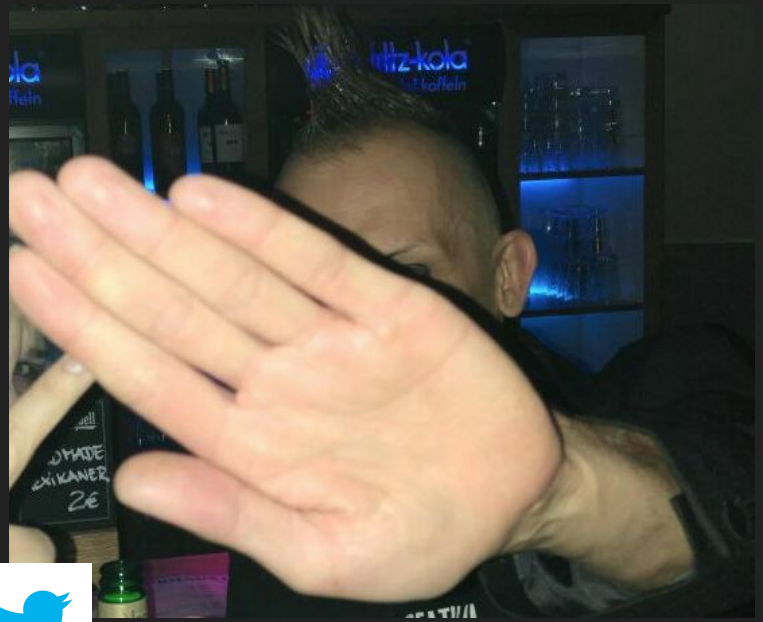
Dependency Injection on Android

frameworks & internals

Mobius 2017
St.Petersburg

VIACOM

GROUPON



@PreusslerBerlin

Lead Android Dev @ Viacom
Google Developer Expert

+stephane nicolas

Senior Android Dev @ Groupon
OSS: Dart, TP, BoundBox, …

Combined ~40 years of Java Coding

GROUPON

The Android Team is Hiring :

jobs.groupon.com/careers/


BET
CMT
COMEDY CENTRAL
Logo.
MTV
VIACOM
nickelodeon
Paramount
Spike
TV LAND
VH1


Dependency
Injection?

Dependency Inversion

?

?

Inversion of Control

Dependency Injection

?

The Dependency Inversion Principle

High level entities
should not depend on
low level details.

Inversion of Control

Who initiates a message

Hollywood's Law:
don't call me, I'll call you.

Inversion of Control

Stop using `new`

```
private final Tracker tracker =
            new GoogleAnalyticsTracker();

@Override
protected void onCreate(Bundle state) {
  ...

    tracker.trackStarted();
}
```

Inversion of Control

Common implementations:

- Factory
- Service Locator
- Dependency Injection

https://martinfowler.com/articles/dipInTheWild.html#YouMeanDependencyInversionRight

Inversion of Control

Common implementations:

- Factory

  ```
  tracker = Factory.createTracker()
  ```
- Service Locator
- Dependency Injection

https://martinfowler.com/articles/dipInTheWild.html#YouMeanDependencyInversionRight

Inversion of Control

Common implementations:

- Factory
- Service Locator

```
tracker = Locator.get(Tracker.class)
```

- Dependency Injection

https://martinfowler.com/articles/dipInTheWild.html#YouMeanDependencyInversionRight

Inversion of Control

Common implementations:

- Factory
- Service Locator
- Dependency Injection

```
@Inject Tracker tracker;
```

https://martinfowler.com/articles/dipInTheWild.html#YouMeanDependencyInversionRight

Dependency Injection

- Field injection
- Constructor injection
- Setter injection
- Method injection

Dependency Injection

- Field injection

  ```
  @Inject Tracker tracker;
  ```
- Constructor injection
- Setter injection
- Method injection

Dependency Injection

- Field injection
- Constructor injection

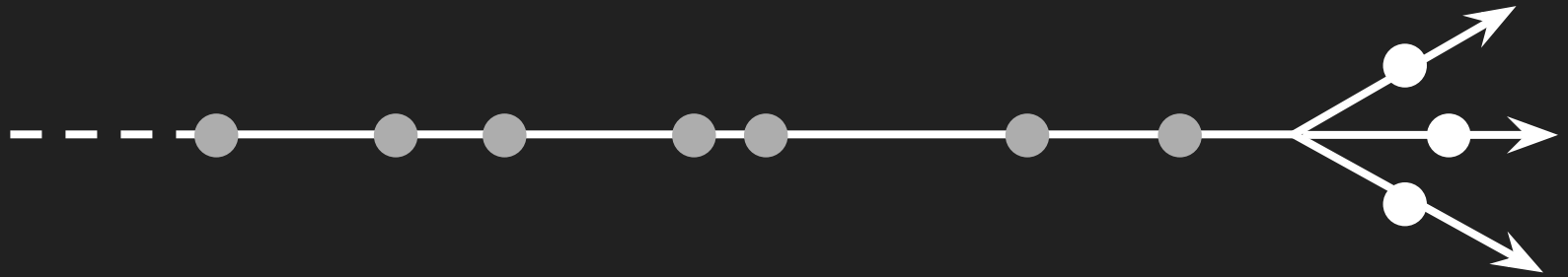
```
@Inject
MyClass(Tracker tracker) {...}
```

- Setter injection
- Method injection

A brief history of DI libs for Java / Android

Revolutions are the locomotives of history.

Karl Marx



A brief history of DI libs for Java / Android

March 8, 2007: Guice 1.0 is released.

Guice is annotation based to perform DI which is a huge improvement over former frameworks.
It uses reflection to access annotations, create instances and inject stuff.

A brief history of DI libs for Java / Android

October 2009: JSR 330 final draft released.

- Guice is de facto the first implementation of the JSR 330

A brief history of DI libs for Java / Android

May 2010: RoboGuice was launched !

