Dagger advanced

You should use ***constructor injection*** the most because it allows you to set all the dependencies for an object when you create it. This is the type of injection you used for the `NewsDetailPresenterImpl` class.

Sometimes this isn't possible because you don't have direct control over the creation of the instance of a class. This is the case of classes like `Activity` and `Fragment` whose lifecycle is the responsibility of the Android environment. In this case, you use the ***field injection***, which injects dependencies into a class field, like so:

```
@Component(modules = [AppModule::class]) // 1
interface AppComponent {

  fun inject(frag: NewsListFragment) // 2

  fun inject(frag: NewsDetailFragment)
}
```

There are two important things to note here:

1. Each component can only provide dependencies from the modules assigned to it.
2. You need to declare each target injection class as a function with the parameter of the type of that class. In this case, the `inject()` functions represent this, but you can essentially name them whatever you want. This is just the convention.

If we annotate some provides method as singelton we also need to annotate component as singelton

```
  @Provides
  @Singleton // HERE
  fun provideRepository(): NewsRepository = MemoryNewsRepository()
}
```

You'll also need to import the related package at the beginning of the class. If you try building the app you'll get the following error:

```
e: ...AppComponent.java:7: error: [Dagger/IncompatiblyScopedBindings]
com.raywenderlich.rwnews.di.AppComponent (unscoped) may not reference scoped
bindings:
```

COPY

This happens because you define the `NewsRepository` implementation in the `AppModule` class, which contains information that the `AppComponent` uses to create the instances of its dependency graphs. If you annotate its `@Provides` method with `@Singleton` you assign it a scope that the `@Component` must know to understand when to create its instance.

Head over to the `AppComponent` and add `@Singleton` under `@Component`:

```
@Component(modules = [AppModule::class])
@Singleton // HERE
interface AppComponent {
```

COPY

WOW

This happens because even though you've marked the `@Component` and the provider as `@Singleton`, it doesn't mean that Dagger will create only one instance of the class.

Managing the @Component Lifecycle

What does it mean to bind the lifecycle of an object to one of the `@Component`s that manages it? It means that if you want a single instance of the `MemoryNewsRepository`, you need to have a single instance of the `AppComponent`, reusing it throughout your app.
The Android solution is to create a custom implementation of the `Application` and store the component within.

Create a new class named **InitApp** in `rwnews` and add the following code:

```
class InitApp : Application() {

  // 1
  private lateinit var appComponent: AppComponent

  override fun onCreate() {
    super.onCreate()
    // 2
    appComponent = DaggerAppComponent.create()
  }

  // 3
  fun appComp() = appComponent
}
```

COPY

```
class NewsListFragment : Fragment(), NewsListView {

  @Inject
  lateinit var newsListPresenter: NewsListPresenter
  - - -
  override fun onAttach(context: Context) {
    (context.applicationContext as InitApp) // HERE
      .appComp().inject(this)
    super.onAttach(context)
  }
  - - -
}
```

COPY

Here you're getting the `applicationContext` and casting it to `InitApp` to call `appComp()` which returns the reference to the single `AppComponent` within `InitApp`.

Using @Binds

The current app works, but you can apply optimizations to reduce the code generation time, as well as the quantity of the generated code. You can rely on `@Binds` instead of `@Provides` to do this.

Open the **AppModule.kt** and look at the following definition:

```
@Module
class AppModule {

  - - -
  @Provides
  @Singleton
  fun provideRepository(): NewsRepository = MemoryNewsRepository()
}
```

`provideRepository()` informs Dagger that the implementation to use for the `NewsRepository` is `MemoryNewsRepository`. You're doing some work here because you're the one that's creating the instance, and defining the provider *contract*.

You can avoid this, and delegate the creation of the implementation to Dagger, by using `@Binds`. It lets you **bind** a specific interface to the class constructor for the implementation, hence the name.

```
@Module
abstract class NewsRepositoryModule {

  @Binds
  abstract fun provideRepository(repoImpl: MemoryNewsRepository):
NewsRepository
}
```

This is a new **@Module** that tells Dagger the class **bound** to the `NewsRepository` is `MemoryNewsRepository`. You do this by defining an abstract method which accepts a single parameter of the implementation type and has the interface type as return type. Because the method is **abstract** the class is also **abstract**.

Hidden Power of @Binds

The true power of both `@Binds` and static `@Provides` is that Dagger doesn't generate factory classes which wrap those functions. This is the part about generating less code and increasing performance. Not only does it increase performance at build-time, but it also increases the runtime performance because Dagger is no longer allocating extra classes for each component you create.

Finally, by using `@Inject` and `@Binds`, you've abstracted away the creation of the repository dependency. This means that if you change the constructor parameters in the `MemoryNewsRepository`, you won't have to change the provider function. It'll update automatically, because of the `@Inject`.

If you did this for all the dependencies, you could freely update the constructors by adding or removing parameters, changing their order, and you wouldn't have to do the extra work of updating the provider/factory functions.

Providing Parameterized Dependencies

If the `@Module` used by a `@Component` needs a parameter, Dagger doesn't generate `create()`. It instead generates an implementation of the **Builder** pattern which requires you to provide the parameterized dependencies. Change the code in the `InitApp` like this:

```
class InitApp : Application() {

  lateinit var appComponent: AppComponent

  override fun onCreate() {
    super.onCreate()
    appComponent = DaggerAppComponent
      .builder()  // 1
      .appModule(AppModule(MemoryNewsRepository())) // 2
      .build() // 3
  }

  fun appComp() = appComponent
}
```

COPY

Improving Parameterized Dependencies

In the previous code, you created the `MemoryNewsRepository` in the `InitApp` and passed it to the builder of the `AppComponent` encapsulating it within an instance of the `AppModule`.

