组会分享
——依赖注入与生命周期

李青蓝

2024.11.28

Spring 01

Guice 02

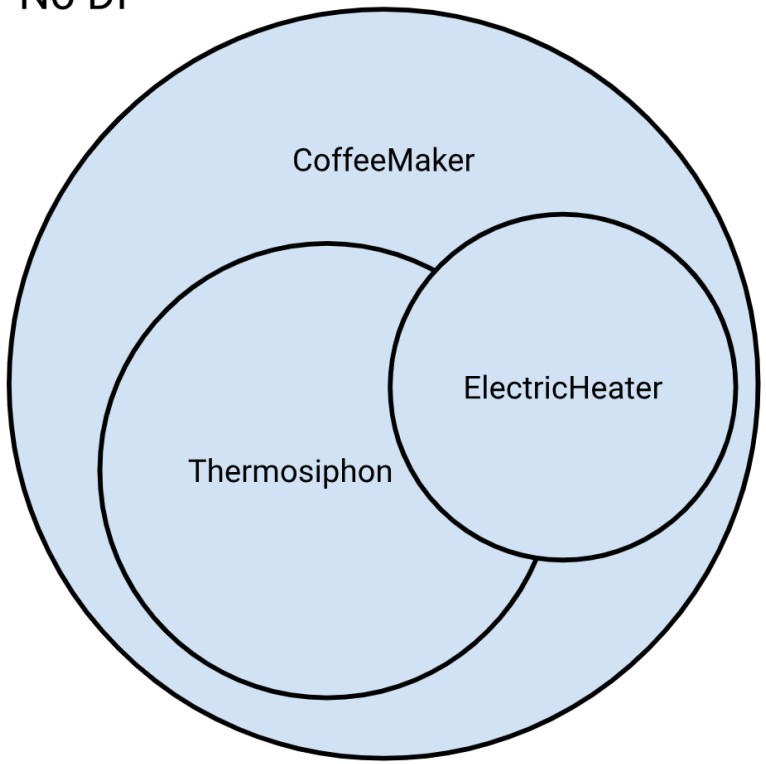
Dagger 03

Hilt 04

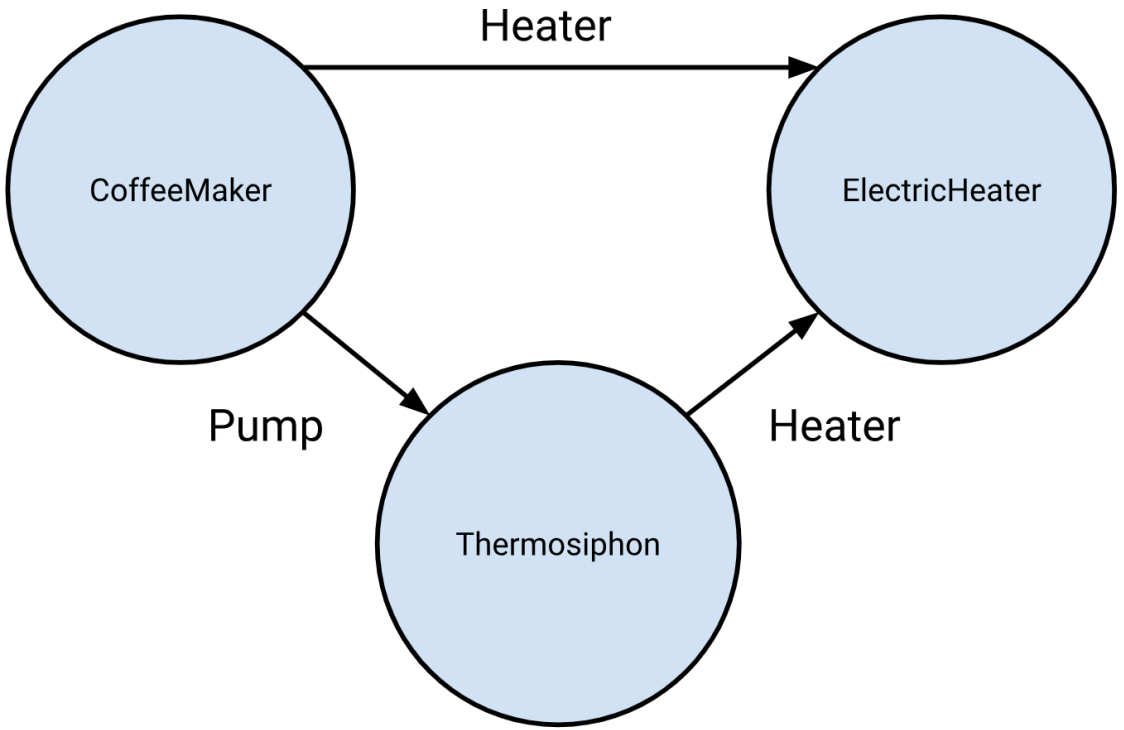
```
class CoffeeMaker {
  private final Heater heater;
  private final Pump pump;

  CoffeeMaker() {
    this.heater = new ElectricHeater();
    this.pump = new Thermosiphon(heater);
  }

  Coffee makeCoffee() {/* … */}
}

class CoffeeMain {
  public static void main(String[] args) {
    Coffee coffee =
      new CoffeeMaker().makeCoffee();
  }
}
```

```java
class CoffeeMaker {
  private final Heater heater;
  private final Pump pump;

  CoffeeMaker(Heater heater, Pump pump) {
    this.heater = checkNotNull(heater);
    this.pump = checkNotNull(pump);
  }

  Coffee makeCoffee() {/* … */}
}

class CoffeeMain {
  public static void main(String[] args) {
    Heater heater = new ElectricHeater();
    Pump pump = new Thermosiphon(heater);
    Coffee coffee =
      new CoffeeMaker(heater, pump).makeCoffee();
  }
}
```

