

# Dagger & Kotlin

Ash Davies

 DevFest.cz  
2019



# What is dependency injection? 🤔

# Dependency Injection

```
class Car {  
  
    private val engine = Engine()  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val car = Car()  
    car.start()  
}
```

# Dependency Injection

```
class Car {  
  
    private val engine = Engine()  
  
    fun start() {  
        engine.start()  
    }  
}  
  
fun main(args: Array) {  
    val car = Car()  
    car.start()  
}
```

# Dependency Injection

```
class Car(private val engine: Engine) {
```

```
    fun start() {
        engine.start()
    }
}
```

```
    fun main(args: Array) {
        val engine = Engine()
        val car = Car(engine)
        car.start()
    }
}
```

**Wtf** 

**Dagger sounds confusing, I'll just use something else**

Dagger2 Vs Koin for dependency

reddit.com/r/androiddev/comments/8ch4cg/dagger2\_vs\_koin\_for\_dependency\_injection/

reddit r/androiddev Search r/androiddev LOG IN SIGN UP

↑ 51 ↓ Dagger2 Vs Koin for dependency injection ? X CLOSE

Posted by u/passiondroid 1 year ago

## 51 Dagger2 Vs Koin for dependency injection ?

I have used Dagger2 in many of my projects. But each time setting up a new project with Dagger2 requires a lot of boilerplate code and as new features are added to the app comes a lot subcomponents and modules as well. So I was thinking of trying Koin for DI. Just wanted to know how many of you have tried it and how easy it is to get started ?

47 Comments Share Save Hide Report 90% Upvoted

This thread is archived New comments cannot be posted and votes cannot be cast

SORT BY BEST

↑ JakeWharton Head of sales at Bob's Discount ActionBars 50 points · 1 year ago

↓ Since Koin isn't a dependency injector but a service locator with a clever reified trick that you can use to manually perform dependency injection, the boilerplate will scale disproportionately. With Dagger (and Guice, et. al.) there's a certain amount of fixed overhead but then you rarely have to significantly alter the shape of your graph as bindings propagate throughout injected types automatically. With manual dependency injection, you have to propagate bindings throughout injected types manually.

If you're writing a small toy app then it won't matter. You might as well not even use a library. But if you're going to write a serious app with hundreds of bindings and hundreds of injected types with a deep graph of types then you're better off with a proper injector that generates the code that you otherwise manually write worth Koin.

r/androiddev 125k Members 526 Online Jul 12, 2009 Cake Day News for Android developers with the who, what, where when and how of the Android community. Probably mostly the how. Here, you'll find: - News for Android developers - Thoughtful, informative articles - Insightful talks and presentations - Useful libraries - Handy tools - Open source applications for studying JOIN ADVERTISEMENT

Dagger2 Vs Koin for dependency

reddit.com/r/androiddev/comments/8ch4cg/dagger2\_vs\_koin\_for\_dependency\_injection/

reddit r/androiddev Search r/androiddev LOG IN SIGN UP

51 Dagger2 Vs Koin for dependency injection ?

Posted by u/passiondroid 1 year ago

## Dagger2 Vs Koin for dependency injection ?

I have used Dagger2 in many of my projects. But each time setting up a new project with Dagger2 requires a lot of boilerplate code and as new features are added to the app comes a lot subcomponents and modules as well. So I was thinking of trying Koin for DI. Just wanted to know how many of you have tried it and how easy it is to get started ?

47 Comments Share Save Hide Report 90% Upvoted

This thread is archived New comments cannot be posted and votes cannot be cast

SORT BY BEST

JakeWharton Head of sales at Bob's Discount ActionBars 50 points · 1 year ago

Since Koin isn't a dependency injector but a service locator with a clever reified trick that you can use to manually perform dependency injection, the boilerplate will scale disproportionately. With Dagger (and Guice, et. al.) there's a certain amount of fixed overhead but then you rarely have to significantly alter the shape of your graph as bindings propagate throughout injected types automatically. With manual dependency injection, you have to propagate bindings throughout injected types manually.