- First DI lib on Android.
- Based on Guice (reflection).
- Supports view bindings, extras, events, etc..

A brief history of DI libs for Java / Android

June 2012: Dagger is started !

- The goal is to create a compile time implementation of JSR 330.

A brief history of DI libs for Java / Android

May 2013: Dagger 1.0.0 is launched !

- Compile time implementation of JSR 330.
  No more reflection or very very limited.
- Annotation processing at compile time.
- Generated code is used to assign members & create instances.

A brief history of DI libs for Java / Android

April 2015: Dagger 2.0.0 is launched !

- Faster than Dagger 1
- Easier error messages

A brief history of DI libs for Java / Android

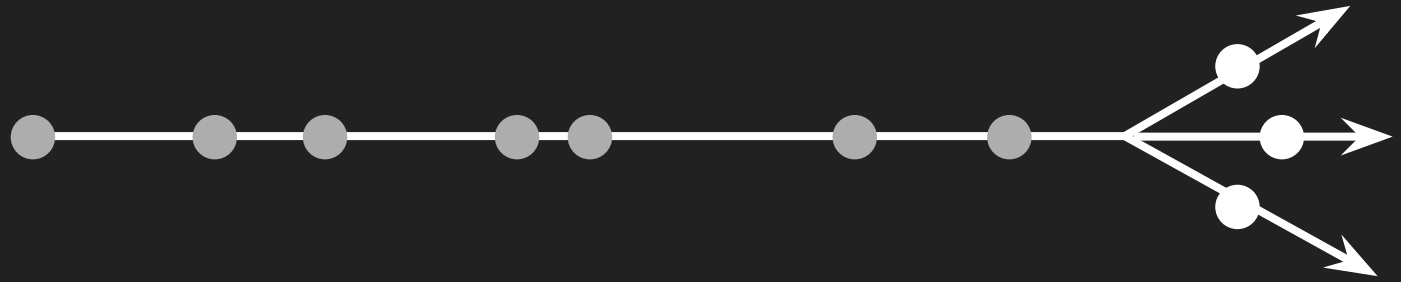
October 2016: Toothpick 1.0.0 !

- As fast as the daggers.
- Hybrid compile time and runtime.
- More flexible, simpler, amazing test support.

A brief history of DI libs for Java / Android

Many libs now :

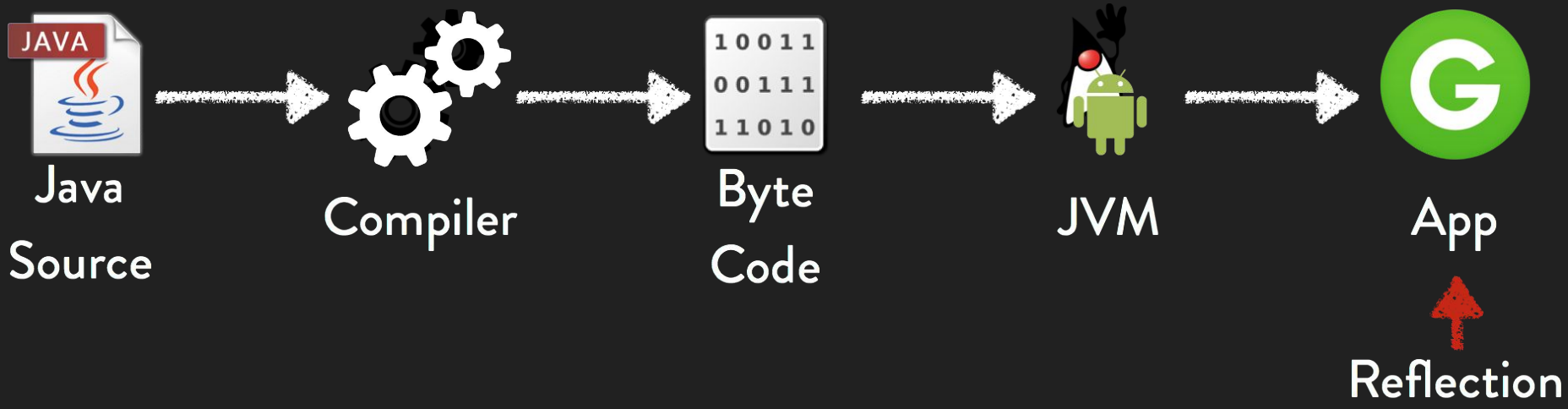
- Light saber (kotlin)
- Proton
- Feather
- Tiger (Dagger 2 improvements)



What is reflection ?

Why is bad on Android ?

What is reflection ?



What is reflection ?

- uses OOO concepts to represent objects, classes, methods, constructors, fields, annotations, etc.
- is an API to get a view of runtime java objects.
- is standard java.
- is relatively easy to use.

What is reflection ?

```
MyClass object = Myclass.class
    .getConstructors()[0].newInstance();

Method setter = Myclass.class
    .getDeclaredMethod("setFoo", {String.class});
setter.setAccessible(true);
setter.invoke(object, "set via reflection");

Field foo = MyClass.class.getDeclaredField("foo");
foo.setAccessible(true);
String value = foo.get(object);
```

Why is reflection slow (on Android) ?