No DI


CoffeeMaker
ElectricHeater
Thermosiphon

DI


Heater
CoffeeMaker
ElectricHeater
Pump
Heater
Thermosiphon

```
DependencyRequest.Factory dependencyRequestFactory =
    new DependencyRequest.Factory(elements, types, keyFactory);
ProvisionBinding.Factory provisionBindingFactory =
    new ProvisionBinding.Factory(elements, types, keyFactory,
dependencyRequestFactory);
InjectionSite.Factory injectionSiteFactory =
    new InjectionSite.Factory(dependencyRequestFactory);
ComponentDescriptor.Factory componentDescriptorFactory =
    new ComponentDescriptor.Factory(elements, types, injectBindingRegistry,
        provisionBindingFactory, dependencyRequestFactory);
```

01

Spring

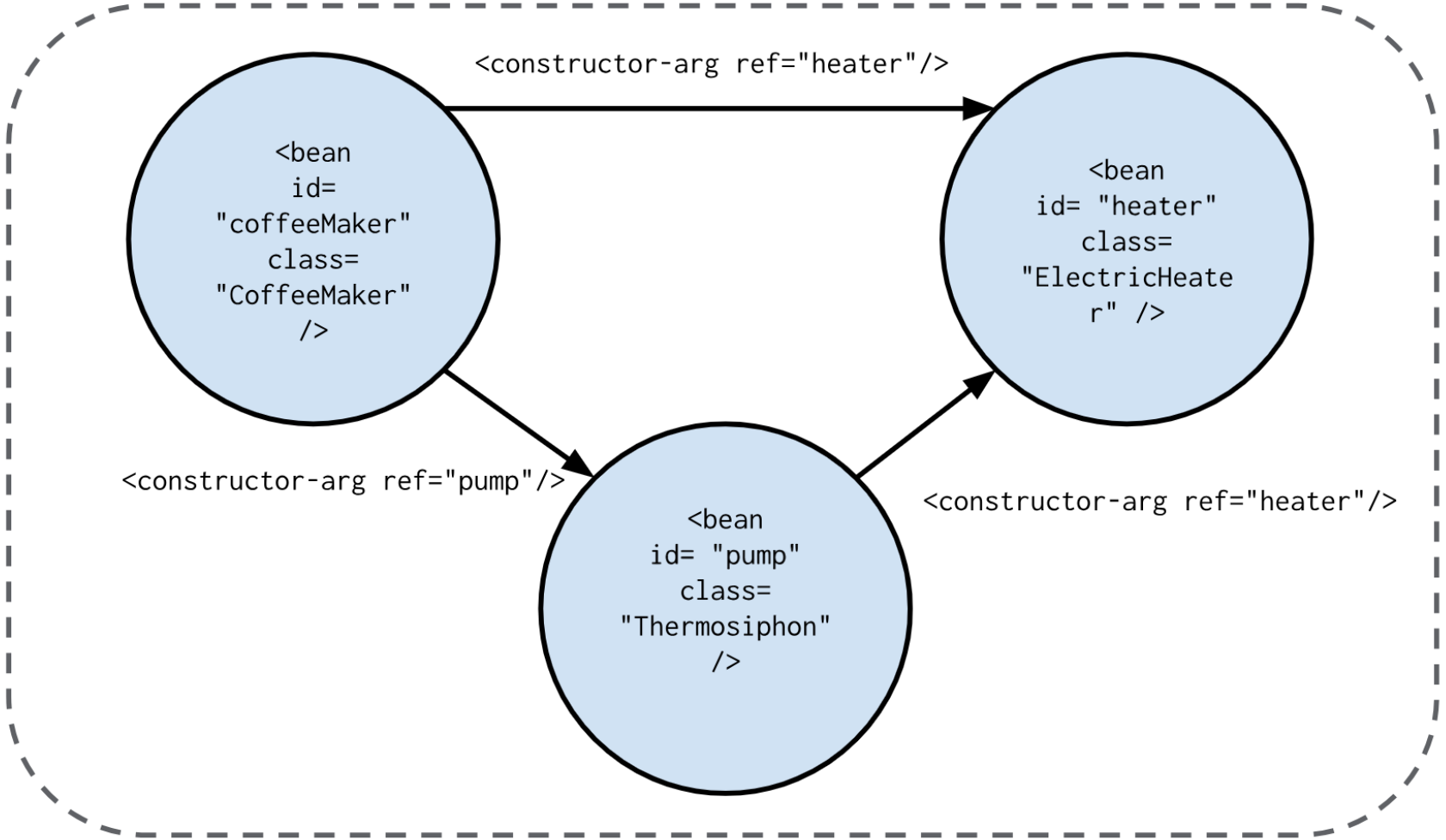
Spring 1 & 2

- Popularized Dependency Injection / Inversion of Control / Hollywood Principle
- Solved initialization ordering
- DI-friendly objects remain *exactly* as written and the configuration for composing them is written as XML