If you're writing a small toy app then it won't matter. You might as well not even use a library. But if you're going to write a serious app with hundreds of bindings and hundreds of injected types with a deep graph of types then you're better off with a proper injector that generates the code that you otherwise manually write worth Koin.

r/androiddev 125k Members 526 Online Jul 12, 2009 Cake Day News for Android developers with the who, what, where when and how of the Android community. Probably mostly the how. Here, you'll find: - News for Android developers - Thoughtful, informative articles - Insightful talks and presentations - Useful libraries - Handy tools - Open source applications for studying JOIN ADVERTISEMENT



TRIGGERED.

# Service Locator

```
object ServiceLocator {
```

```
    fun getEngine(): Engine = Engine()
```

```
}
```

```
class Car {
```

```
    private val engine = ServiceLocator.getEngine()
```

```
    fun start() {
```

```
        engine.start()
```

```
}
```

```
}
```

# What is Dagger



- **Dependency injection implementation**
- **Generates code for injection**
- **Compile time - sans reflection**

# In Java

# In Java, for Java

# In Java, for Java, by Java

# In Java, for Java, by Java, with Java developers

**In Java, for Java, by Java, with Java  
developers, for Ja....**

# Java





HBO

# Code Generation



# AutoValue

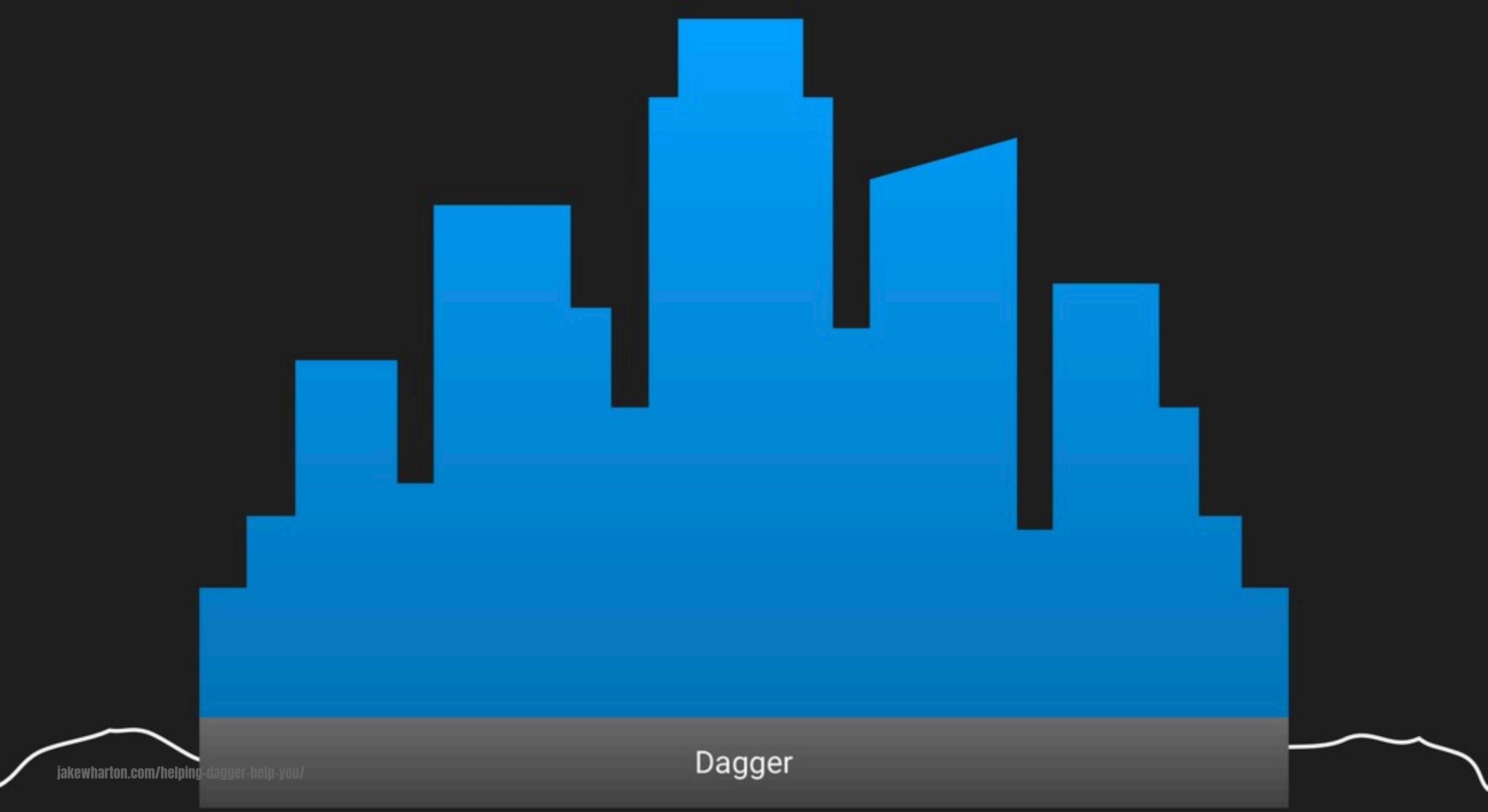


# Lombok



# Dagger





# History



# Guice



(Pronounced 'juice')

# Dagger

(JSR-330)

# Dagger 2 G

(Sans-Reflection)



(noun: freakin' awesome)

# Dagger 2



Kotlin ❤

# Dagger Qualifiers

**Qualifiers used to identify dependencies with identical signatures**

# Retention Annotation

**Use Kotlin retention annotations instead of Java retention**

# Constructor injection

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

# Constructor injection

```
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

```
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;  
    }  
}
```





**OH MY GOD  
THAT'S SO YESTERDAY!**

Kotlin+Dagger best practices/doc #900

github.com/google/dagger/issues/900

Search or jump to... Pull requests Issues Marketplace Explore

google / dagger forked from square/dagger

Unwatch 566 Star 14.4k Fork 2.8k

Code Issues 123 Pull requests 17 Actions Projects 0 Wiki Security Insights

## Kotlin+Dagger best practices/documentation/pain points #900

**Open** ronshapiro opened this issue on 15 Oct 2017 · 42 comments

ronshapiro commented on 15 Oct 2017

Opening this as a tracking bug for all kotlin related documentation that we should be add/best practices that we should call out to make using Dagger w/ Kotlin easier.

One example: How to achieve the effect of `static @Provides` in Kotlin.

