IMMOBILIEN SCOUT 24

Working with Dagger and Kotlin

Sinan Kozak & Ash Davies

@snnkzk | @askashdavies

Introduction 👋

— Dagger 2 is a fast dependency injector for Android and Java
— One of goals is having compile time safety
— It is written for only Java in mind
— It is used extensively outside of Android ecosystem

Dagger 2 and Kotlin ☕ 🗡️

— Dagger 2 can work with Kotlin
— But generated code is plain Java source code
— Kotlin generated code won't like to happen

Dagger Qualifiers

Qualifiers used to identify dependencies with identical signatures

- Factories use qualifiers to decide the instance use
- Can create your own qualifier annotations, or just use @Named.
- Apply qualifiers by annotating the field or parameter of interest.
- The type and qualifier annotation will both be used to identify the dependency.

Retention Annotation

Use Kotlin retention annotations instead of Java retention

- Java retention support is depreceted in Kotlin
- At least BINARY retention but RUNTIME is ideal
- Dagger 2 doesn’t operate on source files
- Annotations are necessary for kapt

Constructor injection

```
class Game @Inject constructor(
    @Named("P1") private val player1: Player,
    @Named("P2") private val player2: Player
)
```

Constructor injection

```
class Game @Inject constructor(
    @Named("P1") private val player1: Player,
    @Named("P2") private val player2: Player
)
```

```
public final class Game {
   private final Player player1;
   private final Player player2;

   @Inject public Game(
     @Named("P1") Player player1,
     @Named("P2") Player player2) {
      super();
      this.player1 = player1;
      this.player2 = player2;
   }
}
```

Constructor injection

```
class Game @Inject constructor(
    @Named("P1") private val player1: Player,
    @Named("P2") private val player2: Player
)
```

```
public final class Game {
   private final Player player1;
   private final Player player2;

   @Inject public Game(
     @Named("P1") Player player1,
     @Named("P2") Player player2) {
      super();
      this.player1 = player1;
      this.player2 = player2;
   }
}
```

lateinit var 🛑

```
class Game @Inject constructor() {

  @Inject @Named("P1") lateinit var player1: Player
  @Inject @Named("P2") lateinit var player2: Player
}
```

Decompiled lateinit var

```
public final class Game {
    @Inject public Player player1;
    @Inject public Player player2;

    @Named("P1") public static void player1$annotations() {}

    public final Player getPlayer1() { ... }

    public final void setPlayer1(Player var1) {...}

    @Named("P2") public static void player2$annotations() {}

    public final Player getPlayer2() { ... }

    public final void setPlayer2(Player var1) {...}
```

Decompiled lateinit var

```
public final class Game {
    @Inject public Player player1;
    @Inject public Player player2;

    @Named("P1") public static void player1$annotations() {}

    public final Player getPlayer1() { ... }

    public final void setPlayer1(Player var1) {...}

    @Named("P2") public static void player2$annotations() {}

    public final Player getPlayer2() { ... }

    public final void setPlayer2(Player var1) {...}
```

Specify annotation

```
class Game @Inject constructor() {

  @Inject @field:Named("P1") lateinit var player1: Player
  @Inject @field:Named("P2") lateinit var player2: Player
}
```

- We need to specify where annotation needs to apply in Java world
  - @field:...
  - @set:...
  - @get:...
  - @param:...
  - @property:...
  - @setparam:...
  - @receiver:...
  - @delegete:...

Specify annotation

```
class Game @Inject constructor() {

  @Inject @field:Named("P1") lateinit var player1: Player
  @Inject @field:Named("P2") lateinit var player2: Player
}
```

```
public final class Game {

   @Inject @Named("P1") public Player player1;
   @Inject @Named("P2") public Player player2;

   public final Player getPlayer1() {...}

   public final void setPlayer1(Player var1) {...}

   public final Player getPlayer2() {...}

   public final void setPlayer2(Player var1) {...}

}
```

Specify annotation

```
public final class Game1 {
    @Inject @Named("P1") public Player player1;
    @Inject @Named("P2") public Player player2;

    public final Player getPlayer1() {...}

    public final void setPlayer1(Player var1) {...}

    public final Player getPlayer2() {...}

    public final void setPlayer2(Player var1) {...}
}
```

Constructor vs Property injection

- Constructor val
  - Easy to use
  - Safe at runtime if project compile successfully
- Property lateinit var injection
  - Kotlin properties uses property access syntax via accessors
  - Unclear where the annotation is applied, accessor or property
  - Dont forget to use with `@field:`

  ^ lateinit var is error prone and hacky to test

Scope Annotations 🎯

@Scope 🎯

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Scope {
}
```

@Singleton

@Singleton != Singleton Pattern

@Singleton != Singleton Pattern

