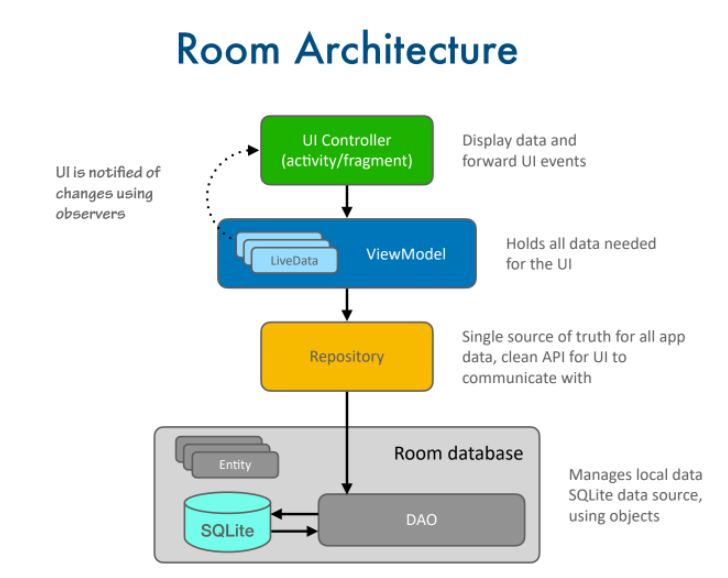
**Final Exam Review**

* In Activities, view components are initialized in onCreate, but in Fragments, they are initialized in onCreateView
* By convention, fragments have a newInstance function in their companion object. The hosting activity uses this to attach the fragment (using the fragment manager)
* Give title a width of 0dp (fill space)
* \_ is a valid variable name. It is used by convention in lambda expressions when the parameter won't be used in the implementation
* Some listeners, like TextWatcher and onCheckChangedListener, are triggered by not only when users interact with them, but also when view state is restored (as on rotation).
* View state is restored after onCreateView but before onStart
* If you set up these listeners in onCreateView, they will be triggered during creation and rotation. We only want them to be triggered by user interaction, so we set them up in onStart
* Other listeners, such as onClickListener, only react to user interaction, so they can be set up before view state is restored
* Adapter -> We don't reflect it here, but getItemCount is called frequently to check if items have been added or removed
* Recall that when the recycler view has enough view holders, it will swap out crime objects as necessary
* A layout (typically Linear or Grid) must always be specified when a recycler view is created
* updateUI:
  + get crimes from view-model
  + use crimes to create adapter
  + assoc. adapter with recycler-view
* Room is a Jetpack architecture component library that simplifies database setup and access. It allows you to define your database structure and queries using annotated Kotlin classes.
* Room is composed of:
  + An API – contains classes you extend to define your database and to build an instance of it
  + Annotations – indicate things like which classes need to be stored in the database, which class represents your database, and which class specifies the accessor functions to your database tables
  + A compiler – processes the annotated classes and generates the implementation of your database
  + Entity – Annotated class that describes a database table when working with Room
  + SQLite database – On device storage. The Room persistence library creates and maintains this database for you
  + DAO – Data access object. A mapping of SQL queries to functions. When you use a DAO, you call the methods, and Room takes care of the rest.
  + Room database – Simplifies database work and serves as an access point to the underlying SQLite database (hides SQLiteOpenHelper). The Room database uses the DAO to issue queries to the SQLite database.
  + Repository – A class you create that is primarily used to manage multiple data sources.
  + ViewModel – Acts as a communication center between the Repository (data) and the UI. The UI no longer needs to worry about the origin of the data. ViewModel instances survive Activity or Fragment recreation.
  + LiveData – A data holder class that can be observed. Always holds or caches the latest version of data, and notifies its observers when data has changed. LiveData is lifecycle aware, so that UI components do not need to start or stop observation – LiveData automatically manages this. 
* @TypeConverter functions convert to and from model objects and database
* typesDAO classes are used for database access (not database creation)
* An @Entity class \*must\* have exactly one field annotated as the @PrimaryKey
* SQLite recognizes a very limited number of types: integer, real, text, and blob for everything else
* If an entity field is not a String or primitive type, use a type converter to convert it
* @Dao interfaces map SQL commands, like queries and inserts, to Kotlin functions
* Queries require SQL code; inserts, updates, and deletes do not
* The @Database annotation tells Room which entities are associated with the database you want to create
* LiveData only notifies active observers about updates. Inactive observers registered to watch LiveData objects aren't notified about changes.
* Advantages of LiveData:
  + Observer pattern
    - Ensures UI matches your data state
  + Lifecycle aware
    - Stopped activities/fragments do not get notified
    - Activities/fragments that resume get notified right away
  + Data passing between threads
    - The main thread handles the UI, while database access must be done using a background thread
    - LiveData passes data between those threads automatically
* The executor creates a new background thread for database manipulation
  + executor = Executors.newSingleThreadExecutor()
* Query functions do not need an executor since LiveData passes data between threads
* Using viewLifecycleOwner instead of "this" prevents problems with rotation and problems when other UI components observe the same data
* Device File Explorer (View → Tool Windows → Device File Explorer)
  + Allows you to see the files on the device
  + Databases are stored in data/data/[package-name]/databases
* LiveData ensures that database access runs on a background thread
* A live data transformation is a way to set up a trigger-response relationship between two LiveData objects. A transformation function takes two inputs:
  + A LiveData object used as a trigger and
  + A mapping function that must return a LiveData object.
* crimeIdLiveData will serve as a trigger
  + crimeIdLiveData = MutableLiveData<UUID>()
* We need to save the crime to the database before we return to the list-view
* Retrofit translates each item in JSON photo array into a GalleryItem

Diagram

Description automatically generated

* Different approach than CriminalIntent to checking if a fragment is already hosted: checks to see if bundle is null.
  + If null => this is a fresh launch of the activity
  + If not null => activity is being reconstructed after a system-initiated destruction (such as rotation or process death)
* @GET does not need a string because @Url tell Retrofit to use the url string
* ResponseBody is a type from the OkHttp library that holds the body of a raw HTTP response
* Unlike in BNR, FlickrFetchr here is an object (singleton)
* @WorkerThread says this function should only be called on a background thread.
* "execute" - instead of "enqueue" - means the request is synchronous.
* Fetch the bytes from the given URL and decode them into a Bitmap
* 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
    - Handler threads only have one Looper and one MessageQueue. In our code, we will not access these objects directly
  + 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
* 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
* H is the object that identifies each download. In our case it will be the PhotoHolder, because the image will be bound to the holder
* onThumbnailDownloaded:
  + This function will run in the UI thread
  + Its implementation will bind the bitmap image to the holder H
* setup will be called when the fragment's onCreate method is called

@OnLifecycleEvent(Lifecycle.Event.ON\_CREATE)

fun setup() {

Log.i(TAG, "Starting background thread")

start()

looper

}

* "start" starts this handler thread; "looper" starts the looper
* tearDown is called when the onDestroy method is called
* A fragment's view lifecycle is shorter than the fragment's – see viewLifecycleOwner
* onLooperPrepared is called once before the looper starts, so it's a good place to setup the handler
* In general, avoid retaining fragments;