First I am taking about we all love independency , We don’t want to dependent on others generally .But there some situations we need that we have to work together or we need some dependency on other . where we have some dependency we need to create some good ideas of doing work .

Same we talking about the classes in oops

If we consider two classes :

```
 public class A {

 private B b;

 public A(){

 b  = new B();

 }

}
```

A---B

That means it is dependency of classes .. it may be more than two classes .

What be the idea : DI

Instead of using new to instantiate the B inside User’s constructor we can pass the B as an argument to the User’s constructor...

```
public class A{

        private B b;

            public A(B b ){

                this.b = b;

        }

}
```

The term “Dependency Injection” or in short “DI” is independent of Android. It is a design pattern in Software Engineering.

So this is What We call as Dependency Injection. Means supplying the dependencies from outside the class. And more specifically for the above code, we can consider it as a Constructor Injection.

Dependency injection is quite interesting Why

Any business application is made up of two or more classes and these classes collaborate with each other to perform some operation. Traditionally, we create the instance of dependent class object and do the required operations. When applying Dependency Injection, the objects are given their dependencies at creation time by some external entity that coordinates each object in the system. It means dependencies are injected into objects.

Most dependency injectors rely on reflection to create and inject dependencies. **Reflection** is awesome but is very slow and time consuming. Also it perform dependency resolution at runtime which leads unexpected errors and crashes the application

On the other hand some injectors uses a **Pre-compiler** that creates all the classes (object graph) it needs to work using Annotation Processor. An Annotation Processor is a way to read the compiled files during build time to generate source code files to be used in the project. So it perform the dependency resolution before the application runs and avoid unexpected errors

Dependency Injection Modes

There are three common means for a client to accept a dependency injection:

- Constructor based injection
- Setter method
- Interface

Dependency Injection reduces to write boiler plate code and make the development process smooth. Implementing proper dependency injection in our apps allows us to have:

- Loose coupling
- Easily testable code

- Code reusability

Popular Libraries comparison

The available popular dependency injection libraries for Android are:

- RoboGuice
- ButterKnife
- Dagger
- Android Annotation

Dagger

Dagger is designed for low-end devices. It is based on the Java Specification Request (JSR) 330. It uses code generation and is based on annotations. The generated code is very relatively easy to read and debug.

Dagger 2 uses the following annotations:

- @Module and @Provides: define classes and methods which provide dependencies
- @Inject: request dependencies. Can be used on a constructor, a field, or a method
- @Component: enable selected modules and used for performing dependency injection
-

Using dependency injection framework may be attractive because it simplify the code we write and also provide an adaptive environment that’s useful for testing and other configuration changes.

Dagger 2 is a compile-time android dependency injection framework and uses the Java Specification Request (JSR) 330 and uses an annotation processor.

Following are the basic annotations used in Dagger 2:

@Module : This is used on the class that does the work of constructing objects that'll be eventually provided as dependencies.

@Provide : This is used on the methods inside the Module class that'll return the object.

@Inject : This is used upon a constructor, field or a method and indicates that dependency has been requested.

@Component : The Module class doesn't provide the dependency directly to the class that's requesting it. For this a Component interface is used that acts like a bridge between @Module and @Inject.

@Singleton : This indicates that only a single instance of the dependency object would be created.

```
compile 'com.google.dagger:dagger:2.10'
annotationProcessor 'com.google.dagger:dagger-compiler:2.10'
```

Dagger is a fully static, compile-time dependency injection framework for both Java and Android.

The two classes that use each other are termed as 'coupled'. The coupling between classes are not binary—it's either 'tight'(strong) or 'loose'(weak)

Why dependencies are bad?

When there are many dependencies floating around the class, it leads to hard dependency problems—which are bad because of the following reasons

- Hard dependencies reduce the reusability
- Hard dependencies make it hard for testing
- Hard dependencies hinders code maintainability when the project scales up

What we need :

When classes and methods are loosely or not coupled or not dependant on many things, it's reusable nature increases. Reusability is one of the core commandment of object-oriented programming.

One more issue :

if there are so many dependencies within the method or the class, it will be hard to test

If a java class creates an instance of another class via the `new` operator, then it cannot be used and tested independently from that class.

Dependencies are of many types, the common ones are

- Class dependency
- Interface dependency

- Method/field dependency
- Direct and indirect dependency

Dagger is a replacement for these `FactoryFactory` classes that implements thedependency injection design pattern without the burden of writing the boilerplate. It allows you to focus on the interesting classes. Declare dependencies, specify how to satisfy them, and ship your app. Dependency injection isn't just for testing. It also makes it easy to create **reusable, interchangeable modules**.

Dagger constructs instances of your application classes and satisfies their dependencies.