```
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

@Singleton != Singleton Pattern

```
object Singleton
```

@Scope

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

@Scope

```
@Module
internal object ApplicationModule {

    @Provides
    @JvmStatic
    @ActivityScope
    fun context(application: Application): Context = application
}
```

@ActivityScope 🤷😏

@Scope 🙅

```
@ActivityScope
class ActivityRepository @Inject constructor() {
}
```

@Reusable

Double Check

```
public final class DoubleCheck<T> implements Provider<T>, Lazy<T> {
  private static final Object UNINITIALIZED = new Object();

  private volatile Provider<T> provider;
  private volatile Object instance = UNINITIALIZED;

  private DoubleCheck(Provider<T> provider) { /* ... */ }

  @Override
  public T get() {
    Object result = instance;
    if (result == UNINITIALIZED) {
      synchronized (this) {
        result = instance;
        if (result == UNINITIALIZED) {
          result = provider.get();
          instance = reentrantCheck(instance, result);
          provider = null;
        }
      }
    }
    return (T) result;
  }

  public static Object reentrantCheck(Object currentInstance, Object newInstance) { /* ... */ }
}
```

Double Check

```
public final class DoubleCheck<T> implements Provider<T>, Lazy<T> {
  private static final Object UNINITIALIZED = new Object();

  private volatile Provider<T> provider;
  private volatile Object instance = UNINITIALIZED;

  private DoubleCheck(Provider<T> provider) { /* ... */ }

  @Override
  public T get() {
    Object result = instance;
    if (result == UNINITIALIZED) {
      synchronized (this) {
        result = instance;
        if (result == UNINITIALIZED) {
          result = provider.get();
          instance = reentrantCheck(instance, result);
          provider = null;
        }
      }
    }
    return (T) result;
  }

  public static Object reentrantCheck(Object currentInstance, Object newInstance) { /* ... */ }
}
```

Single Check

```
public final class SingleCheck<T> implements Provider<T> {
  private static final Object UNINITIALIZED = new Object();

  private volatile Provider<T> provider;
  private volatile Object instance = UNINITIALIZED;

  private SingleCheck(Provider<T> provider) { /* ... */ }

  @Override
  public T get() {
    Object local = instance;
    if (local == UNINITIALIZED) {
      Provider<T> providerReference = provider;
      if (providerReference == null) {
        local = instance;
      } else {
        local = providerReference.get();
        instance = local;
        provider = null;
      }
    }
    return (T) local;
  }
}
```

Kotlin: Lazy

```
private val viewModel by lazy(NONE) { SampleViewModel() }

fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> =
    when (mode) {
        LazyThreadSafetyMode.SYNCHRONIZED -> SynchronizedLazyImpl(initializer)
        LazyThreadSafetyMode.PUBLICATION -> SafePublicationLazyImpl(initializer)
        LazyThreadSafetyMode.NONE -> UnsafeLazyImpl(initializer)
    }
```

Favour @Reusable over @Scope 👍

- Great for expensive dependencies
- Work great in single thread environments
- Not guaranteed same instance in multiple threads
- Prefer to keep your Dagger graph stateless
- Use @Scope if you absolutely need to store state

Dagger: Modules

Status Quo 🎸

```
@Module
public abstract class ApplicationModule {

    @Binds
    abstract Context context(Application application);

    @Provides
    static SampleRepository repository(String name) {
        return new SampleRepository(name);
    }
}
```

Dagger: Modules

```
@Module
abstract class ApplicationModule {

    @Binds
    abstract fun context(application: Application): Context

    @Module
    companion object {

        @Provides
        @JvmStatic
        fun repository(name: String): SampleRepository = SampleRepository(name)
    }
}
```

Dagger: Modules

```
public abstract class ApplicationModule {
   public static final ApplicationModule.Companion Companion = new ApplicationModule.Companion();

   @Binds
   @NotNull
   public abstract Context context(@NotNull Application var1);

   @Provides
   @JvmStatic
   @NotNull
   public static final SampleRepository repository(@NotNull String name) {
      return Companion.repository(name);
   }

   @Module
   public static final class Companion {
      @Provides
      @JvmStatic
      @NotNull
      public final SampleRepository repository(@NotNull String name) {
         return new SampleRepository(name);
      }

      private Companion() {
      }
   }
}
```

Dagger: Modules

```
object ApplicationModule {

    @Provides
    @JvmStatic
    fun context(application: Application): Context = application

    @Provides
    @JvmStatic
    fun repository(name: String): SampleRepository = SampleRepository(name)
}
```

Dagger: Modules

```
public final class ApplicationModule {
   public static final ApplicationModule INSTANCE = new ApplicationModule();

   @Provides
   @JvmStatic
   @NotNull
   public static final Context context(@NotNull Application application) {
      return (Context)application;
   }

   @Provides
   @JvmStatic
   @NotNull
   public static final SampleRepository repository(@NotNull String name) {
      return new SampleRepository(name);
   }

   private ApplicationModule() {
   }
}
```