Feel free to comment new ideas, but don't make "me too" or "i agree with XYZ" comments.

60

google deleted a comment from bejibx on 16 Oct 2017

ZacSweers commented on 16 Oct 2017 · edited

If you have injected properties (as "fields"), qualifiers *must* have `field:` designation.

Good

Assignees  
No one assigned

Labels  
`status=triaged`

Projects  
None yet

Milestone  
No milestone

Notifications  
Customize  
Unsubscribe

You're receiving notifications because you're watching this repository.

# Field Injection: 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$annotations0 {}  
  
    public final Player getPlayer10 { ... }  
  
    public final void setPlayer1(Player var1) {...}  
  
    @Named("P2") public static void player2$annotations0 {}  
  
    public final Player getPlayer20 { ... }  
  
    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$annotations0 {}  
  
    public final Player getPlayer10 { ... }  
  
    public final void setPlayer1(Player var1) {...}  
  
    @Named("P2") public static void player2$annotations0 {}  
  
    public final Player getPlayer20 { ... }  
  
    public final void setPlayer2(Player var1) {...}
```

# Specify Annotations

- **@field:...**
- **@set:...**
- **@get:...**
- **@param:...**
- **@property:...**
- **@setparam:...**
- **@receiver:...**
- **@delegete:...**



# Specify Annotations

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

# Specify Annotations

```
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) {...}  
}
```

# Constructor vs Property injection

## Constructor injection

- **Immutable** 💪
- **Easy to use** 😊
- **Reliable injection** 🔨
- **Compilation safety** 🛠

# Constructor vs Property injection

## Property injection

- Mutable (lateinit) properties 💔
- Annotation target unclear 🤷
- Difficult to configure tests 🔧

# Property injection



# Android

- **Activity**
- **Fragment**
- **Service**

# 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

@ActivityScope

fun context(application: Application): Context = application

}

@ActivityScope



# @Scope



**@ActivityScope // Don't do this!**

```
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.Companion0();

    @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 Companion0 {
        }
    }
}
```

# 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 ApplicationModule0;  
  
    @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 ApplicationModule0 {  
    }  
}
```

# 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);  
    }  
}
```



# But wait...

# Dagger 2.25.2



## Kotlin support

- Qualifier annotations on fields can now be understood without the need for `@field:MyQualifier (646e033)`
- `@Module` object classes no longer need `@JvmStatic` on the provides methods. (`Oda2180`)