Spring

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean id="coffeeMaker" class="CoffeeMaker">
    <constructor-arg ref="heater"/>
    <constructor-arg ref="pump"/>
  </bean>
  <bean id="heater" class="ElectricHeater">
    <constructor-arg ref="pump"/>
  </bean>
  <bean id="pump" class="Thermosiphon"/>
</beans>
```

BeanFactory



Solutions:

- Initialization *ordering*
- Instance management (scoping)

Issues:

- The XML is almost as verbose as the Java
- Runtime validation of configuration
- Runtime validation of the graph
- Untraceable application flow
- Map-like API

02

Guice

Guice

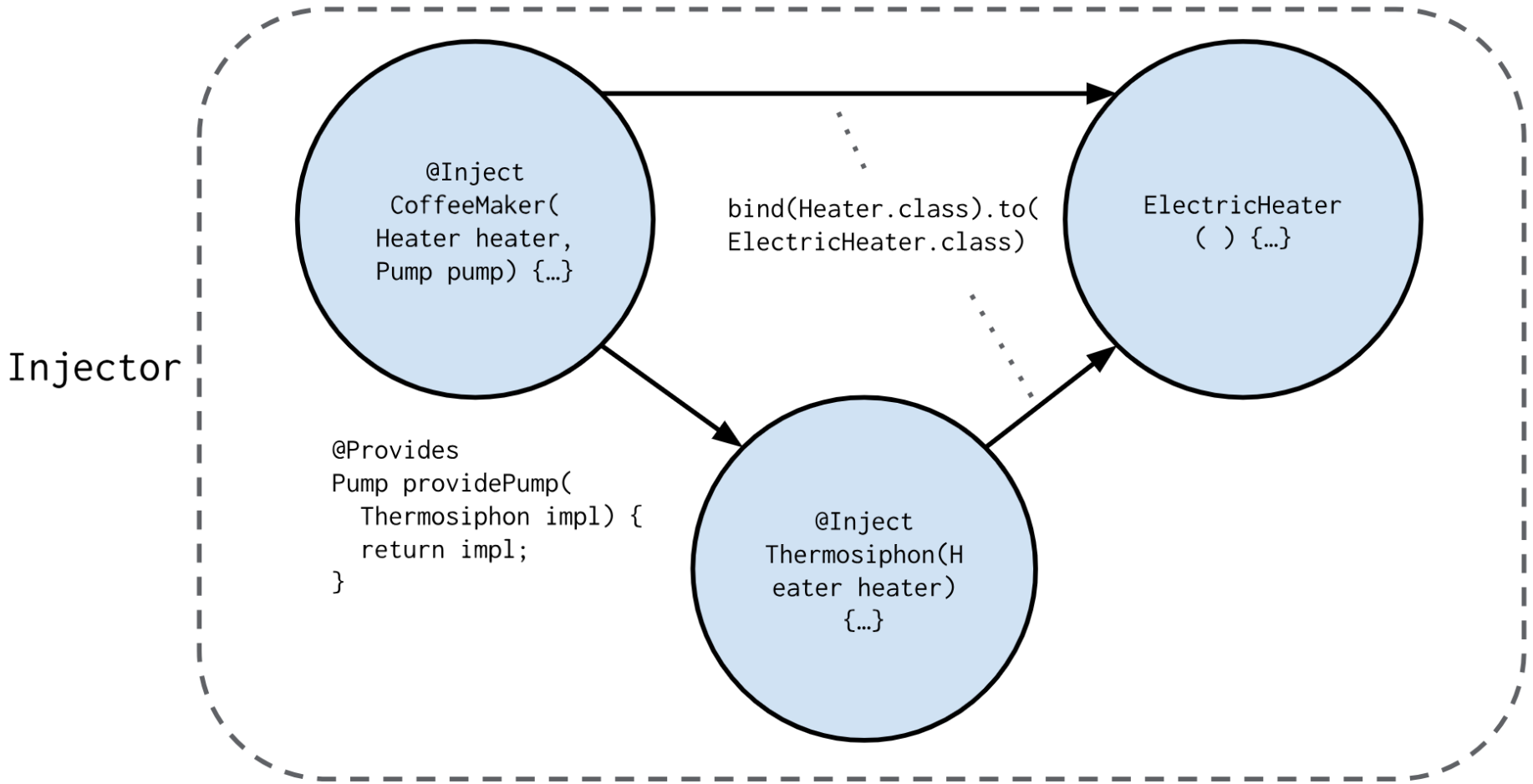
- New configuration options enabled by Java 1.5 language features
- Decreased and decentralized configuration
- Entirely written and executed in Java

Guice Configuration

```
class CoffeeMaker {
  private final Heater heater;
  private final Pump pump;

  @Inject CoffeeMaker(Heater heater, Pump pump) {
    this.heater = checkNotNull(heater);
    this.pump = checkNotNull(pump);
  }

  Coffee makeCoffee() {/* … */}
}
```

Guice Configuration

```
public class DripCoffeeModule extends AbstractModule {
 @Override protected void configure() {
   bind(Heater.class).to(ElectricHeater.class);
 }

 @Provides Pump providePump(Thermosiphon impl) {
   return impl;
 }
}
```


Injector
@Inject
CoffeeMaker(
Heater heater,
Pump pump) {…}
bind(Heater.class).to(
ElectricHeater.class)
ElectricHeater
( ) {…}
@Provides
Pump providePump(
Thermosiphon impl) {
return impl;
}
@Inject
Thermosiphon(H
eater heater)
{…}

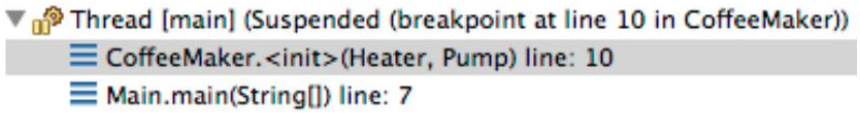
What's the problem?
What happens if we forget a binding?

