): Boolean {
    hasQuit = true
    return super.quit()
}
```

hasQuit (declared above) keeps track of whether the thread has stopped

```kotlin
    fun queueThumbnail(holder: H, url: String) {
        Log.i(TAG, "Got a URL: $url")
        requestMap[holder] = url
        requestHandler.obtainMessage(MESSAGE_DOWNLOAD, holder)
            .sendToTarget()
    }

    private fun handleRequest(holder: H) {
        val url = requestMap[holder] ?: return
        val bitmap = FlickrFetchr.fetchPhoto(url) ?: return

        responseHandler.post(Runnable {
            if (requestMap[holder] != url || hasQuit) {
                return@Runnable
            }

            requestMap.remove(holder)
            onThumbnailDownloaded(holder, bitmap)
        })
    }
}
```

Put the (photo-holder, url) pair into the request map, then create the message and send it to the queue

```
    fun queueThumbnail(holder: H, url: String) {
        Log.i(TAG, "Got a URL: $url")
        requestMap[holder] = url
        requestHandler.obtainMessage(MESSAGE_DOWNLOAD, holder)
            .sendToTarget()
    }

    private fun handleRequest(holder: H) {
        val url = requestMap[holder] ?: return
        val bitmap = FlickrFetchr.fetchPhoto(url) ?: return

        responseHandler.post(Runnable {
            if (requestMap[holder] != url || hasQuit) {
                return@Runnable
            }

            requestMap.remove(holder)
            onThumbnailDownloaded(holder, bitmap)
        })
    }
}
```

Get url from the request-map

Get image from Flickr-fetcher

```
    fun queueThumbnail(holder: H, url: String) {
        Log.i(TAG, "Got a URL: $url")
        requestMap[holder] = url
        requestHandler.obtainMessage(MESSAGE_DOWNLOAD, holder)
            .sendToTarget()
    }

    private fun handleRequest(holder: H) {
        val url = requestMap[holder] ?: return
        val bitmap = FlickrFetchr.fetchPhoto(url) ?: return

        responseHandler.post(Runnable {
            if (requestMap[holder] != url || hasQuit) {
                return@Runnable
            }

            requestMap.remove(holder)
            onThumbnailDownloaded(holder, bitmap)
        })
    }
}
```

If something is wrong, return

Otherwise, remove the (photo-holder, url) pair from the request-map and execute on-thumbnail-downloaded function on the UI thread

Photo Gallery Fragment

- PhotoGalleryFragment

```
class PhotoGalleryFragment : Fragment() {

    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding get() = _binding!!
    private val viewModel: PhotoGalleryViewModel by viewModels()
    private lateinit var thumbnailDownloader: ThumbnailDownloader<PhotoHolder>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val responseHandler = Handler(Looper.getMainLooper())
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler) { photoHolder, bitmap ->
                val drawable = BitmapDrawable(resources, bitmap)
                photoHolder.bindDrawable(drawable)
            }

        lifecycle.addObserver(thumbnailDownloader.fragmentLifecycleObserver)
    }

    ...
```

Declare a thumbnail downloader

It takes a Photo Holder as a type parameter

```kotlin
class PhotoGalleryFragment : Fragment() {

    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding get() = _binding!!
    private val viewModel: PhotoGalleryViewModel by viewModels()
    private lateinit var thumbnailDownloader: ThumbnailDownloader<PhotoHolder>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val responseHandler = Handler(Looper.getMainLooper())
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler) { photoHolder, bitmap ->
                val drawable = BitmapDrawable(resources, bitmap)
                photoHolder.bindDrawable(drawable)
            }

        lifecycle.addObserver(thumbnailDownloader.fragmentLifecycleObserver)
    }

    ...
```

Override the onCreate function

```
class PhotoGalleryFragment : Fragment() {

    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding get() = _binding!!
    private val viewModel: PhotoGalleryViewModel by viewModels()
    private lateinit var thumbnailDownloader: ThumbnailDownloader<PhotoHolder>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val responseHandler = Handler(Looper.getMainLooper())
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler) { photoHolder, bitmap ->
                val drawable = BitmapDrawable(resources, bitmap)
                photoHolder.bindDrawable(drawable)
            }

        lifecycle.addObserver(thumbnailDownloader.fragmentLifecycleObserver
    }

    ...
```

Declare and initialize the response handler

getMainLooper gets the looper for the UI thread

This ensures that the response handler is a handler for the main UI thread