You can do better and pass only what Dagger really needs: The instance of the `MemoryNewsRepository`. To do this, change the current code in the **AppComponent.kt** to get the following implementation:

```
@Component(modules = [AppModule::class])
interface AppComponent {

  fun inject(frag: NewsListFragment)

  fun inject(frag: NewsDetailFragment)

  // 1
  @Component.Builder
  interface Builder {

    // 2
    @BindsInstance
    fun repository(repo: NewsRepository): Builder

    // 3
    fun build(): AppComponent
  }
}
```

Using the @Component.Factory Interface

Like the Builder, the Factory method is a creational GoF pattern. That means it defines some way of creating instances of a specific class. While the former provides you some methods in to pass what you need, the latter provides you a single method with multiple parameters.

The former defines a `build()` that the latter doesn't need. The same difference happens when you use them with Dagger. To prove this, change the **AppComponent.kt** to the following:

```
@Component(modules = [AppModule::class])
interface AppComponent {

  fun inject(frag: NewsListFragment)

  fun inject(frag: NewsDetailFragment)

  // 1
  @Component.Factory
  interface Factory {
    // 2
    fun repository(@BindsInstance repo: NewsRepository): AppComponent
  }
}
```

Wery important about SCOPE

Managing Multiple @Scopes

Dagger doesn't just generate code, it also allows you to manage dependencies and their lifecycle within the components you define. To do that, it needs to understand how different objects connect and how to provide a way to build the related graph at runtime.

Creating the `MemoryNewsRepository` instance should be Dagger's responsibility, as well as creating `NewsListPresenterImpl` and `NewsDetailPresenterImpl`.

All objects need memory when you create them, but you don't always need to create all of the objects at the same time. Your app always needs an instance of `MemoryNewsRepository`, but you only need an instance of `NewsDetailPresenterImpl` when you open `NewsDetailFragment`.

As such, different objects have different lifecycles. This is what `@Scope` represents.

Creating Your Custom @Scope

Creating a custom `@Scope` is simple; it's similar to `@Singleton`'s code.

Create a file named **FeatureScope.kt** in the `di` package and add the following code:

```
import javax.inject.Scope

@Scope
@MustBeDocumented
@Retention(AnnotationRetention.RUNTIME)
annotation class FeatureScope
```

COPY

Component width different scope

Dependencies Between Differently-Scoped @Components

You can fix the problem easily. If one `@Component` wants to use objects from another using the `dependencies` attribute, a function needs to explicitly expose them.

In this case, add the following definition to `AppComponent`:

```
@Component(modules = [AppModule::class])
@Singleton
interface AppComponent {

  fun repository(): NewsRepository // HERE
}
```

COPY

This function tells `FeatureComponent` how to access `NewsRepository`'s implementation, even with a different scope.

Build and run. The app will work from Dagger's side, but you still have to use `@FeatureComponent` instead of `@AppComponent` to inject dependencies.

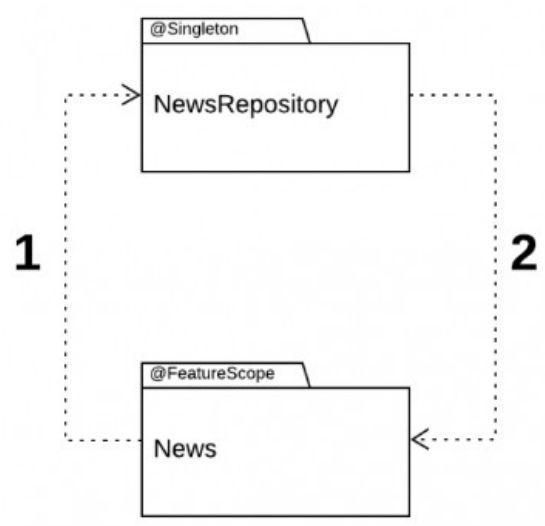
@Component Dependencies Versus @Subcomponent

So far, you've seen two ways of managing dependencies between graph components with different `@Scope`s. But is there a best approach? The answer is: *It depends*. :]

`@Subcomponent`s allow you to write less code because you've exposed all the objects in the dependency graph, not just the ones you publish using specific functions.

On the other hand, each `@Component` needs to know what the `@Subcomponent`s are, which can lead to problems with circular dependencies. For this reason, using the `dependencies` attribute of `@Component` is, at the moment, the only option for apps with multiple modules.

To understand this, consider the following dependency diagram between the module containing the `NewsRepository` implementation and the classes you use for displaying news.



Dependency Cycle

Multibindings - позволяет провайдить не по одному какие-то иожекты а добавлять их в сет

```
@Binds
@IntoSet
fun bindFbAnalycsTracker(fbAnalyticsTracker: FacebookAnalyticsTracker): AnalyticsTracker
```

И в construcor injected мы теперь получаем сет

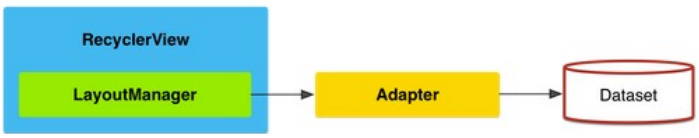
```
class Analytics @Inject constructor(
    private val trackers: Set<@JvmSuppressWildcards AnalyticsTracker>
) {
```

RecyclerView

The `RecyclerView` is a `ViewGroup` that renders any adapter-based view in a similar way. It is supposed to be the successor of ListView and GridView. One of the reasons is that `RecyclerView` has a more extensible framework, especially since it provides the ability to implement both horizontal and vertical layouts. Use the `RecyclerView` widget when you have data collections whose elements change at runtime based on user action or network events.