# Qualifier annotations

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

# Qualifier annotations

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

# Dagger: Modules

```
object ApplicationModule {  
  
    @Provides  
    @JvmStatic  
    fun context(application: Application): Context = application  
  
    @Provides  
    @JvmStatic  
    fun repository(name: String): SampleRepository = SampleRepository(name)  
}
```

# Dagger: Modules

```
object ApplicationModule {  
  
    @Provides  
    fun context(application: Application): Context = application  
  
    @Provides  
    fun repository(name: String): SampleRepository = 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

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

# Kotlin: Generics<? : T>

## Dagger Multi-Binding

```
@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>

## Java Interoperability

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

# Jetpack

# **Jetpack**

## **ViewModel**

# Jetpack

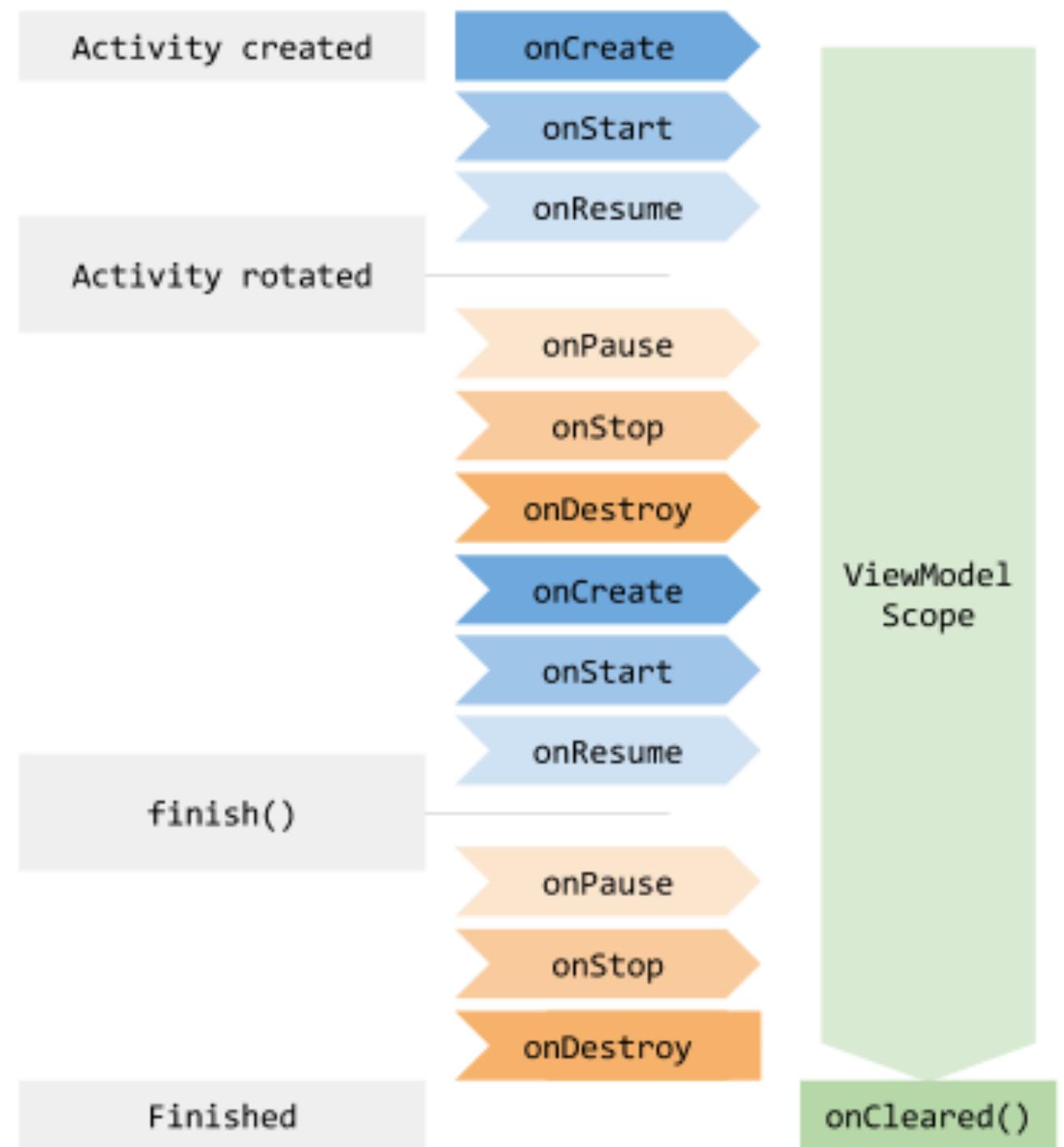
## ViewModel

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



# Jetpack

## ViewModel



# Jetpack

## ViewModel

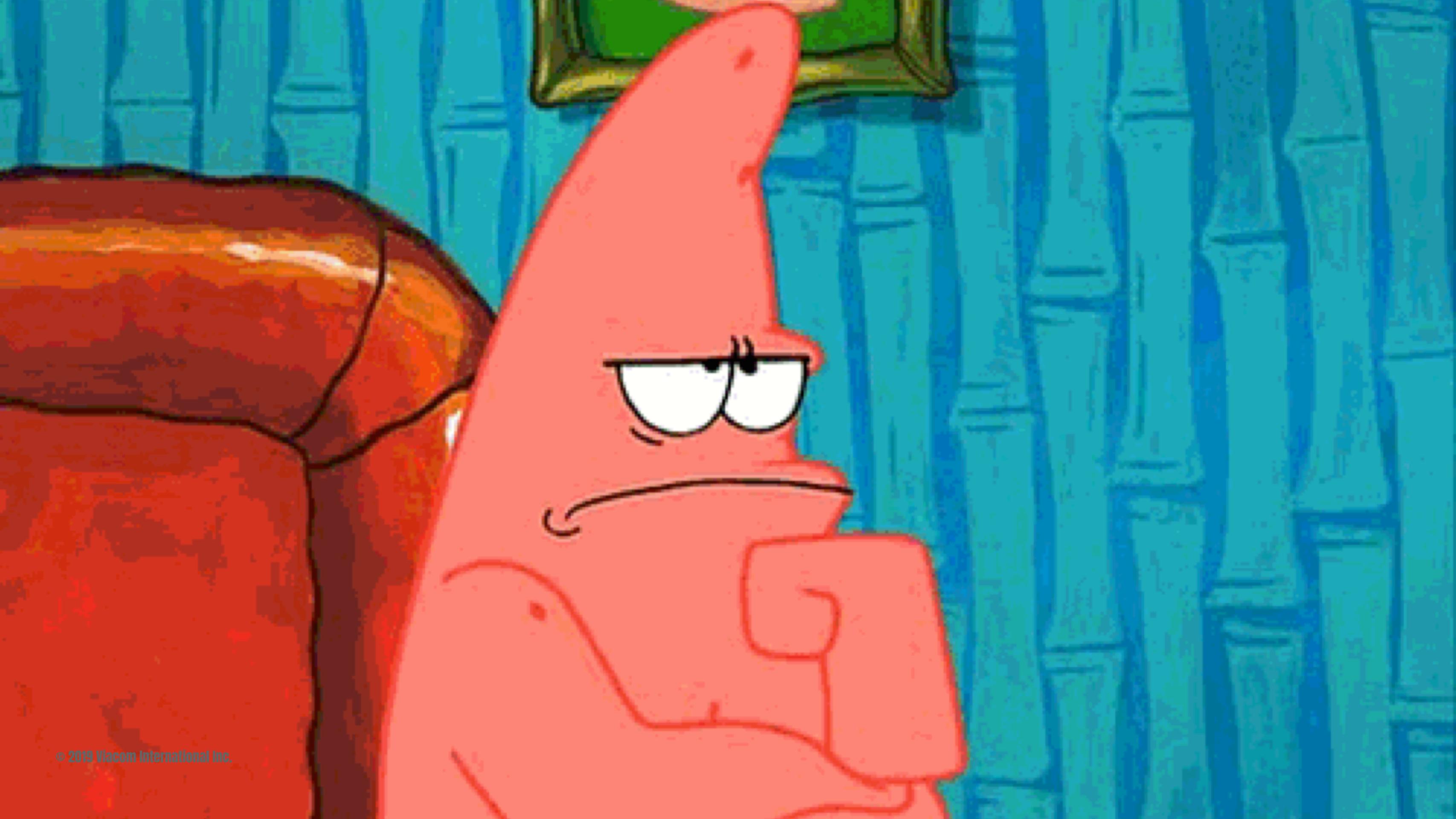
- **Android Application created**
- **Android Activity created**
- **Dagger @Component created**
- **Androidx ViewModel created**
- **Androidx Fragment created**