Dagger: Modules

```
@file:JvmName("ApplicationModule")
@file:Module

@Provides
fun context(application: Application): Context = application

@Provides
fun repository(name: String): SampleRepository = SampleRepository(name)
```

Dagger: Modules

```
public final class ApplicationModule {

    @Provides
    @NotNull
    public static final Context context(@NotNull Application application) {
        return (Context)application;
    }

    @Provides
    @NotNull
    public static final SampleRepository repository(@NotNull String name) {
        return new SampleRepository(name);
    }
}
```

Kotlin: Generics<? : T>

Kotlin: Generics<? : T>

Kotlin: Generics<? : T>

Java Interoperability

Kotlin: Generics<? : T>

Java Interoperability

```
interface Collection<E> extends Iterable<E> {

    boolean addAll(Collection<? extends E> collection);
}
```

Kotlin: Generics<? : T>

Java Interoperability

```
interface Collection<E> extends Iterable<E> {

    boolean addAll(Collection<E> collection);
}
```

Kotlin: Generics<? : T>

Java Interoperability

```
List<String> : List<Object>
```

Kotlin: Generics<? : T>

Java Interoperability

~~`List<String> : List<Object>`~~

Kotlin: Generics<? : T>

Java Interoperability

```
List<String> strings = new ArrayList<String>();
List<Object> objs = strings;
objs.add(1);
String string = strings.get(0);
```

Kotlin: Generics<? : T>

Java Interoperability

```
List<String> strings = new ArrayList<String>();
List<Object> objs = strings;
objs.add(1);
String string = strings.get(0); // 🔥🔥🔥
```

Kotlin: Generics<? : T>

Java Interoperability

```
interface Collection<E> extends Iterable<E> {

    boolean addAll(Collection<? extends E> collection);
}
```

Kotlin: Generics<? : T>

Java Interoperability

```
List<String> box(String value) { /* ... */ }

String unbox(List<? extends String> boxed) { /* ... */ }
```

Kotlin: Generics<? : T>

Java Interoperability

```
class ListAdapter @Inject constructor(strings: List<String>)
```

Kotlin: Generics<? : T>

Java Interoperability

```
class ListAdapter @Inject constructor(strings: List<String>)
```

Kotlin: Generics<? : T>

Java Interoperability

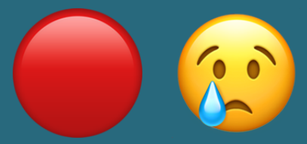
```
public final class ListAdapter {
    @Inject
    public ListAdapter(@NotNull List<? extends String> strings) {
        Intrinsics.checkParameterIsNotNull(strings, "strings");
        super();
    }
}
```

Kotlin: Generics<? : T>

Java Interoperability

```
@Module
object ListModule {

    @IntoSet
    @Provides
    @JvmStatic
    fun hello(): String = "Hello"

    @IntoSet
    @Provides
    @JvmStatic
    fun world(): String = "World"
}
```

Build Failed...



Kotlin: Generics<? : T>

Java Interoperability

```
class ListAdapter @Inject constructor(strings: @JvmSuppressWildcards List<String>)
```

Kotlin: Generics<? : T>

Dagger Multi-Bindings

```
@Module
object ListModule {

    @IntoSet
    @Provides
    @JvmStatic
    fun hello(): String = "Hello"

    @IntoSet
    @Provides
    @JvmStatic
    fun world(): String = "World"
}
```