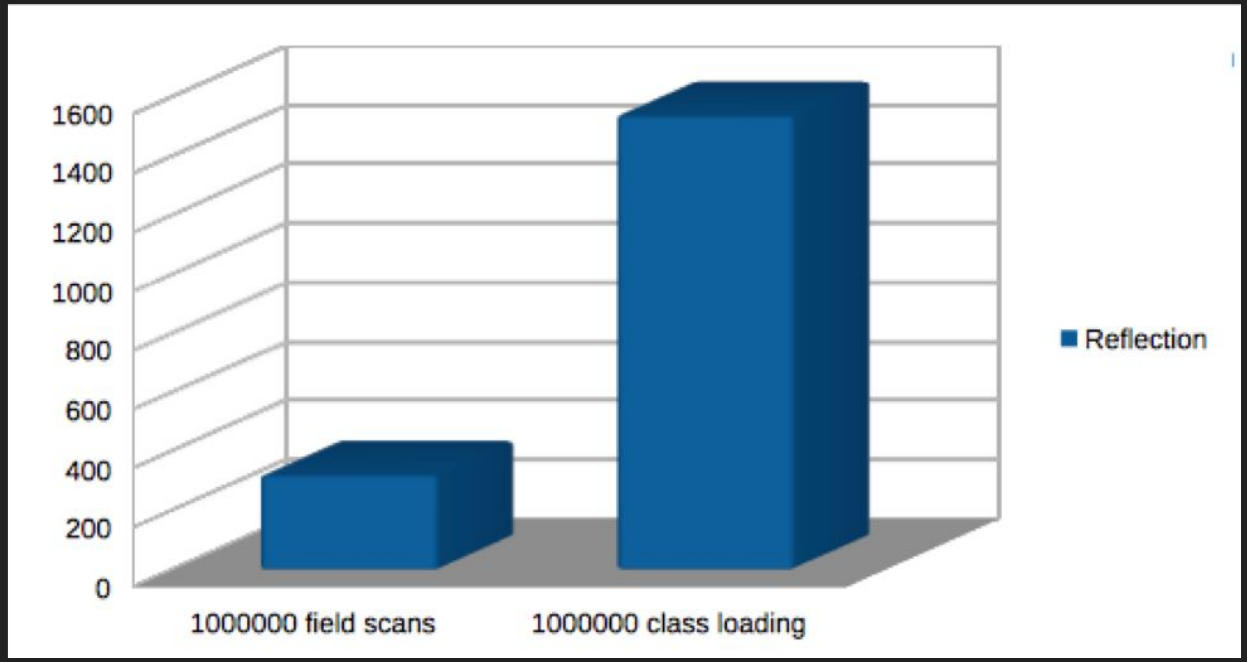
On a PC JVM

- Reflection calls are cached after 15 calls
- They are then transformed into normal code (JIT)
- 15 is parametrized by sun.reflect.inflation system property

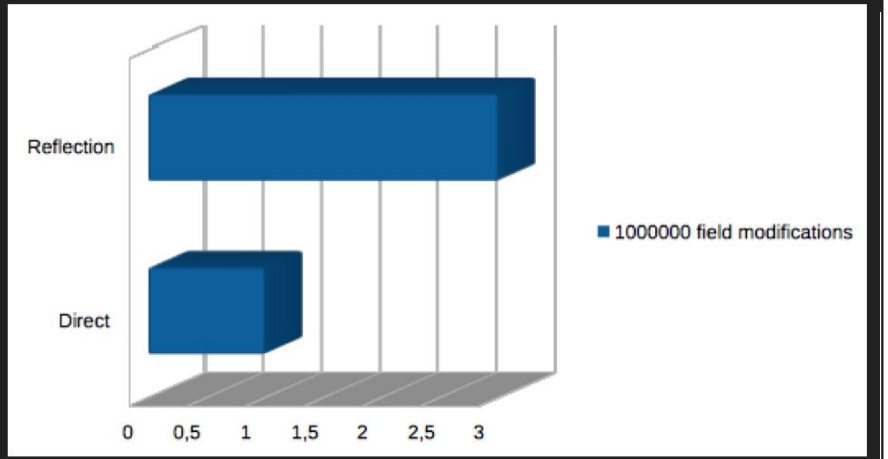
Why is reflection slow (on Android) ?

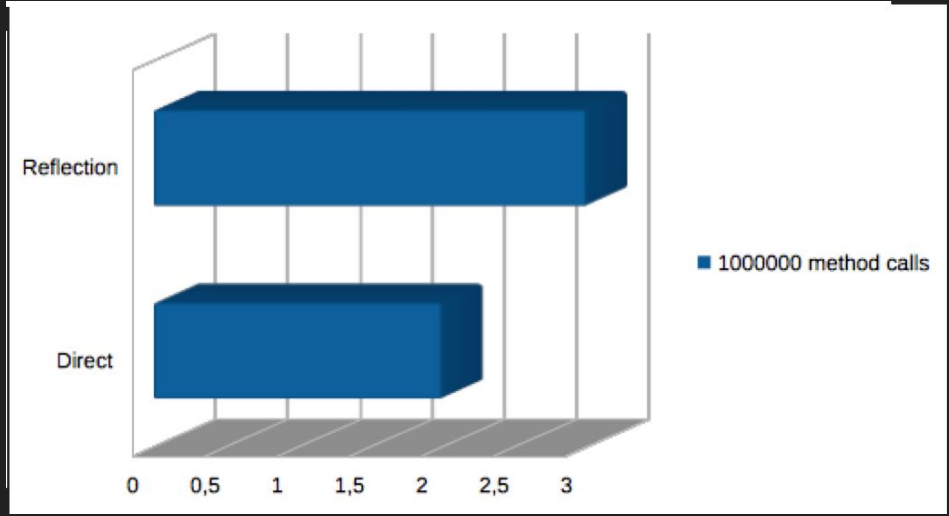
- On Android Dalvik
  - Reflection calls are not cached, no JIT
  - The dex format is not efficient for reflection
  - There was a bug that slowed down access to annotations by reflection (before GingerBread)
- On Android Art
  - In Nougat, reflection calls are now cached using JIT
  - But data structure of odex is still slow
  - Bug is solved

What is slow in reflection on Android ?