# 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 Viacom International Inc.



© 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

# **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**

```
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](https://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**

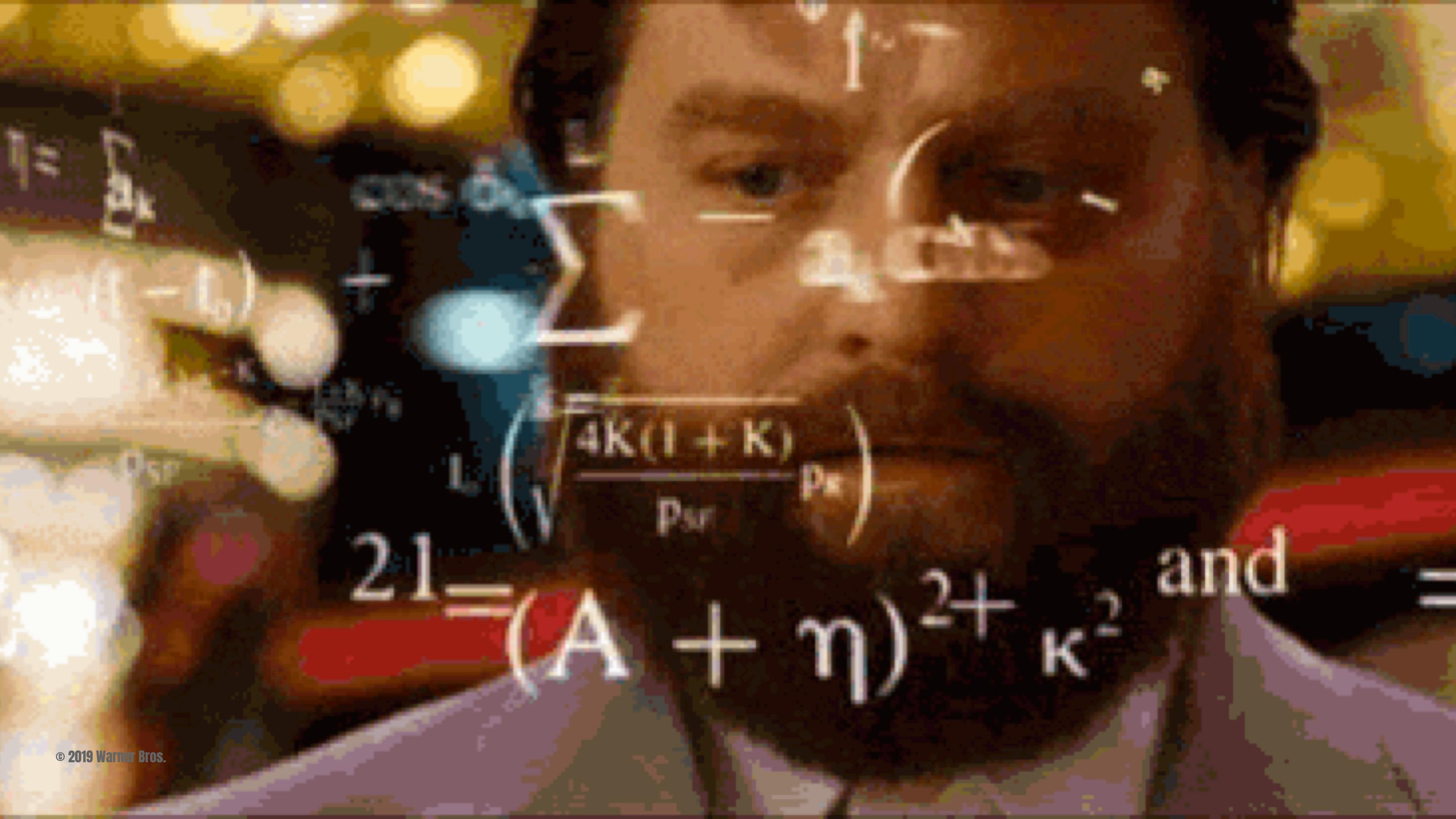
# Good: HomeViewModelFactory

```
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
    }
}
```

# Not So Good: OtherViewModelFactory