If you want to use a `RecyclerView`, you will need to work with the following:

- `RecyclerView.Adapter` - To handle the data collection and bind it to the view
- `LayoutManager` - Helps in positioning the items
- `ItemAnimator` - Helps with animating the items for common operations such as Addition or Removal of item



```java
// Create the basic adapter extending from RecyclerView.Adapter
// Note that we specify the custom ViewHolder which gives us access to our views
public class ContactsAdapter extends
    RecyclerView.Adapter<ContactsAdapter.ViewHolder> {

    // Provide a direct reference to each of the views within a data item
    // Used to cache the views within the item layout for fast access
    public class ViewHolder extends RecyclerView.ViewHolder {
        // Your holder should contain a member variable
        // for any view that will be set as you render a row
        public TextView nameTextView;
        public Button messageButton;

        // We also create a constructor that accepts the entire item row
        // and does the view lookups to find each subview
        public ViewHolder(View itemView) {
            // Stores the itemView in a public final member variable that can be used
            // to access the context from any ViewHolder instance.
            super(itemView);

            nameTextView = (TextView) itemView.findViewById(R.id.contact_name);
            messageButton = (Button) itemView.findViewById(R.id.message_button);
        }
    }
}
```

```java
// Create the basic adapter extending from RecyclerView.Adapter
// Note that we specify the custom ViewHolder which gives us access to our views
class ContactsAdapter : RecyclerView.Adapter<ContactsAdapter.ViewHolder>() {

    // Provide a direct reference to each of the views within a data item
    // Used to cache the views within the item layout for fast access
    inner class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        // Your holder should contain and initialize a member variable
        // for any view that will be set as you render a row
        val nameTextView = itemView.findViewById<TextView>(R.id.contact_name)
        val messageButton = itemView.findViewById<Button>(R.id.message_button)
    }
}
```

The **`onBindViewHolder()`**function is called by `RecyclerView` to display the data for one list item at the specified position. So the `onBindViewHolder()` method takes two arguments: a view holder, and a position of the data to bind. For this app, the holder is the `TextItemViewHolder`, and the position is the position in the list.

`RecyclerView` does only the work necessary to process or draw items that are currently visible on the screen.

- When an item scrolls off the screen, its views are recycled. That means the item is filled with new content that scrolls onto the screen.
- The adapter pattern in software engineering helps an object work together with another API. `RecyclerView` uses an adapter to transform app data into something it can display, without the need for changing how the app stores and processes data.

To display your data in a `RecyclerView`, you need the following parts:

- **RecyclerView** To create an instance of `RecyclerView`, define a `<RecyclerView>` element in the layout file.
- **LayoutManager** A `RecyclerView` uses a `LayoutManager` to organize the layout of the items in the `RecyclerView`, such as laying them out in a grid or in a linear list.

In the `<RecyclerView>` in the layout file, set the `app:layoutManager` attribute to the layout manager (such as `LinearLayoutManager` or `GridLayoutManager`).

You can also set the `LayoutManager` for a `RecyclerView` programmatically. (This technique is covered in a later codelab.)

- **Layout for each item** Create a layout for one item of data in an XML layout file.
- **Adapter** Create an adapter that prepares the data and how it will be displayed in a `ViewHolder`. Associate the adapter with the `RecyclerView`.

When `RecyclerView` runs, it will use the adapter to figure out how to display the data on the screen.

The adapter requires you to implement the following methods: – `getItemCount()` to return the number of items. – `onCreateViewHolder()` to return the `ViewHolder` for an item in the list. – `onBindViewHolder()` to adapt the data to the views for an item in the list.

- **ViewHolder** A `ViewHolder` contains the view information for displaying one item from the item's layout.
- The `onBindViewHolder()` method in the adapter adapts the data to the views. You always override this method. Typically, `onBindViewHolder()` inflates the layout for an item, and puts the data in the views in the layout.
- Because the `RecyclerView` knows nothing about the data, the `Adapter` needs to inform the `RecyclerView` when that data changes. Use `notifyDataSetChanged()`to notify the `Adapter` that the data has changed.

DiffUtil

`RecyclerView` has a class called **`DiffUtil`** which is for calculating the differences between two lists. `DiffUtil` takes an old list and a new list and figures out what's different. It finds items that were added, removed, or changed. Then it uses an algorithm called [Eugene W. Myers's difference algorithm] to figure out the minimum number of changes to make from the old list to produce the new list.

Once `DiffUtil` figures out what has changed, `RecyclerView` can use that information to update only the items that were changed, added, removed, or moved, which is much more efficient than redoing the entire list.

```
class SleepNightDiffCallback : DiffUtil.ItemCallback<SleepNight>() {
}
```

3. Put the cursor in the `SleepNightDiffCallback` class name.
4. Press `Alt+Enter` (`Option+Enter` on Mac) and select **Implement Members**.
5. In the dialog that opens, shift-left-click to select the `areItemsTheSame()` and `areContentsTheSame()` methods, then click **OK**.

This generates stubs inside `SleepNightDiffCallback` for the two methods, as shown below. `DiffUtil` uses these two methods to figure out how the list and items have changed.

```
    override fun areItemsTheSame(oldItem: SleepNight, newItem: SleepNight): Boolean {
        TODO("not implemented") //To change body of created functions use File | Setting
    }

    override fun areContentsTheSame(oldItem: SleepNight, newItem: SleepNight): Boolean {
        TODO("not implemented") //To change body of created functions use File | Setting
    }
```

Header/Another item in Recyclerview

Two ways of adding headers

1) One way to add headers to a list is to modify your adapter to use a different `ViewHolder` by checking indexes where your header needs to be shown. The `Adapter` will be responsible for keeping track of the header. For example, to show a header at the top of the table, you need to return a different `ViewHolder` for the header while laying out the zero-indexed item. Then all the other items would be mapped with the header offset, as shown below.

2) Another way to add headers is to modify the backing dataset for your data grid. Since all the data that needs to be displayed is stored in a list, you can modify the list to include items to represent a header. This is a bit simpler to understand, but it requires you to think about how you design your objects, so you can combine the different item types into a single list. Implemented this way, the adapter will display the items passed to it.