Dragge User Guide Ex

```
class Thermosiphon implements Pump {
  private final Heater heater;

  @Inject
  Thermosiphon(Heater heater) {
    this.heater = heater;
  }

  ...
}
```

Dagger can inject fields directly. In this example it obtains a `Heater` instance for the `heater`field and a `Pump` instance for the `pump` field.

```
class CoffeeMaker {
  @Inject Heater heater;
  @Inject Pump pump;

  ...
}
```

If your class has `@Inject`-annotated fields but no `@Inject`-annotated constructor, Dagger will inject those fields if requested, but will not create new instances. Add a no-argument constructor with the `@Inject` annotation to indicate that Dagger may create instances as well.

Satisfying Dependencies

By default, Dagger satisfies each dependency by constructing an instance of the requested type as described above. When you request a `CoffeeMaker`, it'll obtain one by calling `new CoffeeMaker()` and setting its injectable fields.
But `@Inject` doesn't work everywhere:

- Interfaces can't be constructed.
- Third-party classes can't be annotated.
- Configurable objects must be configured!

```
@Provides static Heater provideHeater() {
  return new ElectricHeater();
}
```

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

All `@Provides` methods must belong to a module. These are just classes that have an `@Module` annotation.

```
@Module
class DripCoffeeModule {
  @Provides static Heater provideHeater() {
    return new ElectricHeater();
  }

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

The `@Inject` and `@Provides`-annotated classes form a graph of objects, linked by their dependencies.

the `@Component` annotation to such an interface and passing
the module types to the `modules` parameter, Dagger 2 then fully generates an implementation of that contract.

```
@Component(modules = DripCoffeeModule.class)
```

```
interface CoffeeShop {
  CoffeeMaker maker();
}
```

Obtain an instance by invoking the `builder()` method on that implementation and use the returned builder to set dependencies and `build()` a new instance.

```
CoffeeShop coffeeShop = DaggerCoffeeShop.builder()
    .dripCoffeeModule(new DripCoffeeModule())
    .build();
```

Second Example of Dagger

Dagger 2 is a compile-time android dependency injection framework and uses the Java Specification Request (JSR) 330 and uses an annotation processor.

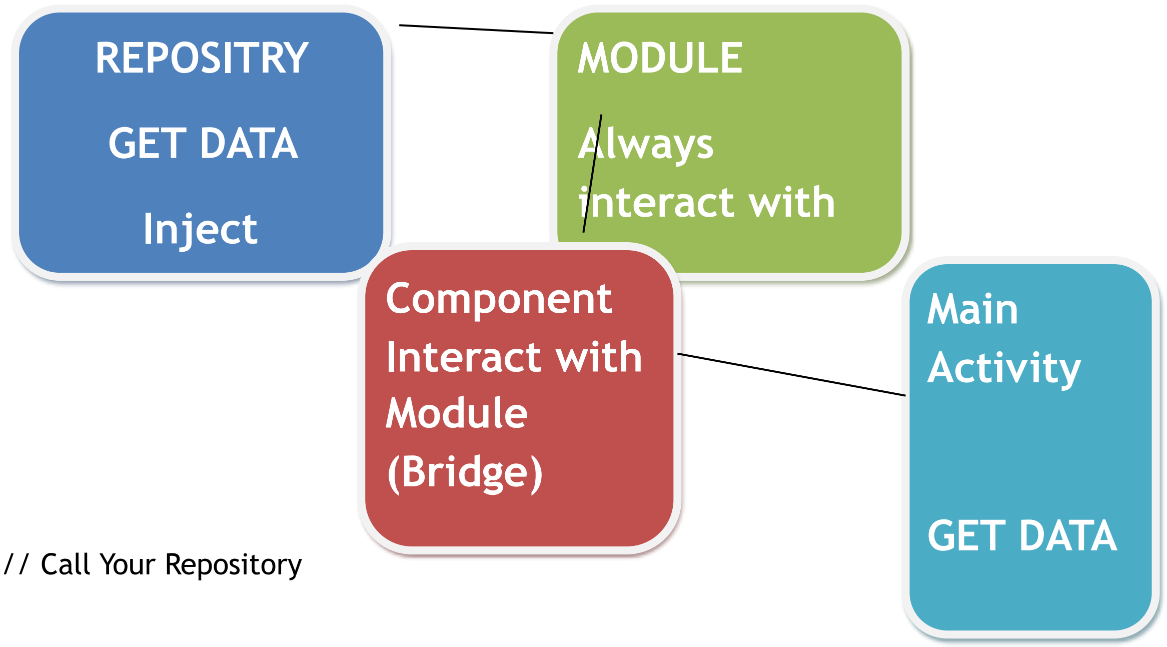
Following are the basic annotations used in Dagger 2:

@Module : This is used on the class that does the work of constructing objects that'll be eventually provided as dependencies.

@Provide : This is used on the methods inside the Module class that'll return the object.

@Inject : This is used upon a constructor, field or a method and indicates that dependency has been requested.

@Component : The Module class doesn't provide the dependency directly to the class that's requesting it. For this a Component interface is used that acts like a bridge between @Module and @Inject.



// Call Your Repository

