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. (Not a service locator)
- 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
- 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
  - At least BINARY retention but RUNTIME is ideal
  - Dagger 2 doesn’t operate on source files

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 Game1 @Inject constructor() {

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

Decompiled lateinit var 🛑

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

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

   @NotNull public final Player getPlayer1() { ... }

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

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

   @NotNull public final Player getPlayer2() { ... }

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

Decompiled lateinit var 🛑


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

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

   @NotNull public final Player getPlayer1() { ... }

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

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

   @NotNull public final Player getPlayer2() { ... }

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


@snnkzk | @askashdavies

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

- Qualified field injection requires @field annotation on property
  - Generated JVM code for both @field:Qualifier and @Qualifer
  - Generated JVM code for property getter and setter
  - Show where @Inject annotation is applied by default
  - Show JVM code for @set:Inject with @Qualifer
- Qualifier also required for constructor injection but clearer
- Java declarations explicit whereas Kotlin requires specification
  - Kotlin uses syntactic sugar to reduce boilerplate
  - Not clear to Kotlin compiler what we want to annotate

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() {
   }
}
```

Use Kotlin interfaces for @Binds modules

- Separate objects from interfaces, reduce complexity
- Should I use an abstract class or interface? Doesn't matter
- Interface more consistent with the documentation for Factory
  - Citation needed
- Interface with default implementation? No.

Inlined method bodies in Kotlin

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

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

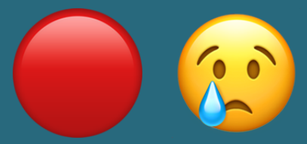
```
public final class ListAdapter {
    @Inject
    public ListAdapter(@NotNull List<? extends String> strings) {
        Intrinsics.checkParameterIsNotNull(strings, "strings");
        super();
    }
}
```

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>

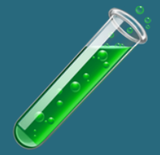
Dagger Multi-Bindings

```
@Module
object ListModule {

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

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

Jetpack: ViewModel

- Jetpack introduced us to ViewModel that persists through config change
  - Most Dagger components scoped to life of instance
  - ViewModel persistence is a form of scoping
  - Scope greater than activity or component instance scope
  - ViewModel can leak dependencies injected from Dagger
  - Dagger and ViewModel should be used with caution
- ViewModel annotated with Inject can be built with graph dependencies
  - Jetpack expects us to build ViewModel with Provider Factory
  - Using both in conjunction requires compatibility glue code
- One simply does not Inject a ViewModel
  - Injected ViewModel will be lost on activity death
  - Persistence achieved through ViewModelStore
  - ViewModel should not exist in Dagger graph
  - How can you still utilise Dagger?
- Implementing a Dagger ViewModelProvider.Factory
  - Plaid: HomeViewModelFactory (Delegation)
    - ViewModel constructor kept without annotation
    - Duplicate class to delegate parameters
  - Plaid: AboutViewModelFactory (Bad Example)
    - ViewModel instantiated every time
    - Duplicate ViewModel not used
  - Scout: ApplicationViewModelFactory (Multibinding)
    - Application scoped view model knows everything
    - Requires binding expression to include ViewModel in set
  - Scout: ViewModelFactory (ViewModel Scope)
    - Ideal, ProviderFactory once per ViewModel
    - Useful for dynamic features, independent of application
    - Constructor Provider generated by Dagger
    - Single responsibility preserved
- Fragment and Activity Factories can be defined in a similar fashion

@snnkzk | @askashdavies

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

Dagger Factory's

Dagger Factory's

- Inject annotated classes generate factory at usage sites
- Prefer inject annotation as the module compiler isn’t necessary
- Modules can be used to generate internal classes by abstract type
- Internal constructor for public classes with internal dependencies
- Dependency required in root module for submodule dependencies
  - Ie submodule using Coroutines requires app to include dependency
  - Even if it’s internal, if in Dagger graph, is required in app module
- Internal implementations can be hidden behind interfaces
- Submodule requires Dagger compiler to function correctly

Default Parameters in Dagger

- Dagger doesn’t recognise default parameters even with @JvmOverloads
- @JvmOverloads will generate all constructors with @Inject
- Types may only contain 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