Glide

4. Inside the `let {}` block, add the line shown below to convert the URL string (from the XML) to a `Uri` object. Import `androidx.core.net.toUri` when requested.

You want the final `Uri` object to use the HTTPS scheme, because the server you pull the images from requires that secure scheme. To use the HTTPS scheme, append `buildUpon.scheme("https")` to the `toUri` builder. The `toUri()` method is a Kotlin extension function from the Android KTX core library, so it just looks like it's part of the `String` class.

```
val imgUri = imgUrl.toUri().buildUpon().scheme("https").build()
```

5. Still inside `let {}`, call `Glide.with()` to load the image from the `Uri` object into the `ImageView`. Import `com.bumptech.glide.Glide` when requested.

```
Glide.with(imgView.context)
        .load(imgUri)
        .into(imgView)
```

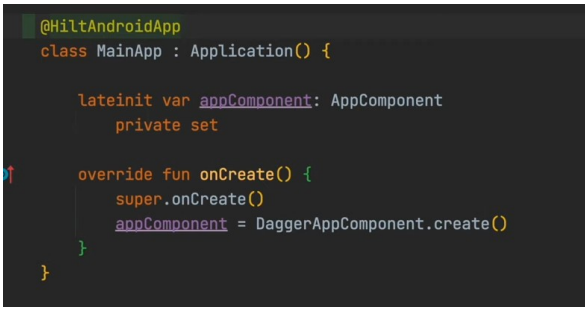
BindingAdapter

Благодаря библиотеке привязки данных почти все вызовы пользовательского интерфейса выполняются в статических методах, называемых адаптерами привязки.

В привязке данных нет магии. Все решается во время компиляции и доступно для чтения в сгенерированном коде.

Hilt

First of all we have to have Application singelton where our instance of Hilt



```
@HiltAndroidApp
class MainApp : Application() {

    lateinit var appComponent: AppComponent
        private set

    override fun onCreate() {
        super.onCreate()
        appComponent = DaggerAppComponent.create()
    }
}
```

We do not need to inject field explicit. We can annotate oue Application or Fragment width `@AndroidEntryPoint`

We have alredy created Scopes

Component	Injector for
SingletonComponent	Application
ViewModelComponent	ViewModel
ActivityComponent	Activity
FragmentComponent	Fragment
ViewComponent	View
ViewWithFragmentComponent	View with @WithFragmentBindings
ServiceComponent	Service

Now module dedecate what component it shuld be added width annotation @InstallIn()

```
@InstallIn(SingletonComponent::class)
@Module
interface AppBindModule {
```

Passing `Context` to a Dagger component using `@BindsInstance` is a common pattern. This is not needed in Hilt as `Context` is **already available** as a predefined binding.

`Context` is usually needed to access resources, databases, shared preferences, and etc. Hilt simplifies injecting to context by using the Qualifier `@ApplicationContext` and `@ActivityContext`.

While migrating your app, check which types require `Context` as a dependency and replace them with the ones Hilt provides.

In this case, `SharedPreferencesStorage` has `Context` as a dependency. In order to tell Hilt to inject the context, open `SharedPreferencesStorage.kt. SharedPreferences` requires application's `Context`, so add `@ApplicationContext` annotation to the context parameter.

SharedPreferencesStorage.kt

```
class SharedPreferencesStorage @Inject constructor(
    @ApplicationContext context: Context
) : Storage {
```

For each Android class that can be injected by Hilt, there's an associated Hilt component. For example, the `Application` container is associated with `SingletonComponent`, and the `Fragment` container is associated with `FragmentComponent`.

```
@InstallIn(SingletonComponent::class)
@Module
```

Each Hilt container comes with a set of default bindings that can be injected as dependencies into your custom bindings. This is the case with `applicationContext`. To access it, you need to annotate the field with **`@ApplicationContext.`**

```
fun provideDatabase(@ApplicationContext appContext: Context): AppDataba
```

To provide different implementations (multiple bindings) of the same type, you can use qualifiers.

```
@Qualifier
annotation class InMemoryLogger

@Qualifier
annotation class DatabaseLogger

@InstallIn(SingletonComponent::class)
@Module
abstract class LoggingDatabaseModule {

    @DatabaseLogger
    @Singleton
    @Binds
    abstract fun bindDatabaseLogger(impl: LoggerLocalDataSource): LoggerDataSource
}
```

@EntryPoint annotation

`@EntryPoint` annotation which is used to **inject dependencies in classes not supported by Hilt**.

An entry point is an interface with an accessor method for each binding type we want (including its qualifier). Also, the interface must be annotated with `@InstallIn` to specify the component in which to install the entry point.

The best practice is adding the new entry point interface inside the class that uses it. Therefore, include the interface in `LogsContentProvider.kt` file:

```
class LogsContentProvider: ContentProvider() {

    @InstallIn(SingletonComponent::class)
    @EntryPoint
    interface LogsContentProviderEntryPoint {
        fun logDao(): LogDao
    }

    ...
}
```

RxJava

Observable – Источник данных

```
Observable<String> myObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> sub) {
            sub.onNext("Hello, world!"); // порождает данные
            sub.onCompleted();
        }
    }
);
```

Subscriber – потребитель

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) {
System.out.println(s); // обрабатывает данные полученные из
 }

    @Override
    public void  onCompleted() { }

    @Override
    public void onError(Throwable e) { }
};
```

`myObservable.subscribe(mySubscriber);` - подписываемся на источник

вызывается **onNext потом onCompleted**

Упростим код :

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
              System.out.println(s);
        }
    });
```

Упростим ещё код

```
Observable.just("Hello, world!")
    .subscribe(s -> System.out.println(s));
```

операторы

map()