Jetpack

ViewModel

@snnkzk | @askashdavies

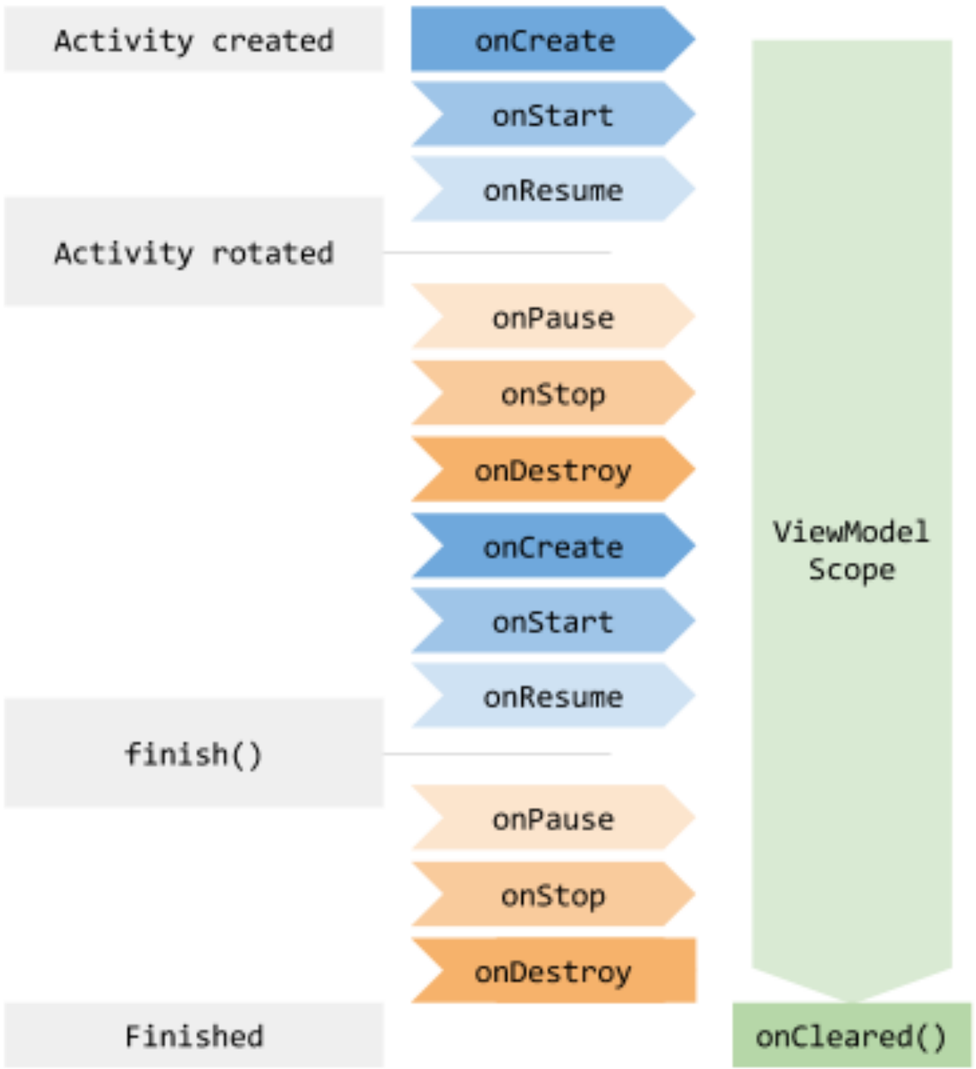
Jetpack

ViewModel

- Introduced at Google IO 2018
- Bootstrap Android development
- Opinionated implementations
- Break up support libraries
- Migrate to androidx namespace

Jetpack

ViewModel



Jetpack

ViewModel

- Android `Application` created
- Android `Activity` created
- Dagger `@Component` created
- Androidx `ViewModel` created
- Androidx `Fragment` created