```
Exception in thread "main" com.google.inject.ConfigurationException: Guice configuration errors:

1) No implementation for com.google.whiteboard.guice.Pump was bound.
  while locating com.google.whiteboard.guice.Pump
    for parameter 1 at com.google.whiteboard.guice.CoffeeMaker.<init>(CoffeeMaker.java:9)
  while locating com.google.whiteboard.guice.CoffeeMaker

1 error
      at com.google.inject.internal.InjectorImpl.getProvider(InjectorImpl.java:1004)
      at com.google.inject.internal.InjectorImpl.getProvider(InjectorImpl.java:961)
      at com.google.inject.internal.InjectorImpl.getInstance(InjectorImpl.java:1013)
      at com.google.whiteboard.guice.Main.main(Main.java:7)
```

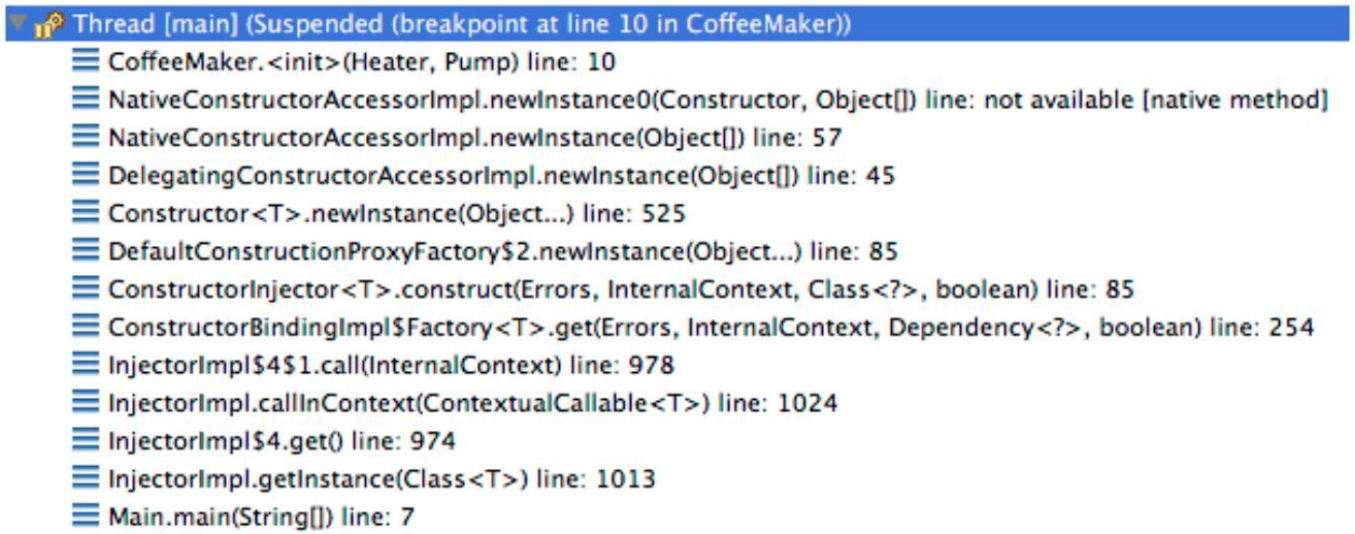
What happens if you declare a binding twice?

```
Exception in thread "main" com.google.inject.CreationException: Guice creation errors:

1) A binding to com.google.whiteboard.guice.Pump was already configured at
com.google.whiteboard.guice.DripCoffeeModule.configure(DripCoffeeModule.java:10).
  at com.google.whiteboard.guice.DripCoffeeModule.providePump(DripCoffeeModule.java:20)