Интересным свойством `map()` является то, что он не обязан порождать данные того же самого типа, что и исходный `Observable`.
Допустим, что наш `Subscriber` должен выводить не порождаемый текст, а его хэш:

```
Observable.just("Hello, world!")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
            return s.hashCode();
        }
    })
    .subscribe(i -> System.out.println(Integer.toString(i)));
```

С лямбдами

```
Observable.just("Hello, world!")
    .map(s -> s.hashCode())
    .subscribe(i -> System.out.println(Integer.toString(i)));
```

Observable.from()

Берёт коллекцию и «испускает» один элемент этой коллекции за другим:

```
Observable.from("url1", "url2", "url3")
    .subscribe(url -> System.out.println(url));
```

Observable.flatMap()

`Observable.flatMap()` принимает на вход данные, излучаемые одним `Observable`, и возвращает данные, излучаемые другим `Observable`, подменяя таким образом один `Observable` на друго

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .subscribe(url -> System.out.println(url));
```

Новый `Observable` — это то, что увидит в итоге наш `Subscriber`. Он не получит `List<String>`, он получит поток индивидуальных объектов класса `String` так, как он получил бы от `Observable.from()`

`flatMap` можно комбинировать столько сколько захочешь)

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .subscribe(title -> System.out.println(title));
```

filter()

Испускает тот же самый элемент потока данных, который он получил, но только если он проходит проверку.

```
query("Hello, world!")
      .filter(title -> title != null) // if(title != null)
```

take()

Возвращает не больше заданного количества элементов

```
query("Hello, world!")
    .take(5)
```

doOnNext()

Позволяет нам добавить некоторое дополнительное действие, происходящее всякий раз, как мы получаем новый элемент данных

```
query("Hello, world!")
 .doOnNext(title -> saveTitle(title))
```

subscribeOn(Schedulers.io())

Определяет в каком потоке будет выполняться Observer

observeOn(AndroidSchedulers.mainThread())

Определяет в каком потоке будет обробатываться результат выполнения Observer

Когда вы вызываете Observable.subscribe(), вам в ответ возвращается объект класса Subscription

```
Subscription subscription = Observable.just("Hello, World!")
    .subscribe(s -> System.out.println(s));
```

В дальнейшем можно использовать полученный нами Subscription для того, чтобы прекратить подписку:

```
subscription.unsubscribe();
```

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) {
System.out.println(s); // обрабатывает данные полученные из
 }

    @Override
    public void  onCompleted() { }
```

```
    @Override
    public void onError(Throwable e) { }
};
```

`onError()` вызывается вне зависимости от того, *когда* было выброшено исключение

RxAndroid

RxAndroid is an extension to RxJava built just for Android. It includes special bindings that will make your life easier.

First, there's `AndroidSchedulers` which provides schedulers ready-made for Android's threading system. Need to run some code on the UI thread? No problem - just use `AndroidSchedulers.mainThread()`:

```
retrofitService.getImage(url)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(bitmap -> myImageView.setImageBitmap(bitmap));
```

Next we have `AndroidObservable` which provides more facilities for working within the Android lifecycle. There is `bindActivity()` and `bindFragment()` which, in addition to automatically using `AndroidSchedulers.mainThread()` for observing, will also stop emitting items when your `Activity` or `Fragment` is finishing

```
AndroidObservable.bindActivity(this, retrofitService.getImage(url))
    .subscribeOn(Schedulers.io())
    .subscribe(bitmap -> myImageView.setImageBitmap(bitmap));
```

`AndroidObservable.fromBroadcast()`, which allows you to create an `Observable` that works like a `BroadcastReceiver`. Here's a way to be notified whenever network connectivity changes:

```
IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
AndroidObservable.fromBroadcast(context, filter)
    .subscribe(intent -> handleConnectivityChange(intent));
```

Finally, there is `ViewObservable`, which adds a couple bindings for `Views`. There's `ViewObservable.clicks()` if you want to get an event each time a `View` is clicked, or `ViewObservable.text()` to observe whenever a `TextView` changes its content.

```
ViewObservable.clicks(mCardNameEditText, false)
    .subscribe(view -> handleClick(view));
```

Retrofit

With RxJava installed, you can have it return an `Observable` instead:

```
@GET("/user/{id}/photo")
Observable<Photo> getUserPhoto(@Path("id") int id);
```

zip()

```
Observable.zip(
    service.getUserPhoto(id),
    service.getPhotoMetadata(id),
    (photo, metadata) -> createPhotoWithData(photo, metadata))
    .subscribe(photoWithData -> showPhoto(photoWithData));
```

Lifecycle

There are two issues that crop up over and over again:

1. Continuing a `Subscription` during a configuration change (e.g. rotation).

   Suppose you make REST call with Retrofit and then want to display the outcome in a `ListView`. What if the user rotates the screen? You want to continue the same request, but how?

2. Memory leaks caused by `Observables` which retain a copy of the `Context`.

   This problem is caused by creating a subscription that retains the `Context` somehow, which is not difficult when you're interacting with `Views`! If `Observable` doesn't complete on time, you may end up retaining a lot of extra memory.

The first problem can be solved with some of RxJava's built-in caching mechanisms, so that you can unsubscribe/resubscribe to the same `Observable` without it duplicating its work. In particular, `cache()` (or `replay()`

```
Observable<Photo> request = service.getUserPhoto(id).cache();
Subscription sub = request.subscribe(photo -> handleUserPhoto(photo));

// ...When the Activity is being recreated...
sub.unsubscribe();

// ...Once the Activity is recreated...
request.subscribe(photo -> handleUserPhoto(photo));
```

The second problem can be solved by properly unsubscribing from your subscriptions in accordance with the lifecycle. It's a common pattern to use a `CompositeSubscription` to hold all of your `Subscriptions`, and then unsubscribe all at once in `onDestroy()` or `onDestroyView()`