Jetpack

ViewModel

— **Android `Application` created** ⟵

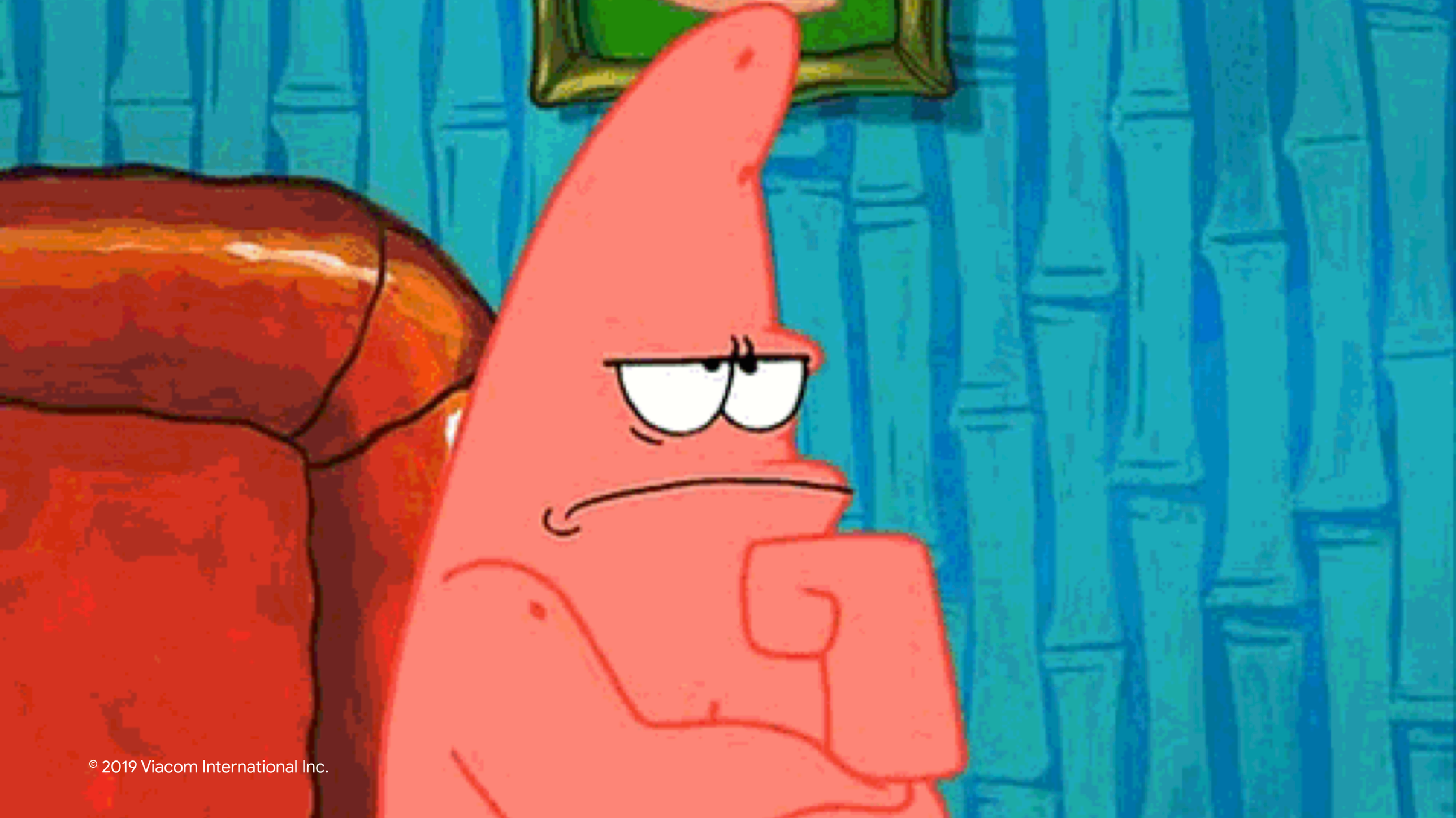
— ~~Android `Activity` created~~ 💀

— ~~Dagger `@Component` created~~ 💀

— **AndroidX `ViewModel` created** ⟵

— **AndroidX `Fragment` created** ⟵

⚠️ *Caution: A* `ViewModel` *must never reference a view,* `Lifecycle`*, or any class that may hold a reference to the activity context.*




© 2019 20th Century Fox

JetPack

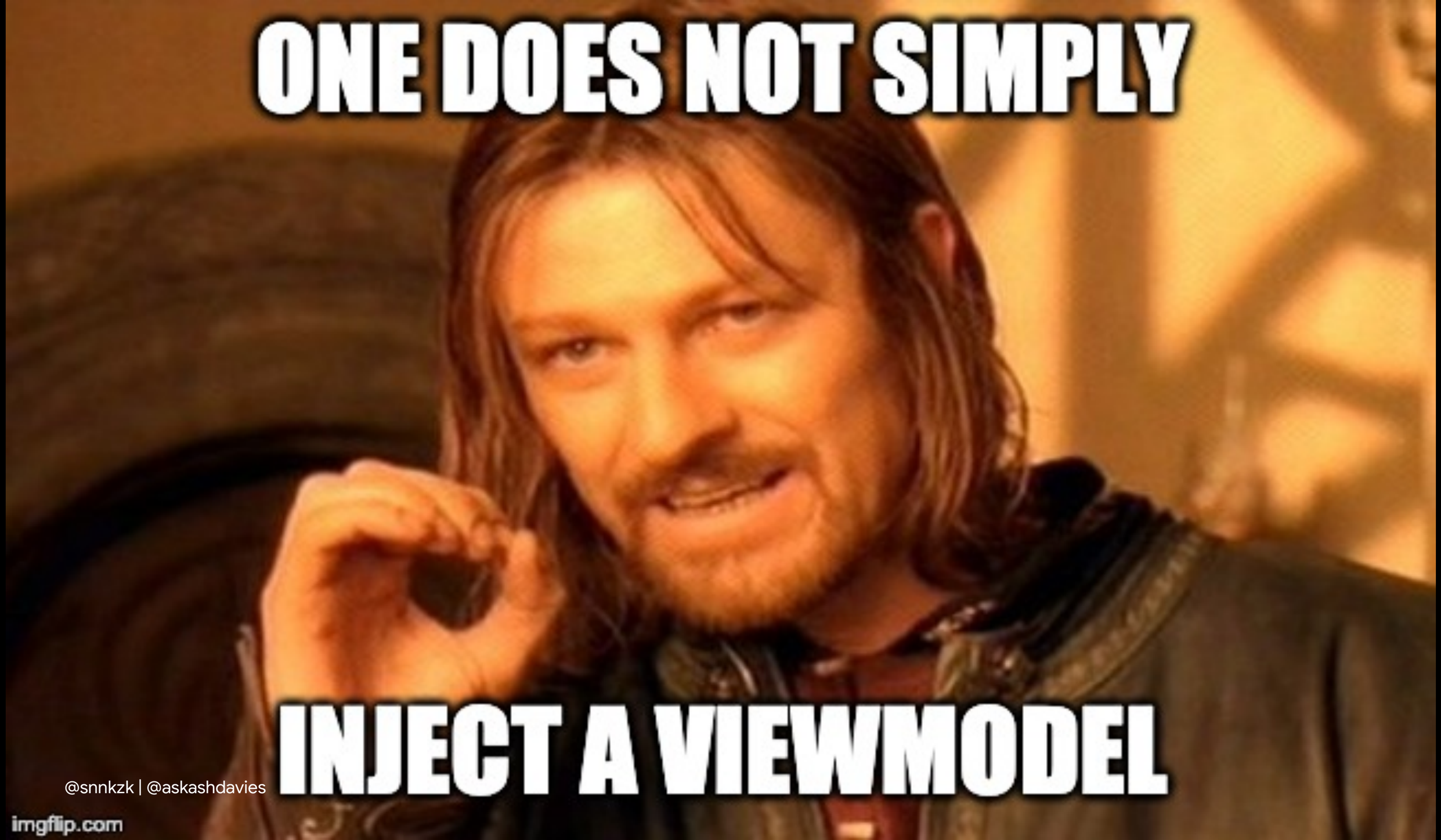
ViewModel

```
class SampleViewModel @Inject constructor() : ViewModel {
}

class Activity : DaggerAppCompatActivity {

    @Inject lateinit var model: SampleViewModel
}
```