1 error
      at com.google.inject.internal.Errors.throwCreationExceptionIfErrorsExist(Errors.java:435)
      at com.google.inject.internal.InternalInjectorCreator.initializeStatically(InternalInjectorCreator.java:154)
      at com.google.inject.internal.InternalInjectorCreator.build(InternalInjectorCreator.java:106)
      at com.google.inject.Guice.createInjector(Guice.java:95)
      at com.google.inject.Guice.createInjector(Guice.java:72)
      at com.google.inject.Guice.createInjector(Guice.java:62)
      at com.google.whiteboard.guice.Main.main(Main.java:7)
```

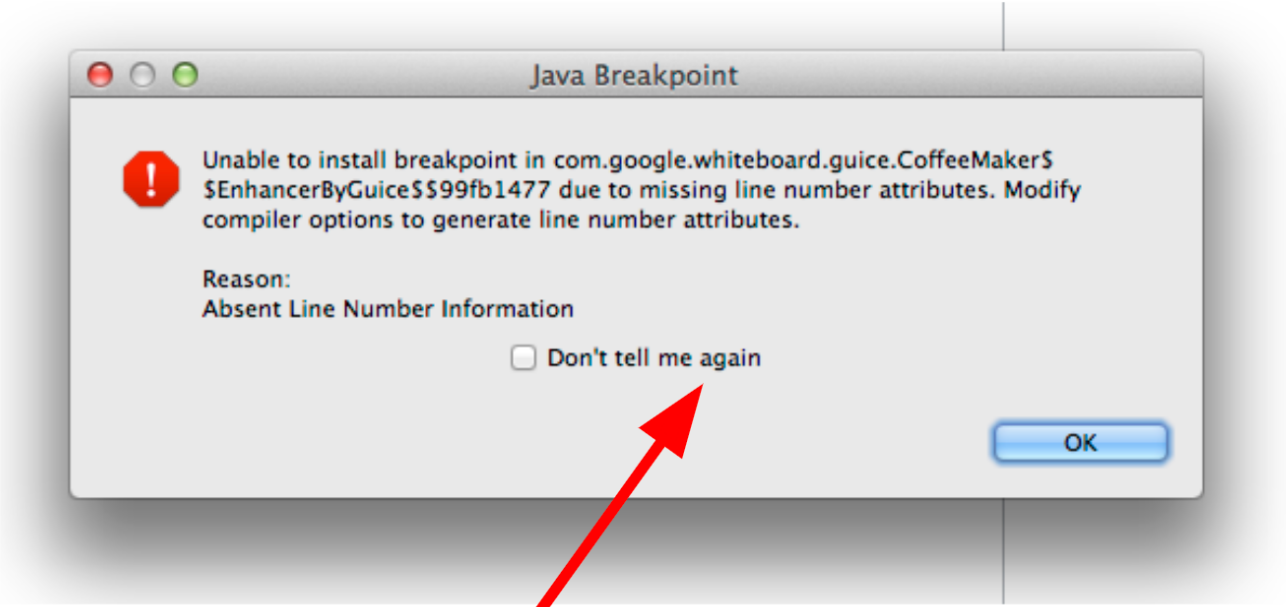
How do we debug?



Becomes

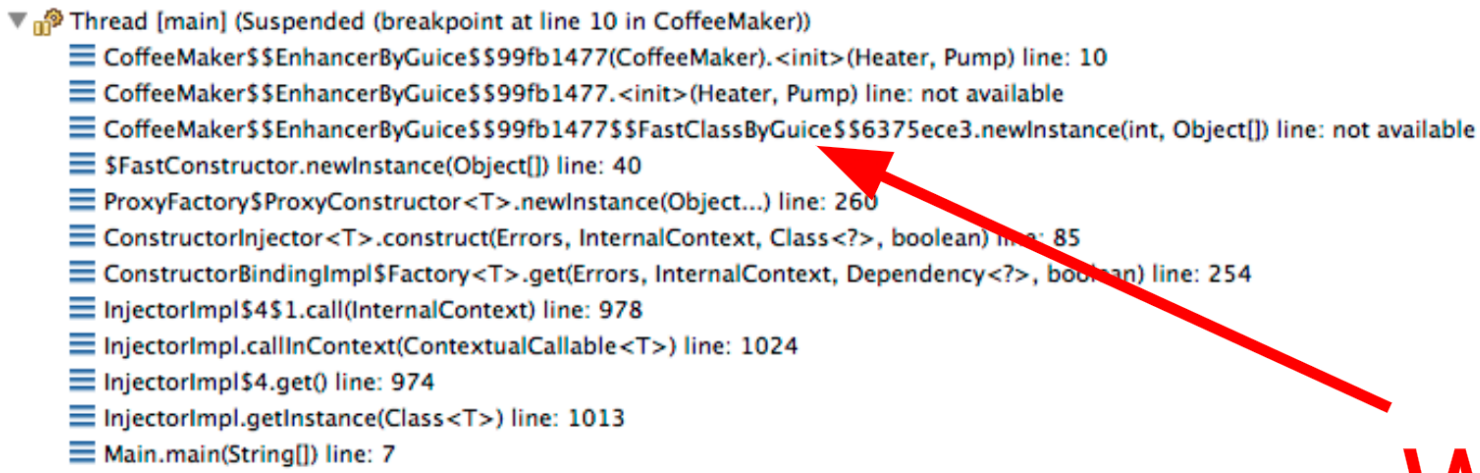


Or worse yet, if you enable the wrong feature…



You'd better click that, or you're going to see this message a lot

And stack traces look like…



WAT?

With lots of…

There may not be source to find.

But worst of all, code becomes untraceable.

How do I know which implementation of `Heater` will be used?
Well, let's find the places that call the constructor?

'com.google.whiteboard.guice.CoffeeMaker.CoffeeMaker(Heater, Pump)' - 0 references in workspace (no JRE) (0 matches filtered from view)

Guice

Solutions:

- Binding discovery
- Greatly reduced configuration
- Configuration near configured code
- Pure Java configuration

Issues:

- Runtime validation of the graph
- Untraceable application flow
- Synthetic classes
- Map-like API

03

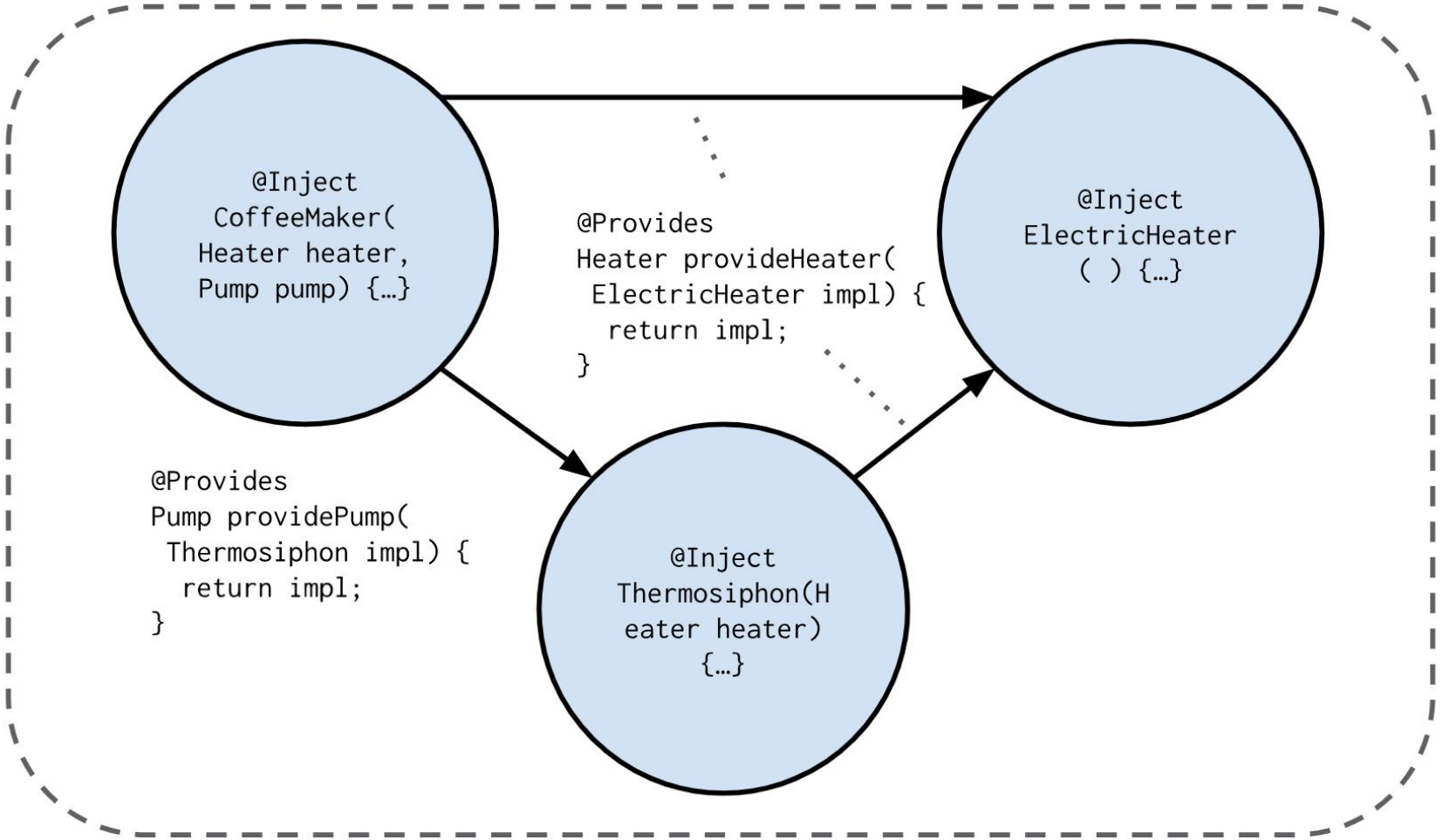
Dagger

Dagger 1

- Configuration API is basically the *statically inspectable* API from Guice
- Reduce features to streamline only to the safest, most effective featureset
- Move portions of the workload from runtime to compile time
  - Error reporting at compile time without special tooling
  - Efficient provision
- Take the reflection overhead out of provision

Dagger 1

ObjectGraph



What happens if we forget a binding?