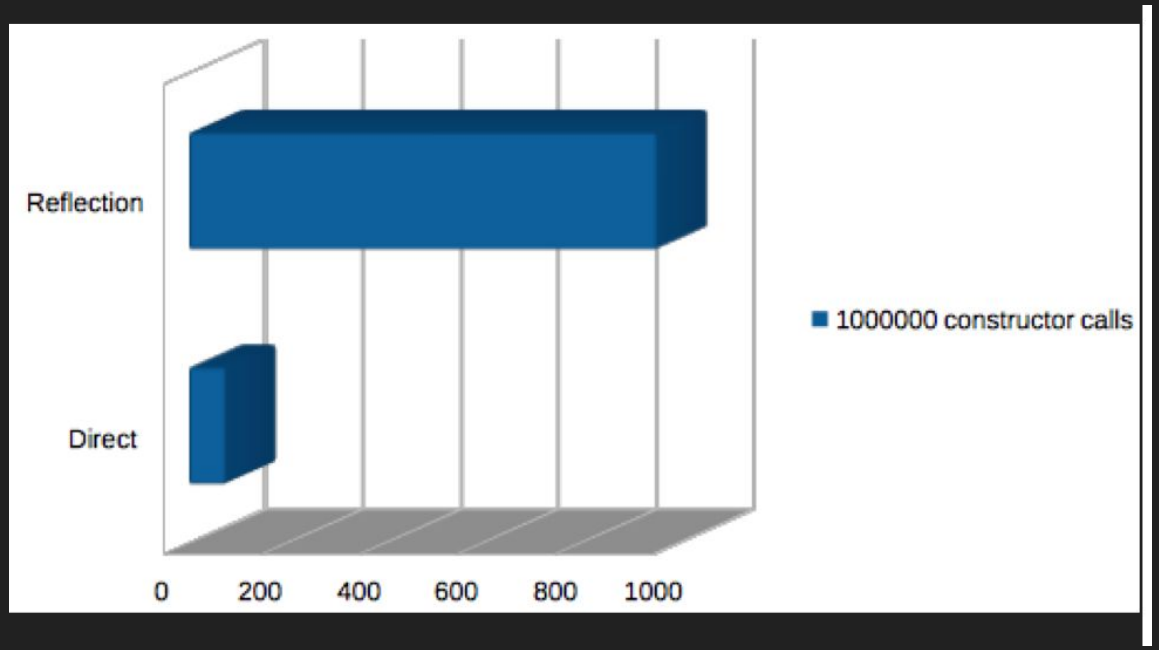


What is slow in reflection on Android ?





What is slow in reflection on Android ?



Dagger
Vs
Toothpick

Dagger vs Toothpick

- Usage
- Setup
- Scopes
- Tests
- Performance

Dagger vs Toothpick: Round 1: Usage

```
@Inject Tracker tracker;
```

Dagger vs Toothpick: Round 1: Usage

```
DaggerDependencies_AppComponent
        .builder()
        .build()
        .inject(this)
```

Dagger vs Toothpick: Round 1: Usage

```
DaggerDependencies_AppComponent
        .builder()
        .baseModule(new BaseModule(context))
        .build()
        .inject(this)
```

Dagger vs Toothpick: Round 1: Usage

```
@Inject Tracker tracker;
```

Toothpick

Dagger vs Toothpick: Round 1: Usage

```
openScope("APPLICATION").inject(this);
```

Toothpick

Dagger vs Toothpick: Round 1: Usage

```
Scope scope = openScope("APPLICATION");
scope.installModules(new BaseModule(context));
scope.inject(this);
```

Toothpick

Dagger vs Toothpick: Round 2: Setup

```
@Module
class BaseModule {

    BaseModule(Application context){}
    …
    @Provides
    public Tracker provideTracker() {
        return new GoogleTracker();
    }
}
```

Dagger

Dagger vs Toothpick: Round 2: Setup

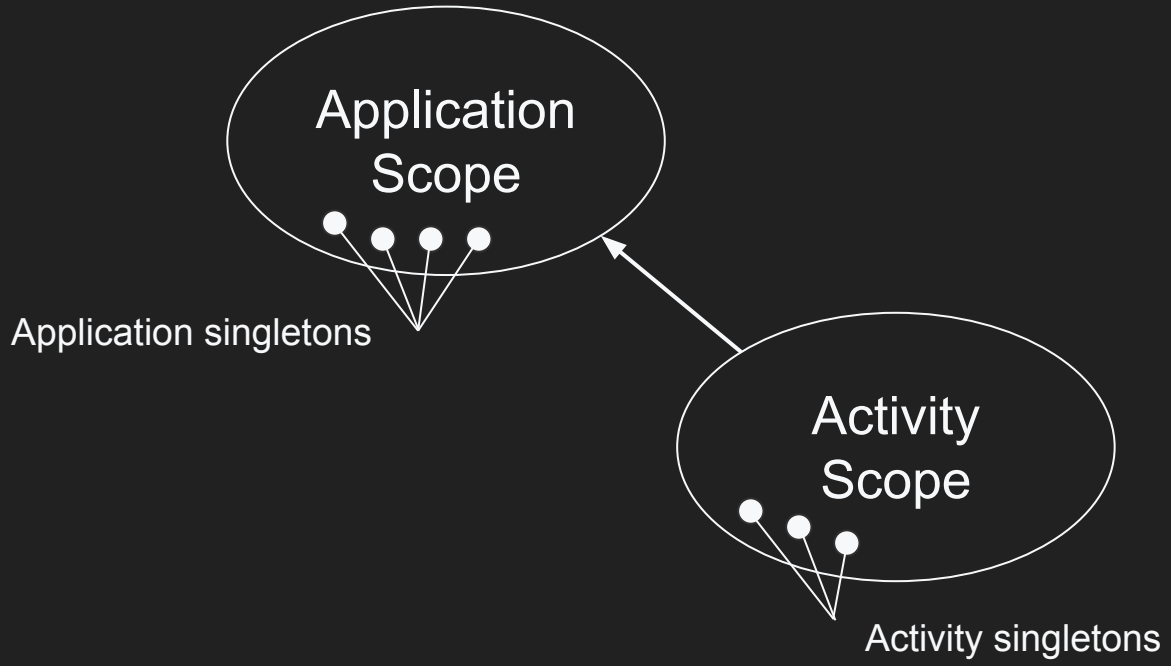
```
@Component(modules = {BaseModule.class})
interface AppComponent {
   void inject(MyActivity activity);
}
```