```
private CompositeSubscription mCompositeSubscription
    = new CompositeSubscription();

private void doSomething() {
    mCompositeSubscription.add(
                AndroidObservable.bindActivity(this, Observable.just("Hello, World
        .subscribe(s -> System.out.println(s)));
}

@Override
protected void onDestroy() {
    super.onDestroy();

    mCompositeSubscription.unsubscribe();
}
```

Backpressure

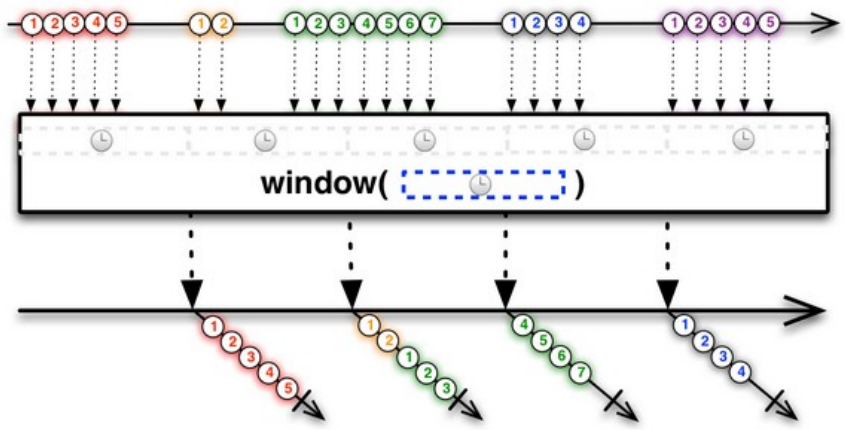
In RxJava it is not difficult to get into a situation in which an Observable is emitting items more rapidly than an operator or subscriber can consume them. This presents the problem of what to do with such a growing backlog of unconsumed items.

A cold Observable emits a particular sequence of items, but can begin emitting this sequence when its Observer finds it to be convenient, and at whatever rate the Observer desires, without disrupting the integrity of the sequence. For example if you convert a static Iterable into an Observable, that Observable will emit the same sequence of items no matter when it is later subscribed to or how frequently those items are observed. Examples of items emitted by a cold Observable might include the results of a database query, file retrieval, or web request.

A hot Observable begins generating items to emit immediately when it is created. Subscribers typically begin observing the sequence of items emitted by a hot Observable from somewhere in the middle of the sequence, beginning with the first item emitted by the Observable subsequent to the establishment of the subscription. Such an Observable emits items at its own pace, and it is up to its observers to keep up. Examples of items emitted by a hot Observable might include mouse & keyboard events, system events, or stock prices.

window

`window` is similar to `buffer`. One variant of `window` allows you to periodically emit Observable windows of items at a regular interval of time:



Drawables (class)

When you need to display static images in your app, you can use the `Drawable` class and its subclasses to draw shapes and images. A `Drawable` is a general abstraction for *something that can be drawn*. The various subclasses help with specific image scenarios, and you can extend them to define your own drawable objects that behave in unique ways.

There are two ways to define and instantiate a `Drawable` besides using the class constructors:

- Inflate an image resource (a bitmap file) saved in your project.
- Inflate an XML resource that defines the drawable properties.

And dont forget about Ondraw(Canwas) and inside the canvas we can create any graphical primitives

Vector drawables

A `VectorDrawable` is a vector graphic defined in an XML file as a set of points, lines, and curves along with its associated color information. The major advantage of using a vector drawable is image scalability.

AnimatedVectorDrawable

It consist three types of file :

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
   android:height="64dp"
   android:width="64dp"
   android:viewportHeight="600"
   android:viewportWidth="600" >
   <group
      android:name="rotationGroup"
      android:pivotX="300.0"
      android:pivotY="300.0"
      android:rotation="45.0" >
      <path
         android:name="vectorPath"
         android:fillColor="#000000"
         android:pathData="M300,70 l 0,-70 70,70 0,0 -70,70z" />
   </group>
</vector>
```

- AnimatedVectorDrawable's XML file: `avd.xml`

```
<animated-vector xmlns:android="http://schemas.android.com/apk/res/android"
   android:drawable="@drawable/vd" >
     <target
         android:name="rotationGroup"
         android:animation="@anim/rotation" />
     <target
         android:name="vectorPath"
         android:animation="@anim/path_morph" />
</animated-vector>
```

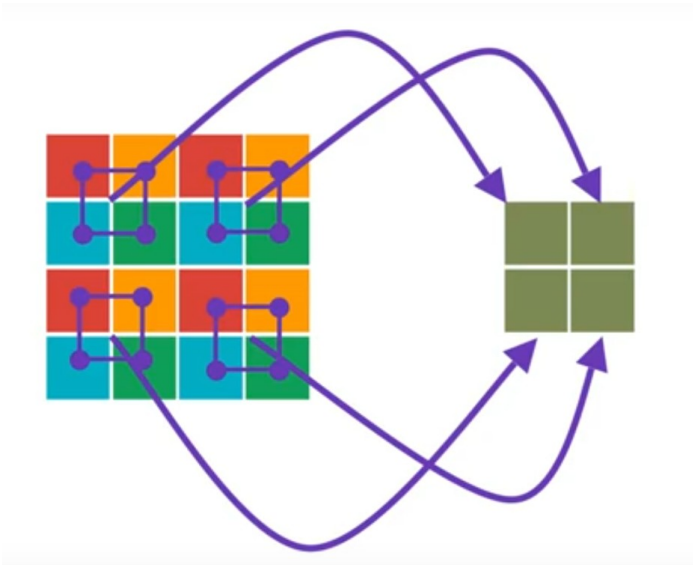
- Animator XML files that are used in the AnimatedVectorDrawable's XML file: `rotation.xml` and `path_morph.xml`

```
<objectAnimator
   android:duration="6000"
   android:propertyName="rotation"
   android:valueFrom="0"
   android:valueTo="360" />
```