JetPack

ViewModel

```
class SampleViewModel @Inject constructor() : ViewModel {
}

class Activity : DaggerAppCompatActivity {

    @Inject lateinit var model: SampleViewModel
}
```


ONE DOES NOT SIMPLY
INJECT A VIEWMODEL
@snnkzk | @askashdavies
imgflip.com

Jetpack: ViewModel

Dagger Multi-Binding

Jetpack: ViewModel

Dagger Multi-Binding

```
class ActivityViewModel @Inject constructor() : ViewModel() {
}
```

Jetpack: ViewModel

Dagger Multi-Binding

```
@MapKey
@Retention(RUNTIME)
annotation class ViewModelKey(val value: KClass<out ViewModel>)

@Module
interface ActivityViewModelModule {

  @Binds
  @IntoMap
  @ViewModelKey(ViewModel::class)
  fun model(model: ActivityViewModel): ViewModel
}
```

Jetpack: ViewModel

Dagger Multi-Binding

```
class ViewModelFactory @Inject constructor(
    private val creators: Map<Class<out ViewModel>,
    @JvmSuppressWildcards Provider<ViewModel>>
) : ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
        override fun <T : ViewModel> create(kls: Class<T>): T {
        var creator: Provider<out ViewModel>? = creators[kls]

        creator ?: creators.keys.firstOrNull(kls::isAssignableFrom)?.apply { creator = creators[this] }
        creator ?: throw IllegalArgumentException("Unrecognised class $kls")

        return creator.get() as T
    }
}
```

Jetpack: ViewModel

Dagger Multi-Binding

```
class ViewModelActivity : DaggerAppCompatActivity {

    private lateinit var model: ActivityViewModel

    @Inject internal lateinit var factory: ViewModelFactory

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

        model = ViewModelProviders
            .of(this, factory)
            .get(ActivityViewModel::class.java)
    }
}
```