Dagger vs Toothpick: Round 2: Setup

```
class BaseModule extends Module {
  public BaseModule(Application context){
    bind(Tracker.class)
        .to(GoogleTracker.class);
```

Toothpick

Dagger vs Toothpick: Round 3: Scopes



Dagger vs Toothpick: Round 3: Scopes

```
@Scope
@Retention(RUNTIME)
public @interface ActivityScope {
}
```

Dagger vs Toothpick: Round 3: Scopes

```
@ActivityScope
@Subcomponent(modules = {ScopeModule.class})
interface ScopeComponent {
   void inject(ScopeActivity activity);
}
```

Dagger

Dagger vs Toothpick: Round 3: Scopes

```
@Module
static class ScopeModule {
...
   @Provides @ActivityScope
   public Activity provideActivity() {
      return activity;
   }
}
```

Dagger

Dagger vs Toothpick: Round 3: Scopes

```
@Component(modules = {BaseModule.class})
interface AppComponent {
   void inject(LonelyActivity activity);
   ScopeComponent plus(ScopeModule module);
}
```

Dagger

Dagger vs Toothpick: Round 3: Scopes

Scope verification

- Dagger: **compile-time**
- Toothpick: **runtime**

That’s
annotation porn!

Dagger vs Toothpick: Round 3: Scopes

```
@Override
public void onCreate() {
  super.onCreate();
   DaggerDependencies_AppComponent.builder()
       .baseModule(new BaseModule(this)).build()
       .plus(new ScopeModule(this))
       .inject(this)
```

Dagger

Dagger vs Toothpick: Round 3: Scopes

```
Scope scope =
  openScope("APPLICATION", "MY_ACTIVITY")

scope.installModules(new ScopeModule(this));
```

Toothpick

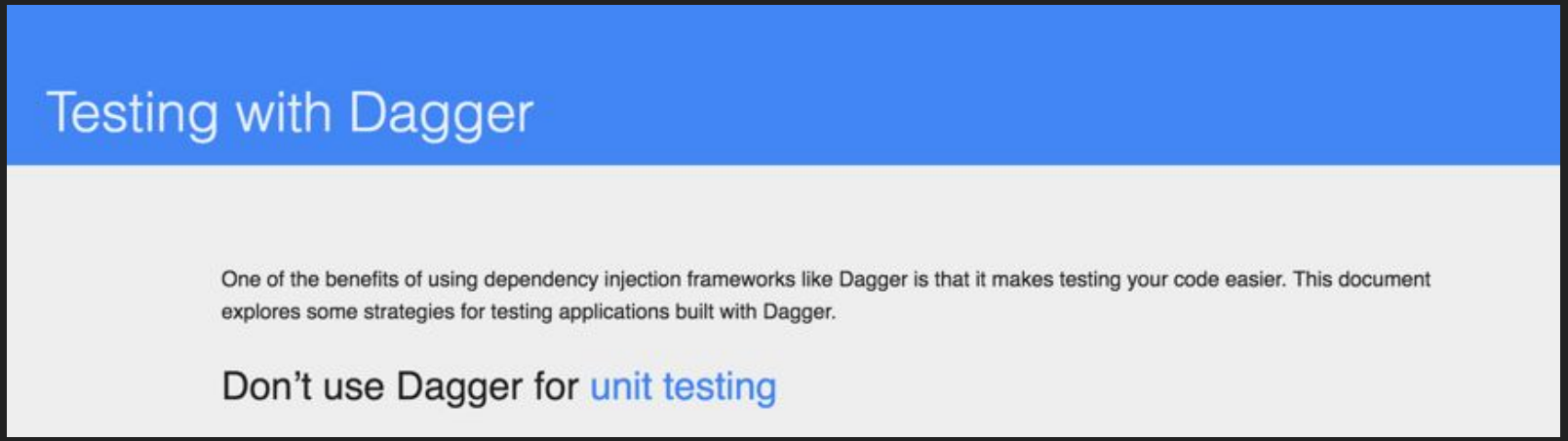
Dagger vs Toothpick: Round 3: Scopes

```
Scope scope =
  openScope("APPLICATION", "MY_ACTIVITY")

scope.installModules(new ScopeModule(this));
.inject(this)
```

Toothpick

Dagger vs Toothpick: Round 4: Tests



Dagger

Dagger vs Toothpick: Round 4: Tests

```
@Mock Tracker tracker;

class TestModule extends Dependencies.BaseModule {
      @Provides
      public Tracker provideTracker() {
         return tracker;
      }
   }
}
```

Dagger

Dagger vs Toothpick: Round 4: Tests

```
MyApplication.set(
    DaggerDependencies_AppComponent
        .builder()
        .baseModule(
            new TestModule()).build());
```

Dagger vs Toothpick: Round 4: Tests

```
@Mock Tracker tracker;

@Rule
public ToothPickRule toothPickRule =
  new ToothPickRule(
          this, "APPLICATION_SCOPE");
```

Toothpick

Dagger vs Toothpick: Round 4: Tests

```
@Mock Tracker tracker;
@Mock Navigator navigator;
@Mock Logger logger;

class TestModule extends Dependencies.BaseModule {

     @Provides
     public Tracker provideTracker() {
          return tracker;
     }

     @Provides
     public Navigator provideNavigator() {
          return tracker;
     }

     @Provides
     public Logger provideLogger() {
          return logger;
     }
}
```

Dagger

Dagger vs Toothpick: Round 4: Tests

```
@Mock Tracker tracker;
@Mock Navigator navigator;
@Mock Logger logger;

@Rule
public ToothPickRule toothPickRule =
  new ToothPickRule(
     this, "APPLICATION_SCOPE");
```