```
public class DripCoffeeModule {
```

No binding for com.google.whiteboard.dagger.Pump required by com.google.whiteboard.dagger.CoffeeMaker for com.google.whiteboard.dagger.DripCoffeeModule

…says javac (and/or Eclipse).

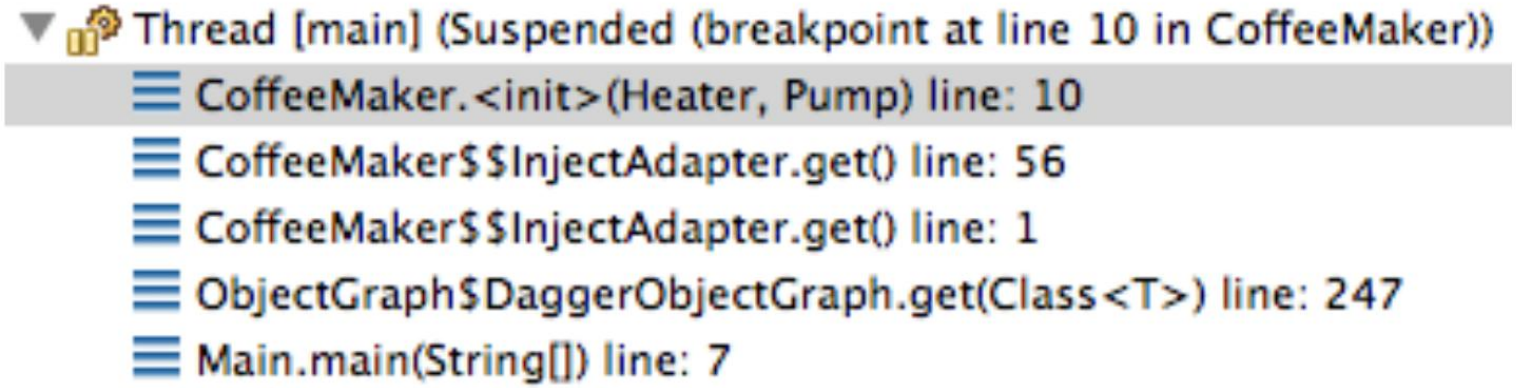
What happens if you declare a binding twice?

```
@Provides Pump providePump(Thermosiphon thermosiphon) {
  return thermosiphon;
}
```

Duplicate bindings for com.google.whiteboard.dagger.Pump:
com.google.whiteboard.dagger.DripCoffeeModule.provideOtherPump()
com.google.whiteboard.dagger.DripCoffeeModule.providePump()

...says javac (and/or Eclipse).

And debugging?



That looks saner

- Not quite 2 stack frames, but much better than 13
- Inject adapters have names meant for humans
- The `$$` classes are generated, but still have sources available

Dagger 1 - The generated code

```
public final class CoffeeMaker$$InjectAdapter extends Binding<CoffeeMaker>
    implements Provider<CoffeeMaker> {
  private Binding<Heater> heater;
  private Binding<Pump> pump;

  public CoffeeMaker$$InjectAdapter() {
    super("com.google.whiteboard.dagger.CoffeeMaker", "members/com.google.whiteboard.dagger.CoffeeMaker", NOT_SINGLETON, CoffeeMaker.class);
  }