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
   <objectAnimator
      android:duration="3000"
      android:propertyName="pathData"
      android:valueFrom="M300,70 l 0,-70 70,70 0,0   -70,70z"
      android:valueTo="M300,70 l 0,-70 70,0  0,140 -70,0 z"
      android:valueType="pathType"/>
</set>
```

Scailed bitmap

Если мы показываем bitmap сжатым но мы всё равно храним все данные про него и это занимает много памяти для того чтобы этого не случалось мы можем создать Scailed bitmap который при отоброжании в меньшем екране будет хранить меньше значения



AVIF

Android 12 (API level 31) and higher support images that use the AV1 Image File Format (AVIF). AVIF is a container format for images and sequences of images encoded using AV1. AVIF takes advantage of the intra-frame encoded content from video compression. This dramatically improves image quality for the same file size when compared to older image formats, such as JPEG.

JPG

If you are using JPG images, there are several small changes you can make that potentially provide significant file-size savings. These include:

- Producing a smaller file size through different encoding methods (without impacting quality).
- Adjusting quality slightly in order to yield better compression.

Pursuing these strategies can often net you file-size reductions of up to 25%.

WebP

WebP is a newer image format supported from Android 4.2.1 (API level 17). This format provides superior lossless and lossy compression for images on the web. Using WebP, developers can create smaller, richer images. WebP lossless image files are, on average, 26% smaller than PNGs. These image files also support transparency (also known as alpha channel) at a cost of just 22% more bytes.

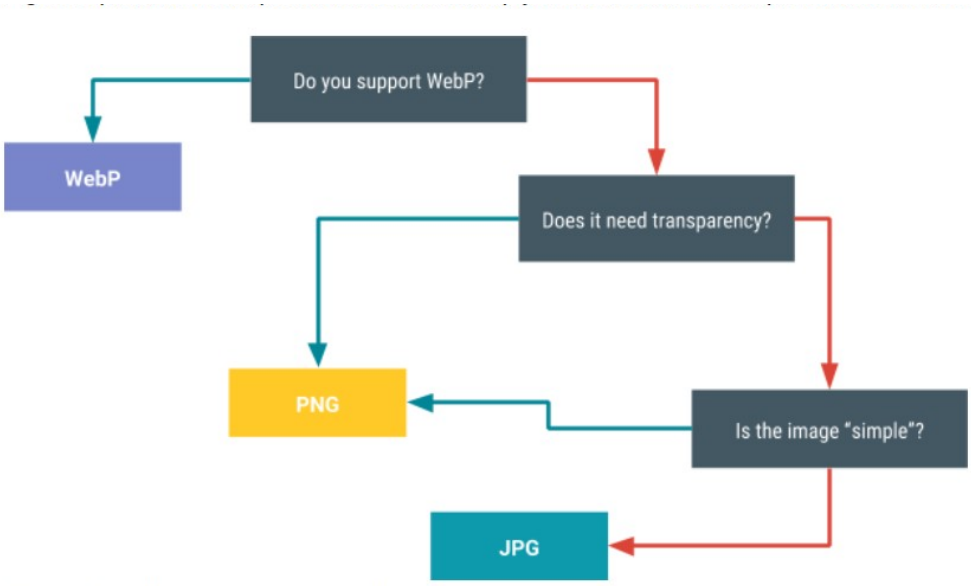


Figure 6. Deciding on a compression scheme

Executor

First of all in our Application class we create Thread pull for executor

```
class MyApplication : Application() {
    val executorService: ExecutorService = Executors.newFixedThreadPool(4)
}
```

The Executor's `execute()` method takes a `Runnable`. A `Runnable` is a Single Abstract Method (SAM) interface with a `run()` method that is executed in a thread when invoked.

```
executor.execute {
    val ignoredResponse = makeSynchronousLoginRequest(url, jsonBody)
}
```

Communicating with the main thread

For such target we can use Callback width resoult of our execte

```
fun makeLoginRequest(
    jsonBody: String,
    callback: (Result<LoginResponse>) -> Unit
) {
    executor.execute {
        try {
            val response = makeSynchronousLoginRequest(jsonBody)
            callback(response)
        } catch (e: Exception) {
            val errorResult = Result.Error(e)
            callback(errorResult)
        }
    }
}
```

And to receive resault

```
fun makeLoginRequest(username: String, token: String) {
    val jsonBody = "{ username: \"$username\", token: \"$token\"}"
    loginRepository.makeLoginRequest(jsonBody) { result ->
        when(result) {
            is Result.Success<LoginResponse> -> // Happy path
            else -> // Show error in UI
        }
    }
}
```

Using handlers

You can use a `Handler` to enqueue an action to be performed on a different thread. To specify the thread on which to run the action, construct the `Handler` using a `Looper` for the thread. A `Looper` is an object that runs the message loop for an associated thread. Once you've created a `Handler`, you can then use the `post(Runnable)` method to run a block of code in the corresponding thread.

`Looper` includes a helper function, `getMainLooper()`, which retrieves the `Looper` of the main thread. You can run code in the main thread by using this `Looper` to create a `Handler`. As this is something you might do quite often, you can also save an instance of the `Handler` in the same place you saved the `ExecutorService`:

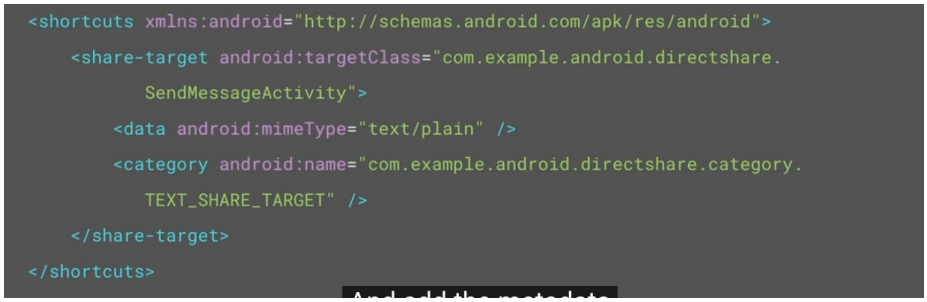
Kotlin Java