```
internal class OtherViewModelFactory @Inject constructor() : ViewModelProvider.Factory {  
  
    @Inject lateinit var otherViewModel: OtherViewModel  
  
    @Suppress("UNCHECKED_CAST")  
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {  
        return if (modelClass.isAssignableFrom(OtherViewModel::class.java)) {  
            otherViewModel as T  
        } else {  
            throw IllegalArgumentException(  
                "Class ${modelClass.name} is not supported in this factory."  
            )  
        }  
    }  
}
```



$$21 = A + m^2 + \kappa^2 \text{ and}$$

$$P_{\text{eff}} = \frac{4K(1+K)}{P_0}$$

# Jetpack: ViewModel

[bit.ly/view-model-provider](https://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](https://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  
}
```

PE MÍS CHI CHIS

NAE TOQUEM  
HUEVOS

SERGA ES FT  
ENIA.

WAIT, WAIT,  
THERE'S MORE.



# Android: \*Factory

- **AppComponentFactory**

Android Pie 

- **FragmentManager**

fragmentx:1.1.0

- **AbstractSavedStateVMFactory**

lifecycle-viewmodel-savedstate:1.0.0-alpha05

- **LayoutInflater.Factory2**

Android Cupcake 

# Android: LayoutInflater.Factory2

[github.com/square/AssistedInject](https://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](https://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

# @Binds

# @Binds

@Module

```
interface MySuperAwesomeHappyFantasticModule {
```

@Binds

```
fun activity(activity: FragmentActivity): MySuperAwesomeHappyFantasticActivity
```

```
}
```

**@Binds**

**@Module**

**abstract class MySuperAwesomeHappyFantasticModule {**

**@Binds**

**abstract fun activity(activity: FragmentActivity): MySuperAwesomeHappyFantasticActivity**  
**}**

# @Binds

@Module

```
interface MySuperAwesomeHappyFantasticModule {
```

@Binds

```
fun activity(activity: FragmentActivity): MySuperAwesomeHappyFantasticActivity
```

```
}
```

# Default Implementations?

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



HAHAHA

# Inlined method bodies in Kotlin

Kotlin return types can be inferred from method body

# Default Parameters?

# Default Parameters?



# @JvmOverloads

@JvmOverloads



# Hope 🙏

**[bit.ly/dagger-kotlin-support](https://bit.ly/dagger-kotlin-support)**

#AndroidDevSummit

# Dependency Injection on Android



[youtube.com/watch?v=o-ins1nvbDg](https://youtube.com/watch?v=o-ins1nvbDg)



# dagger.android



## @ContributesAndroidInjector

[dagger.android](#) is NOT deprecated. At [#AndroidDeveloperSummit](#), we communicated that we won't be adding new features (major improvements) to the library but we're committed to maintaining it until a suitable stable replacement is available.



**Manuel Vivo** @manuelvicnt  
5:52pm - 28 Oct 2019

# Further Reading



- Manuel Vivo: An Opinionated Guide to Dependency Injection on Android  
[youtube.com/watch?v=o-ins1nvbDg](https://youtube.com/watch?v=o-ins1nvbDg)
- Google Codelab: Using Dagger in your Android app  
[codelabs.developers.google.com/codelabs/android-dagger/](https://codelabs.developers.google.com/codelabs/android-dagger/)
- Dave Leeds: Inline Classes and Autoboxing  
[typealias.com/guides/inline-classes-and-autoboxing/](https://typealias.com/guides/inline-classes-and-autoboxing/)
- Jake Wharton: Helping Dagger Help You  
[jakewharton.com/helping-dagger-help-you/](https://jakewharton.com/helping-dagger-help-you/)
- Dagger: Kotlin Dagger Best Practices  
[github.com/google/dagger/issues/900](https://github.com/google/dagger/issues/900)
- Fred Porciúncula: Dagger 2 Official Guidelines  
[proandroiddev.com/dagger-2-on-android-the-official-guidelines-you-should-be-following-2607fd6c002e](https://proandroiddev.com/dagger-2-on-android-the-official-guidelines-you-should-be-following-2607fd6c002e)
- Zac Sweers: Dagger Party tricks  
[zacsweers.dev/dagger-party-tricks-deferred-okhttp-init/](https://zacsweers.dev/dagger-party-tricks-deferred-okhttp-init/)

# Thanks!



<> DevFest.cz  
2019