  @Override @SuppressWarnings("unchecked") public void attach(Linker linker) {
    heater = (Binding<Heater>) linker.requestBinding("com.google.whiteboard.dagger.Heater", CoffeeMaker.class, getClass().getClassLoader());
    pump = (Binding<Pump>) linker.requestBinding("com.google.whiteboard.dagger.Pump", CoffeeMaker.class, getClass().getClassLoader());
  }

  @Override public void getDependencies(Set<Binding<?>> getBindings, Set<Binding<?>> injectMembersBindings) {
    getBindings.add(heater);
    getBindings.add(pump);
  }

  @Override public CoffeeMaker get() {
    CoffeeMaker result = new CoffeeMaker(heater.get(), pump.get());
    return result;
  }
}
```

Dagger 1

Solutions:

- Compile-time validation of *portions* of the graph
- Easy debugging; entirely concrete call stack
- Highly efficient provision

Issues:

- Ugly generated code
- Runtime graph composition
- Inefficient graph creation
- Partial traceability
- Map-like API

Dagger 2 - Focus on the developer experience

- Traceability
  - Navigate the entire application with “find usages” and “open declaration”
  - Trace through code that is clear and well-structured
- Clarify the API
  - Simpler `@Module` declarations
  - No more “maps”
- Performance
  - As fast as hand-written code
  - No coding around the framework

Dagger 2 - The strategy

- Keep (more-or-less) the Dagger API for configuration. It works.
- Generate simple Java at compile time *for the entire stack* that looks as close to hand-written DI code as possible
- Validate **all** framework/configuration preconditions at compile time
- Unify user-defined types with generated types through limited, well-defined entry points

Dagger 2 - The implementation

The standard DI API (JSR 330) declares exactly 1 interface:

```
public interface Provider<T> {
  T get();
}
```

E.g.:

```
Provider<CoffeeMaker> coffeeMakerProvider = …;
CoffeeMaker coffeeMaker = coffeeMakerProvider.get();
```

Part 1: Generate the providers (implement JSR 330)

```
public final class CoffeeMaker$$Factory implements Provider<CoffeeMaker> {
 private final Provider<Heater> heaterProvider;
 private final Provider<Pump> pumpProvider;

 public CoffeeMaker$$Factory(
       Provider<Heater> heaterProvider,
       Provider<Pump> pumpProvider) {
   this.heaterProvider = heaterProvider;
   this.pumpProvider = pumpProvider;
 }

 public CoffeeMaker get() {
   return new CoffeeMaker(heaterProvider.get(), pumpProvider.get());
 }
}
```

Part 1: Generate the providers (implement JSR 330)

```
public final class Pump$$Factory implements Provider<Pump> {
 private final PumpModule module;
 private final Provider<Thermosiphon> thermosiphonProvider;

 public Pump$$Factory(
       PumpModule module,
       Provider<Thermosiphon> thermosiphonProvider) {
   this.module = module;
   this.thermosiphonProvider = thermosiphonProvider;
 }

 public Pump get() {
   return module.providePump(thermosiphonProvider.get());
 }
}
```

The Client API

Spring:

```
BeanFactory beanFactory =
    new XmlBeanFactory(new ClassPathResource("beans.xml"));
CoffeeMaker coffeeMaker = (CoffeeMaker) beanFactory.getBean("coffeeMaker");
```

Guice:

```
Injector injector =
    Guice.createInjector(new DripCoffeeModule());
CoffeeMaker coffeeMaker = injector.getInstance(CoffeeMaker.class);
```

Dagger 1:

```
ObjectGraph objectGraph =
    ObjectGraph.create(DripCoffeeModule.class);
CoffeeMaker coffeeMaker = objectGraph.get(CoffeeMaker.class);
```

Dagger 2 - The @Component

```
@Component(modules = DripCoffeeModule.class)
interface CoffeeMakerComponent {
 CoffeeMaker getCoffeeMaker();
}


CoffeeMakerComponent component =
    Dagger_CoffeeMakerComponent.create();
CoffeeMaker coffeeMaker = component.getCoffeeMaker();
```

Part 2: Generate the `@Component` implementation

```
public final class Dagger_CoffeeMakerComponent implements CoffeeMakerComponent {
  private final Provider<ElectricHeater> electricHeaterProvier;
  private final Provider<Heater> heaterProvider;
  private final Provider<Thermosiphon> thermosiphonProvider;
  private final Provider<Pump> pumpProvider;
  private final Provider<CoffeeMaker> coffeeMakerProvider;

  private Dagger_CoffeeMakerComponent(DripCoffeeModule dripCoffeeModule) {
    this.electricHeaterProvider = new ElectricHeater$$Factory();
    this.heaterProvider = new DripCoffeeModule$$ProvideHeaterFactory(dripCoffeeModule, electricHeaterProvider);
    this.thermosiphonProvider = new Thermosiphon$$Factory(heaterProvider);
    this.pumpProvider = new DripCoffeeModule$$PumpFactory(dripCoffeeModule,
        dripCoffeeModulethermosiphonProvider);
    this.coffeeMakerProvider = new DripCoffeeModule$$PumpFactory(heaterProvider, pumpProvider);
  }