```
class MyApplication : Application() {
    val executorService: ExecutorService = Executors.newFixedThreadPool(4)
    val mainThreadHandler: Handler = HandlerCompat.createAsync(Looper.getMainLooper())
}
```

```
executor.execute {
    try {
        val response = makeSynchronousLoginRequest(jsonBody)
        resultHandler.post { callback(response) }
    } catch (e: Exception) {
        val errorResult = Result.Error(e)
        resultHandler.post { callback(errorResult) }
    }
}
```

Android Sharesheet

We can create our castom shortcut and modify it whenewwer we want



```
<shortcuts xmlns:android="http://schemas.android.com/apk/res/android">
    <share-target android:targetClass="com.example.android.directshare.
            SendMessageActivity">
        <data android:mimeType="text/plain" />
        <category android:name="com.example.android.directshare.category.
            TEXT_SHARE_TARGET" />
    </share-target>
</shortcuts>
```

File sharing

Question 1

How does `RecyclerView` display items? Select all that apply.

▢ Displays items in a list or a grid.

▢ Scrolls vertically or horizontally.

▢ Scrolls diagonally on larger devices such as tablets.

▢ Allows custom layouts when a list or a grid is not enough for the use case.

Question 2

What are the benefits of using `RecyclerView`? Select all that apply.

▢ Efficiently displays large lists.

▢ Automatically updates the data.

▢ Minimizes the need for refreshes when an item is updated, deleted, or added to the list.

▢ Reuses view that scrolls off screen to display the next item that scrolls on screen.

Question 3

What are some of the reasons for using adapters? Select all that apply.

▢ Separation of concerns makes it easier to change and test code.

▢ `RecyclerView` is agnostic to the data that is being displayed.

▢ Data processing layers do not have to concern themselves with how data will be displayed.

▢ The app will run faster.

Question 4

Which of the following are true of `ViewHolder`? Select all that apply.

▢ The `ViewHolder` layout is defined in XML layout files.

▢ There is one `ViewHolder` for each unit of data in the dataset.

▢ You can have more than one `ViewHolder` in a `RecyclerView`.

▢ The `Adapter` binds data to the `ViewHolder`.

Question 1

Which of the following are necessary to use `DiffUtil`? Select all that apply.

▢ Extend the `ItemCallBack` class.

▢ Override `areItemsTheSame()`.

▢ Override `areContentsTheSame()`.

▢ Use data binding to track the differences between items.

Question 2

Which of the following are true about binding adapters?

▢ A binding adapter is a function annotated with `@BindingAdapter`.

▢ Using a binding adapter allows you to separate data formatting from the view holder.

▢ You must use a `RecyclerViewAdapter` if you want to use binding adapters.

▢ Binding adapters are a good solution when you need to transform complex data.

Question 3

When should you consider using `Transformations` instead of a binding adapter? Select all that apply.

- ▢ Your data is simple.
- ▢ You are formatting a string.
- ▢ Your list is very long.
- ▢ Your `ViewHolder` only contains one view.

Question 1

Which of the following are layout managers provided by Android? Select all that apply.

- ▢ `LinearLayoutManager`
- ▢ `GridLayoutManager`
- ▢ `CircularLayoutManager`
- ▢ `StaggeredGridLayoutManager`

Question 2

What is a "span"?

- ▢ The size of a grid created by `GridLayoutManager`.
- ▢ The width of a column in the grid.
- ▢ The dimensions of an item in a grid.
- ▢ The number of columns in a grid that has a vertical orientation.

Question 1

What are the two key things Retrofit needs to build a web services API?

- ▢ The base URI for the web service, and a `GET` query.
- ▢ The base URI for the web service, and a converter factory.
- ▢ A network connection to the web service, and an authorization token.
- ▢ A converter factory, and a parser for the response.

Question 2

What is the purpose of the Moshi library?

- ▢ To get data back from a web service.
- ▢ To interact with Retrofit to make a web service request.

▢ To parse a JSON response from a web service into Kotlin data objects.

▢ To rename Kotlin objects to match the keys in the JSON response.

Question 1

Which Glide method do you use to indicate the `ImageView` that will contain the loaded image?

▢ `into()`

▢ `with()`

▢ `imageview()`

▢ `apply()`

Question 2

How do you specify a placeholder image to show when Glide is loading?

▢ Use the `into()` method with a drawable.

▢ Use `RequestOptions()` and call the `placeholder()` method with a drawable.

▢ Assign the `Glide.placeholder` property to a drawable.

▢ Use `RequestOptions()` and call the `loadingImage()` method with a drawable.

Question 3

How do you indicate that a method is a binding adapter?

▢ Call the `setBindingAdapter()` method on the `LiveData`.

▢ Put the method into a Kotlin file called `BindingAdapters.kt`.

▢ Use the `android:adapter` attribute in the XML layout.

▢ Annotate the method with `@BindingAdapter`.

Question 1

What does the `<import>` tag in an XML layout file do?

▢ Include one layout file in another.

▢ Embed Kotlin code inside the layout file.

▢ Provide access to data-bound properties.

▢ Enable you to reference classes and class members in binding expressions.

Question 2

How do you add a query option to a REST web service call in Retrofit?

▢ Append the query to the end of the request URL.

▢ Add a parameter for the query to the function that makes the request, and annotate that parameter with `@Query`.

▢ Use the `Query` class to build a request.

▢ Use the `addQuery()` method in the Retrofit builder.