Jetpack: ViewModel

androidx.activity:activity-ktx:1.0.0-rc01

```
class ViewModelActivity : DaggerAppCompatActivity {

    private val model: ActivityViewModel by viewModels { factory }

    @Inject internal lateinit var factory: ViewModelFactory

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

Jetpack: ViewModel

bit.ly/view-model-factory

- Uses Dagger Multi-Binding to build map of `Provider`'s
- Global `Factory` to create all ViewModel's
- `Factory` injected into `Activity` to create `ViewModel`
- Complicated initial set-up configuration
- Needs map binding `@Module` for every `ViewModel`
- Application graph polluted with all `Factory`'s

Plaid: HomeViewModelFactory

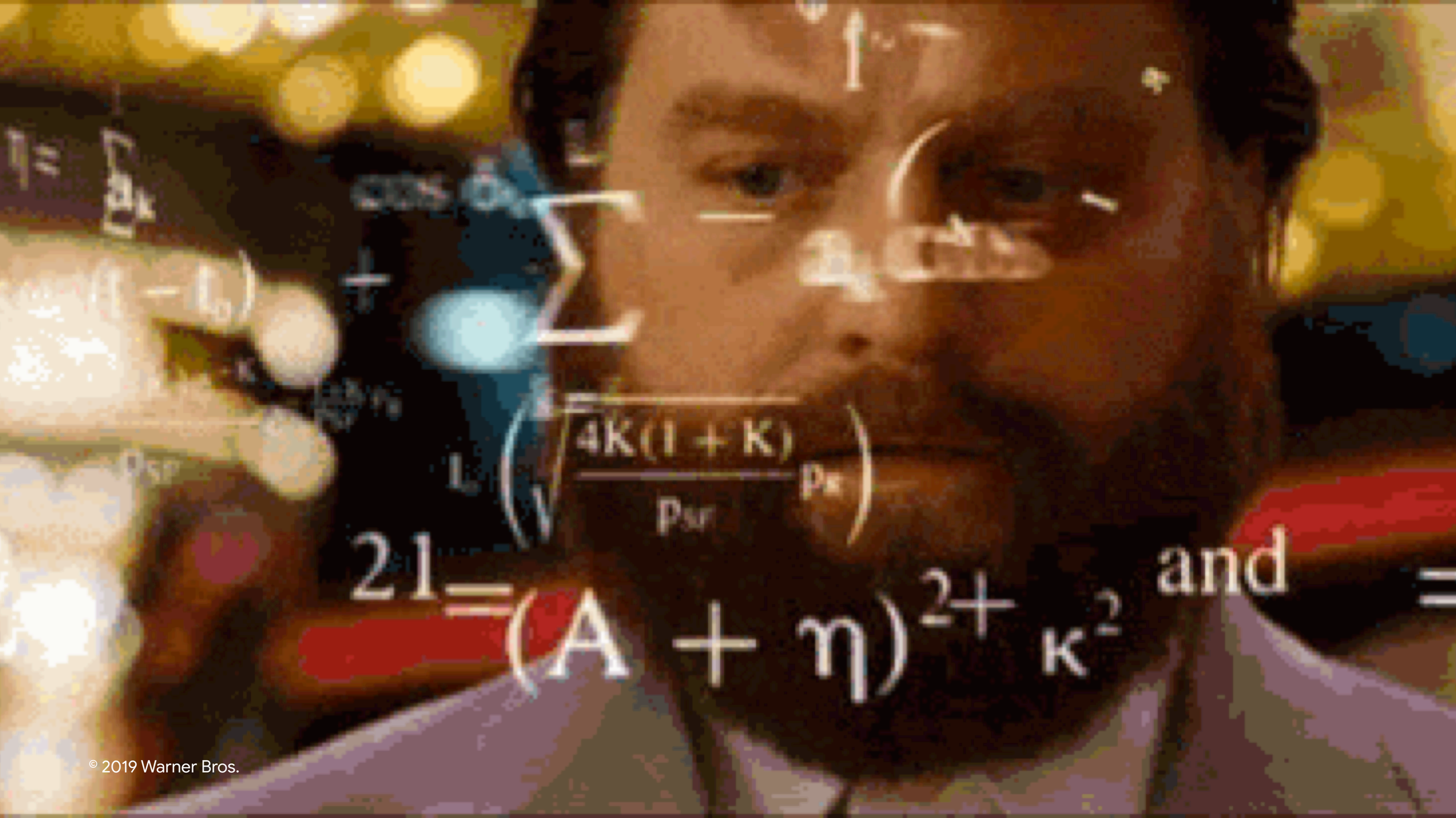
```
class HomeViewModelFactory @Inject constructor(
    private val dataManager: DataManager,
    private val designerNewsLoginRepository: LoginRepository,
    private val sourcesRepository: SourcesRepository,
    private val dispatcherProvider: CoroutinesDispatcherProvider
) : ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass != HomeViewModel::class.java) {
            throw IllegalArgumentException("Unknown ViewModel class")
        }
        return HomeViewModel(
            dataManager,
            designerNewsLoginRepository,
            sourcesRepository,
            dispatcherProvider
        ) as T
    }
}
```

github.com/android/plaid/blob/master/app/src/main/java/io/plaidapp/ui/HomeViewModelFactory.kt

Plaid: AboutViewModelFactory

```
internal class AboutViewModelFactory @Inject constructor() : ViewModelProvider.Factory {

    @Inject lateinit var aboutViewModel: AboutViewModel

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return if (modelClass.isAssignableFrom(AboutViewModel::class.java)) {
            aboutViewModel as T
        } else {
            throw IllegalArgumentException(
                "Class ${modelClass.name} is not supported in this factory."
            )
        }
    }
}
```

https://github.com/android/plaid/blob/master/about/src/main/java/io/plaidapp/about/ui/model/AboutViewModelFactory.kt



Jetpack: ViewModel

bit.ly/view-model-provider

```
internal class ViewModelFactory(
    private val provider: Provider<out ViewModel>
) : ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return try {
            provider.get() as T
        } catch (exception: ClassCastException) {
            throw IllegalArgumentException(
                "Class ${modelClass.name} is not supported by this factory",
                exception
            )
        }
    }
}
```

Jetpack: ViewModel

bit.ly/view-model-provider