  @Override public CoffeeMaker getCoffeeMaker() {
    return coffeeMakerProvider.get();
  }
}
```

Dagger 2

Solutions:

- Compile-time validation of the **entire** graph
- Easy debugging; entirely concrete call stack for provision *and* creation
- Fully traceable
- POJO API
- Performance

Issues:

- Less flexible
- No dynamism
- No automated migration path from Guice

04

Hilt

Dagger（匕首）

复杂、灵活、难用

Java平台

Hilt（刀柄）

简单、固定、易用

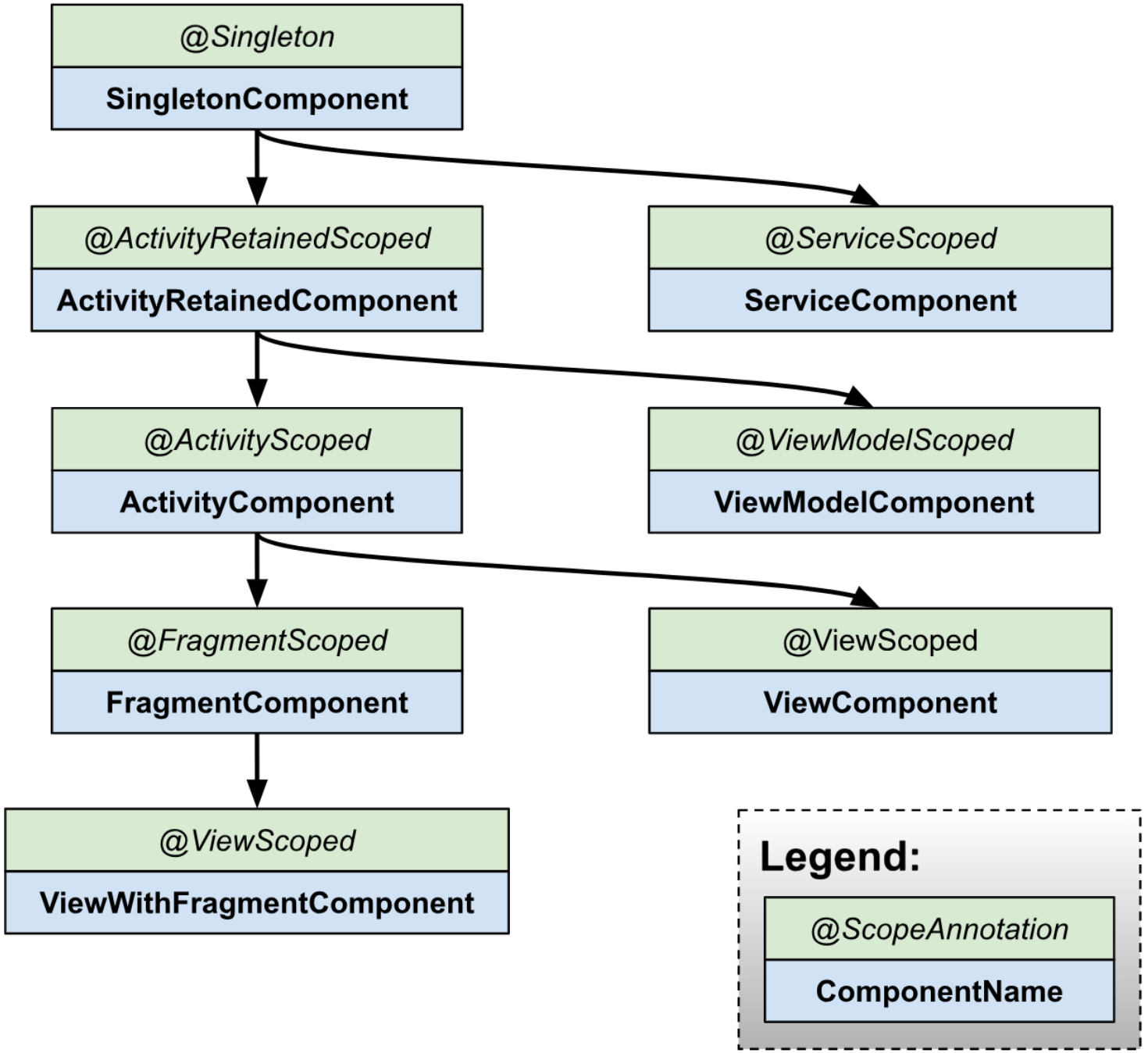
Android平台

Reduced boilerplate

The goal of Hilt is to enable users to focus on the Dagger binding definitions and usages without needing to worry about the rest of the Dagger setup. This means hiding things like component definitions with module and interface lists, code to create and hold on to components at the right points in the lifecycle, interfaces and casts to get the parent component, etc.

Standardization

Hilt standardizes the component hierarchy. This means that libraries that integrate with Hilt can easily add or consume bindings from these known components. This allows for more complex libraries to be built that can integrate cleanly and more simply into any Hilt app.


@Singleton
SingletonComponent
@ActivityRetainedScoped
ActivityRetainedComponent
@ServiceScoped
ServiceComponent
@ActivityScoped
ActivityComponent
@ViewModelScoped
ViewModelComponent
@FragmentScoped
FragmentComponent
@ViewScoped
ViewComponent
@ViewScoped
ViewWithFragmentComponent
Legend:
@ScopeAnnotation
ComponentName

Component	Scope	Created at	Destroyed at
SingletonComponent	@Singleton	Application#onCreate()	Application process is destroyed
ActivityRetainedComponent	@ActivityRetainedScoped	Activity#onCreate()[1]	Activity#onDestroy()[1]
ViewModelComponent	@ViewModelScoped	ViewModel created	ViewModel destroyed
ActivityComponent	@ActivityScoped	Activity#onCreate()	Activity#onDestroy()
FragmentComponent	@FragmentScoped	Fragment#onAttach()	Fragment#onDestroy()
ViewComponent	@ViewScoped	View#super()	View destroyed
ViewWithFragmentComponent	@ViewScoped	View#super()	View destroyed
ServiceComponent	@ServiceScoped	Service#onCreate()	Service#onDestroy()

感谢观看！