Toothpick

Dagger vs Toothpick: Round 5: Performance

Costs of creating a Component/Scope

- Dagger 1: 20 ms
- Dagger 2: 22 ms
- Toothpick: 1 ms

Dagger vs Toothpick: Round 5: Performance

Costs of usage with 1000 injections:

- Dagger 1: 33 ms
- Dagger 2: 31 ms
- Toothpick: 35 ms

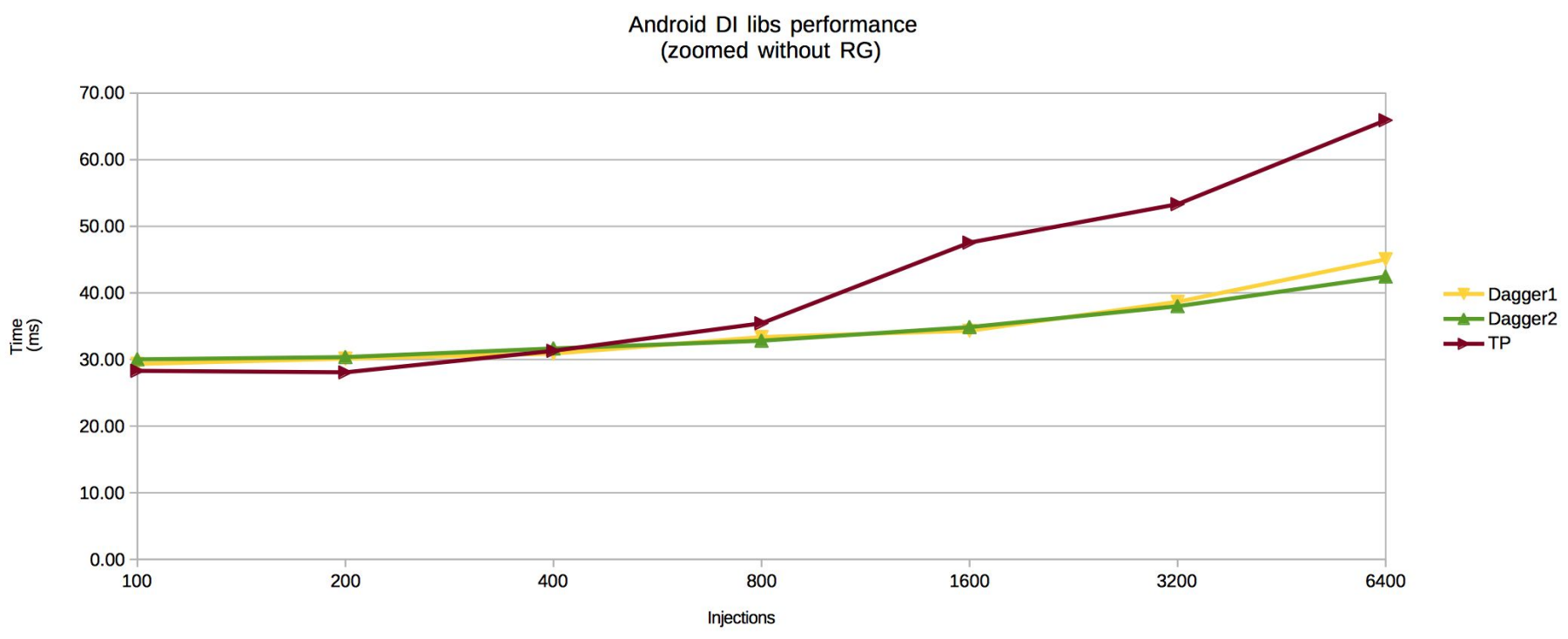
Dagger vs Toothpick: Round 5: Performance

Costs of usage with 6400 injections:

- Dagger 1: 45 ms
- Dagger 2: 42 ms
- Toothpick: 66 ms

Dagger vs Toothpick: Round 5: Performance

Costs of usage:

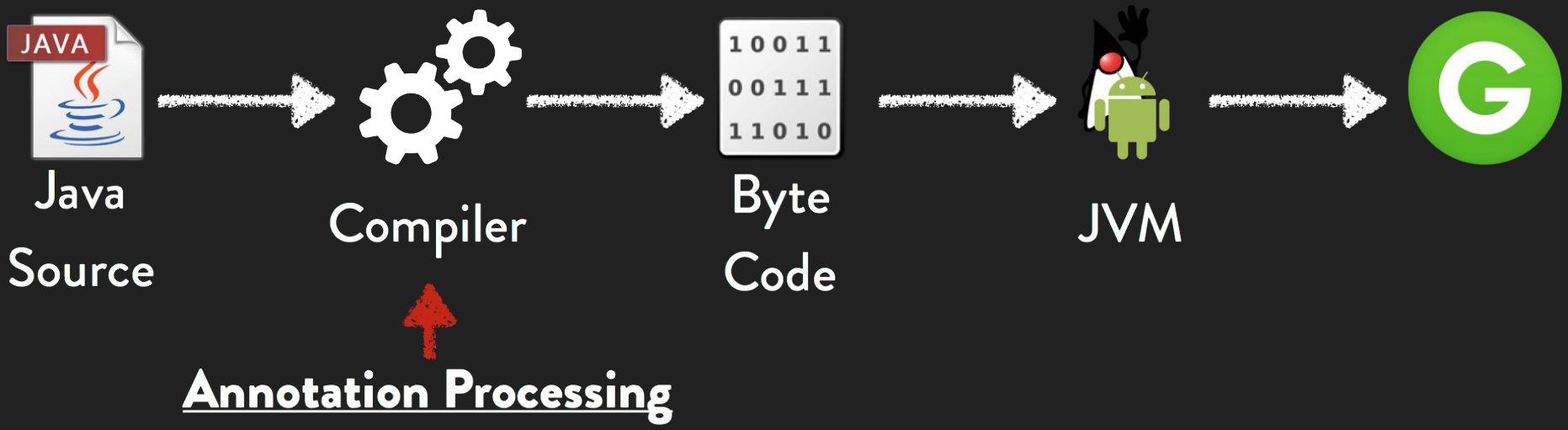


Dagger & Toothpick

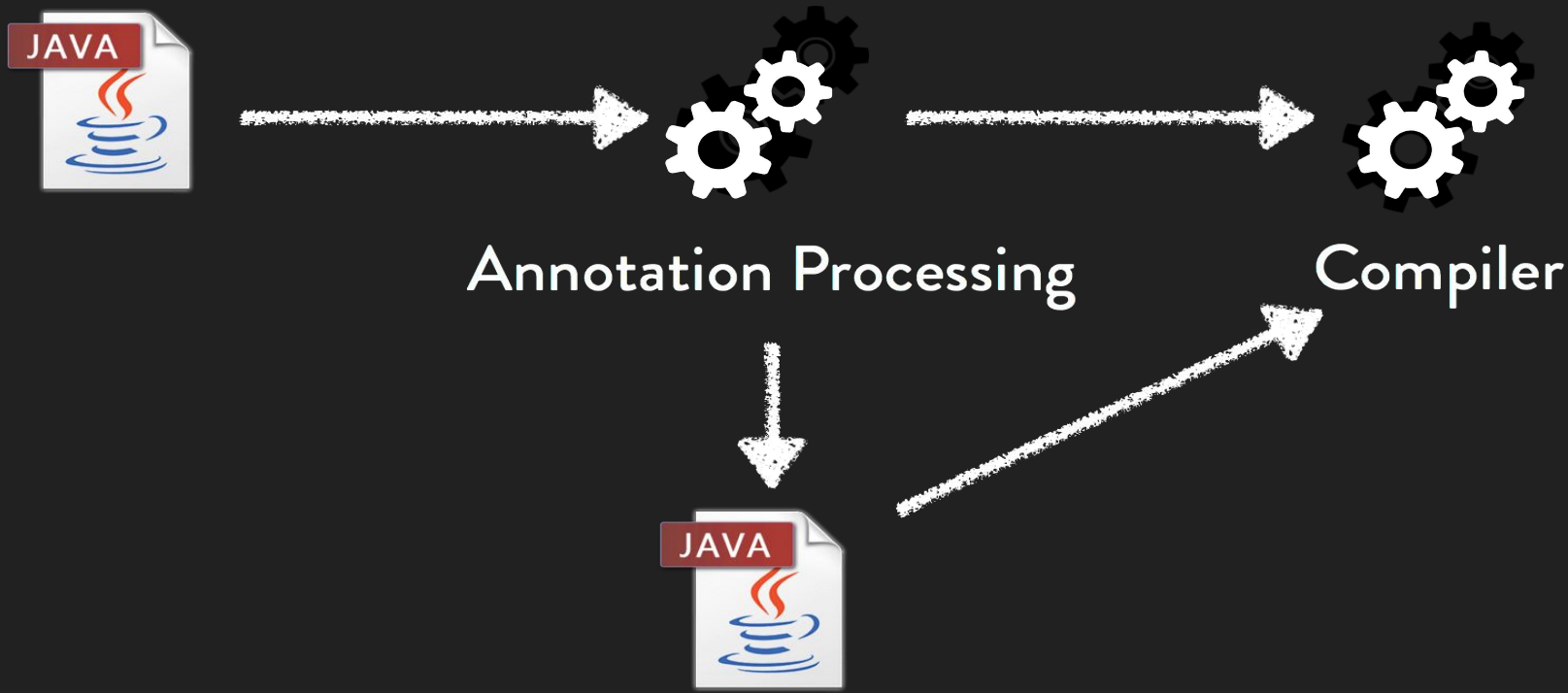
Let’s talk about the internals

Let’s talk about Annotation Processing

What is annotation processing ?



What is annotation processing ?



What is annotation processing ?

Annotation processing:

- is an API to get a view of java classes before they are compiled.
- is standard java.
- uses different concepts to represent classes (mirrors & TypeElements), methods & constructors (ExecutableElements), constructors, fields (Elements), annotations, etc.
- is not easy to use, not easy to debug, not easy to memorize and learn.
- annotation processors can generate code and/or resources.

What is annotation processing ?

```
TypeElement enclosingElement = (TypeElement)
                               element.getEnclosingElement();

Set<Modifier> modifiers = executableElement.getModifiers();
if (modifiers.contains(PRIVATE)) {
   ...
}
```

Which code is generated ?

```
@Module
class SlidesModule {
 @Provides
 DisplayOut displayOut(Resolution resolution){
    return new UcsbDisplayOut(resolution);
  }
}
```

Dagger

Which code is generated ?

```
@Generated
public final class SlidesModule_DisplayOutFactory
  implements Factory<DisplayOut>  {

  private final SlidesModule module;
  private final Provider<Resolution> resolutionProvider;

  public static SlidesModule_DisplayOutFactory create(
    SlidesModule module,
    Provider<Resolution> resolutionProvider) {..}

  @Override public DisplayOut get() {
    return module.displayOut(resolutionProvider.get()};
  }
}
```

Dagger

Which code is generated ?

```
class UcsbDisplayOut {
    @Inject
    UcsbDisplayOut(Resolution resolution) {
        ….
   }
}
```

Toothpick

Which code is generated ?

```
public final class UcsbDisplayOut$$Factory
  implements Factory<UcsbDisplayOut> {

  @Override
  public UcsbDisplayOut createInstance(Scope scope) {
   Resolution resolution = scope.getInstance(Resolution.class);
   return new UcsbDisplayOut(resolution);
  }
}
```

Toothpick

Which code is generated ?

Basically, both libs generate:

- Factories to create instances
- MemberInjectors to assign members

Moreover Dagger generates code for:

- Modules, Components

And Tootpick can also generate code for:

- Registries

Which code is generated ?