```
class PhotoGalleryFragment : Fragment() {

    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding get() = _binding!!
    private val viewModel: PhotoGalleryViewModel by viewModels()
    private lateinit var thumbnailDownloader: ThumbnailDownloader<PhotoHolder>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val responseHandler = Handler(Looper.getMainLooper())
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler) { photoHolder, bitmap ->
                val drawable = BitmapDrawable(resources, bitmap)
                photoHolder.bindDrawable(drawable)
            }

        lifecycle.addObserver(thumbnailDownloader.fragmentLifecycleObserver
    }

    ...
```

Initialize TD

Pass in the response handler and a lambda function

The lambda becomes on-thumbnail-downloaded inside of TD

```
class PhotoGalleryFragment : Fragment() {

    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding get() = _binding!!
    private val viewModel: PhotoGalleryViewModel by viewModels()
    private lateinit var thumbnailDownloader: ThumbnailDownloader<PhotoHolder>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val responseHandler = Handler(Looper.getMainLooper())
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler) { photoHolder, bitmap ->
                val drawable = BitmapDrawable(resources, bitmap)
                photoHolder.bindDrawable(drawable)
            }

        lifecycle.addObserver(thumbnailDownloader.fragmentLifecycleObserver)
    }

    ...
```

Add the TD's FLO property as an observer of the fragment's lifecycle

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    viewLifecycleOwner.lifecycle.addObserver(
        thumbnailDownloader.viewLifecycleObserver
    )
    _binding = FragmentPhotoGalleryBinding.inflate(inflater, container, false)
    val view = binding.root
    binding.photoRecyclerView.layoutManager = GridLayoutManager(context, 3)
    return view
}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    viewModel.galleryItemLiveData.observe(
        viewLifecycleOwner
    ) { galleryItems ->
        binding.photoRecyclerView.adapter = PhotoAdapter(galleryItems)
    }
}
```

Add the TD's VLO property as an observer of the fragment's *view* lifecycle

No changes to on-view-created

```
override fun onDestroyView() {
    super.onDestroyView()
    viewLifecycleOwner.lifecycle.removeObserver(
        thumbnailDownloader.viewLifecycleObserver
    )
}

override fun onDestroy() {
    super.onDestroy()
    lifecycle.removeObserver(
        thumbnailDownloader.fragmentLifecycleObserver
    )
}
```

Remove the TD's VLO as a view lifecycle observer

Remove the TD's FLO as a fragment lifecycle observer

```
    private inner class PhotoAdapter(private val galleryItems: List<GalleryItem>)
        : RecyclerView.Adapter<PhotoHolder>() {

        override fun onCreateViewHolder(
            parent: ViewGroup,
            viewType: Int
        ): PhotoHolder {
            val view = layoutInflater.inflate(
                R.layout.list_item_gallery,
                parent,
                false
            ) as ImageView
            return PhotoHolder(view)
        }

        override fun getItemCount(): Int = galleryItems.size

        override fun onBindViewHolder(holder: PhotoHolder, position: Int) {
            val galleryItem = galleryItems[position]
            val placeholder: Drawable = ContextCompat.getDrawable(
                requireContext(),
                R.drawable.placeholder
            ) ?: ColorDrawable()
            holder.bindDrawable(placeholder)
            thumbnailDownloader.queueThumbnail(holder, galleryItem.url)
        }
    }

    companion object {
        fun newInstance() = PhotoGalleryFragment()
    }
}
```

Finally, comment in the code to queue the image URL