```
class ActivityViewModel @Inject constructor() : ViewModel() {

    class Factory @Inject constructor(
        provider: Provider<ActivityViewModel>
    ) : ViewModelFactory(provider)
}

class ViewModelActivity : DaggerAppCompatActivity {

    private val model: ActivityViewModel by viewModels { factory }

    @Inject internal lateinit var factory: ActivityViewModel.Factory
}
```

Android: *Factory

- `AppComponentFactory`
  Android Pie 🥧
- `FragmentFactory`
  fragmentx:1.1.0-rc01
- `AbstractSavedStateVMFactory`
  lifecycle-viewmodel-savedstate:1.0.0-alpha02
- `LayoutInflater.Factory2`
  Android Cupcake 🧁

Android: LayoutInflater.Factory2

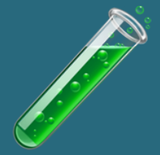
github.com/square/AssistedInject

```
class ImageLoaderView @AssistedInject constructor(
    @Assisted context: Context,
    @Assisted attrs: AttributeSet,
    private val loader: ImageLoader
) : ImageView(context, attrs) {

    @AssistedInject.Factory
    interface Factory {

        fun create(
            context: Context,
            attrs: AttributeSet
        ): ImageLoaderView
    }
}
```

Android: LayoutInflater.Factory2

github.com/square/AssistedInject

```
class ApplicationLayoutInflaterFactory @Inject constructor(
    private val imageLoaderFactory: ImageLoaderView.Factory
) : LayoutInflater.Factory {

    override fun onCreateView(
        name: String,
        context: Context,
        attrs: AttributeSet
    ): View? {
        if (name == ImageLoaderView::class.java.name) {
            return imageLoaderFactory.create(context, attrs)
        }
        return null
    }
}
```

Kotlin: Experimental 

Kotlin: Experimental 🧪

Inline Classes

- Wrapping types can introduce runtime overhead
- Performance worse for primitive types
- Initialised with single backing property
- Inline classes represented by backing field at runtime
- Sometimes represented as boxed type...

Kotlin: Experimental 🧪

Inline Classes

- Dagger recognises inline class as it's backing type
- Module @Provide not complex enough to require wrapper
- @Inject sites not complex enough to require wrapper
- Can cause problems if backing type not qualified
- Operates the same for typealias

Use Kotlin interfaces for @Binds modules

- Delegating one type to another
  - Use `@Binds` instead of a `@Provides` method
  - No code generation involved
- Should I use an `abstract class` or `interface`?
  - Doesn’t matter
  - `interface` is more cleaner
  - `abstract` can have internal method
- `interface` with default implementation?
  - No

Inlined method bodies in Kotlin

- Kotlin return types can be inferred from method body
- Android Studio shows inlining return types
- Return types to hide implementation detail easily missed
  - Interface vs Implemetation
- Best practice to explicitly specify return type
- Easier to review, easier to understand, avoids compiler errors
- Framework types (Fragment.context) can be assumed nullable

Dagger Factory's

- Inject annotated classes generate factory at usage site
- If `@Module` is not necessary in the gradle module
  - Prefer `@Inject` annotation
  - Don't use dagger compiler where possible

Dagger Factory's

- For keeping implementation internal prefer `abstract` module and use internal methods
- Injected constructor can be internal
- Root module needs dependencies for submodule
  - if in Dagger graph, is required in app module

Default Parameters in Dagger

- Dagger doesn't recognise default parameters even with `@JvmOverloads`
- `@JvmOverloads` will generate all constructors with `@Inject`
- Class can only have one `@Inject` constructor
- Best practice to define an alternative annotated constructor

Further Reading 📖

- **Dave Leeds: Inline Classes and Autoboxing**
  - https://typealias.com/guides/inline-classes-and-autoboxing/
- **Kotlin: Declaration Site Variance**
  - https://kotlinlang.org/docs/reference/generics.html#declaration-site-variance
- **Kotlin: Variant Generics**
  - https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html#variant-generics
- **Jake Wharton: Helping Dagger Help You**
  - https://jakewharton.com/helping-dagger-help-you/
- **Dagger: Kotlin Dagger Best Practices**
  - https://github.com/google/dagger/issues/900
- **Fred Porciúncula: Dagger 2 Official Guidelines**
  - https://proandroiddev.com/dagger-2-on-android-the-official-guidelines-you-should-be-following-2607fd6c002e
- **Warren Smith: Dagger & Kotlin**
  - https://medium.com/@naturalwarren/dagger-kotlin-3b03c8dd6e9b
- **Nazmul Idris: Advanced Dagger 2 w/ Android and Kotlin**
  - https://developerlife.com/2018/10/21/dagger2-and-kotlin/

IMMOBILIEN SCOUT 24

Thanks!

Sinan Kozak & Ash Davies

@snnkzk | @askashdavies