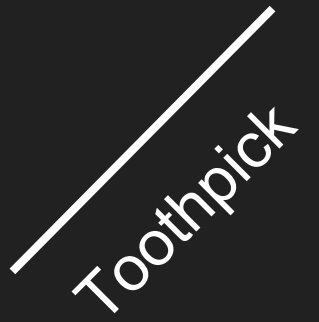
Dagger:

- generates a static graph,
- generated code only calls generated code
- very efficient
- but all wiring is static
- hard to modify for testing

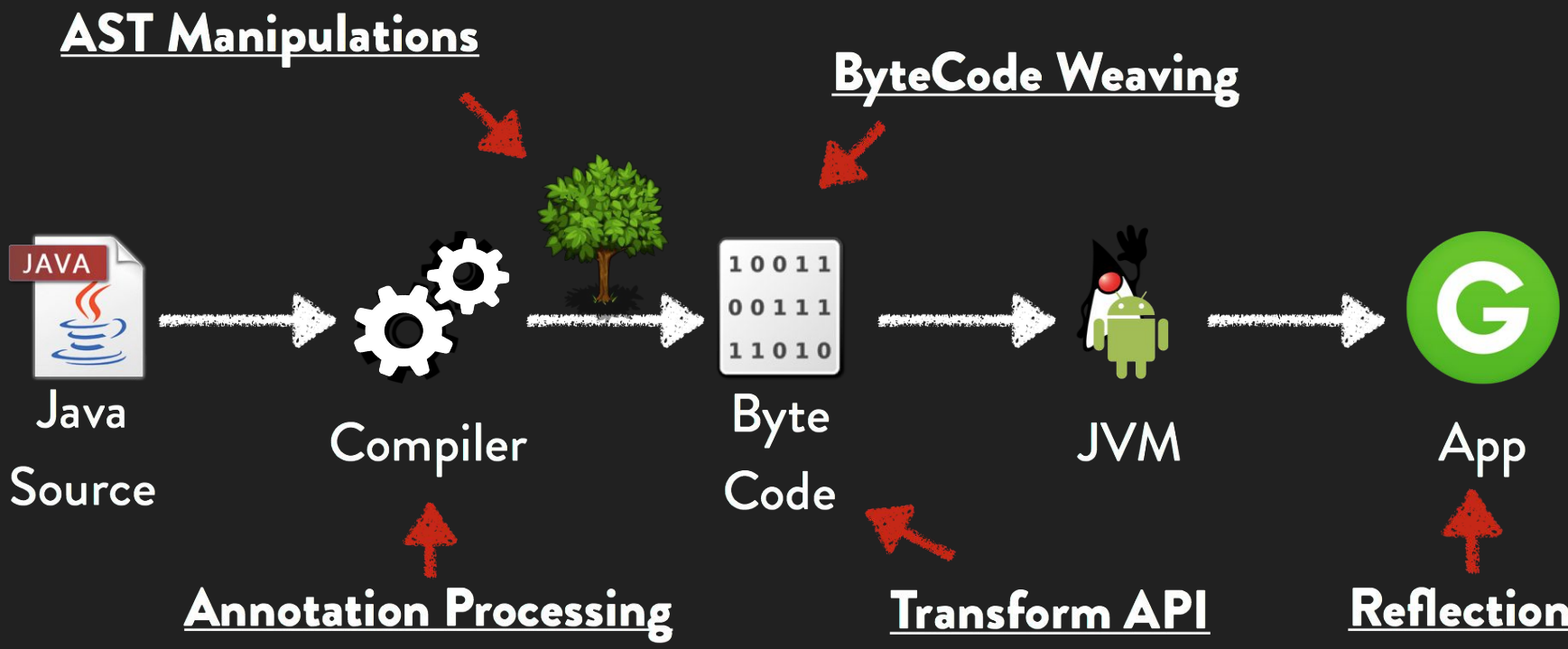
Which code is generated ?

Toothpick:

- generates a dynamic graph
- generated code calls runtime code to get the bindings
- a bit less efficient
- but more flexible
- easier to change for testing.



Alternatives to reflection & annotation Processing ?



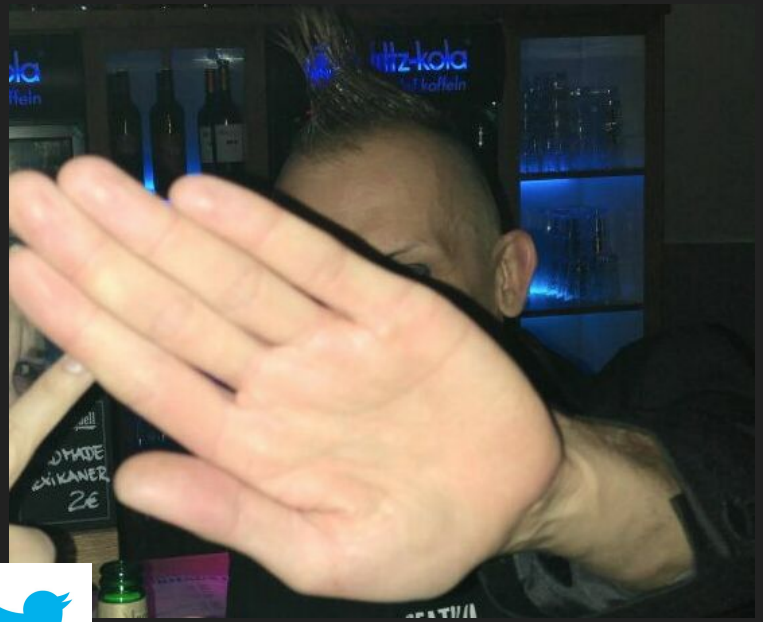
Dagger vs Toothpick: Overall

Conclusion:

- Dagger provides compile-time scope verification
- Dagger might be a little more efficient

- Toothpick avoids boilerplate code
- Toothpick is easier for testing
- Toothpick scopes are more clear

Thank you, see you tomorrow



@PreusslerBerlin

+stephane nicolas

stephanenicolas / **toothpick** ★ Star