```
import javax.inject.Inject;


// Retro call or you call function from server
public class TokenRepository {
    @Inject
    public  TokenRepository(){};

    public String getAllTokens(){
        return "here my Tokens";
    }
}
```

Make your module

```
@Module
public class TokenModule {

    @Provides
    public TokenRepository getTokenRepository(){
        return new TokenRepository();
    }


}
```

```
Use module for inject

@Component(modules={TokenModule.class})
public interface TokenComponent {
    public void inject(MainActivity mainActivity);
    TokenRepository getTokenRep();
}
```

```
// Inject in main activity
@Inject
TokenRepository tokenRepository;

DaggerTokenComponent daggerTokenComponent;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    tokenRepository = DaggerTokenComponent.create().getTokenRep();
    Log.d("value",""+tookenRepository.getAllTokens().toString());



}
```

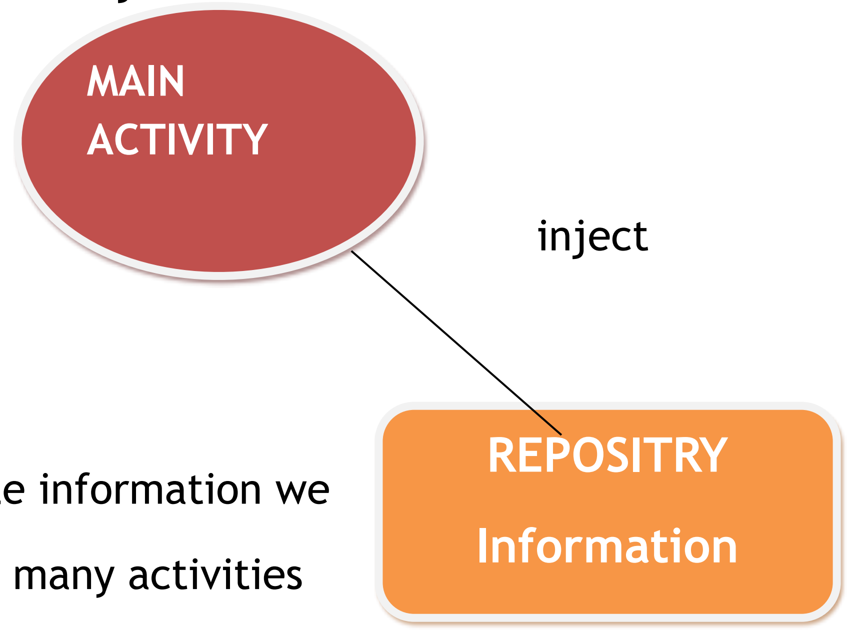
More Detail

Dagger (KOTLIN)

MAIN
ACTIVITY
REPOSITRY
(information)

If we want single class information in to our activity we just use it directly here ...

Why We need Dagger why we inject the information



If we want to provide

Information to

Activity then we need to

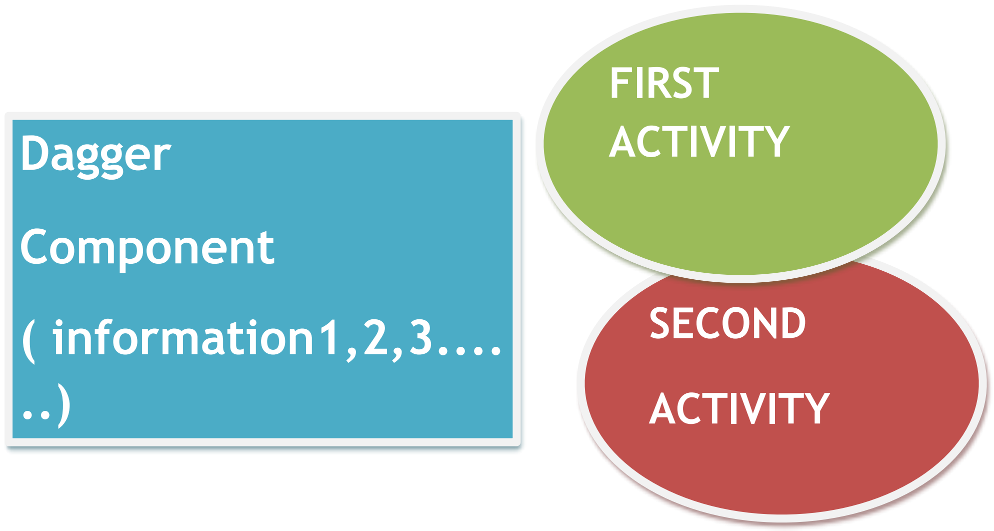
Inject it ..

Our app not work on single information we

Work or many classes and many activities

Sometimes it create complexity to provide information in many activities and in many fragments or classes so we use the dagger in that cases

So we need a component in dagger which contain all information which need to be provide so it bridge between source and destination ..



And many more

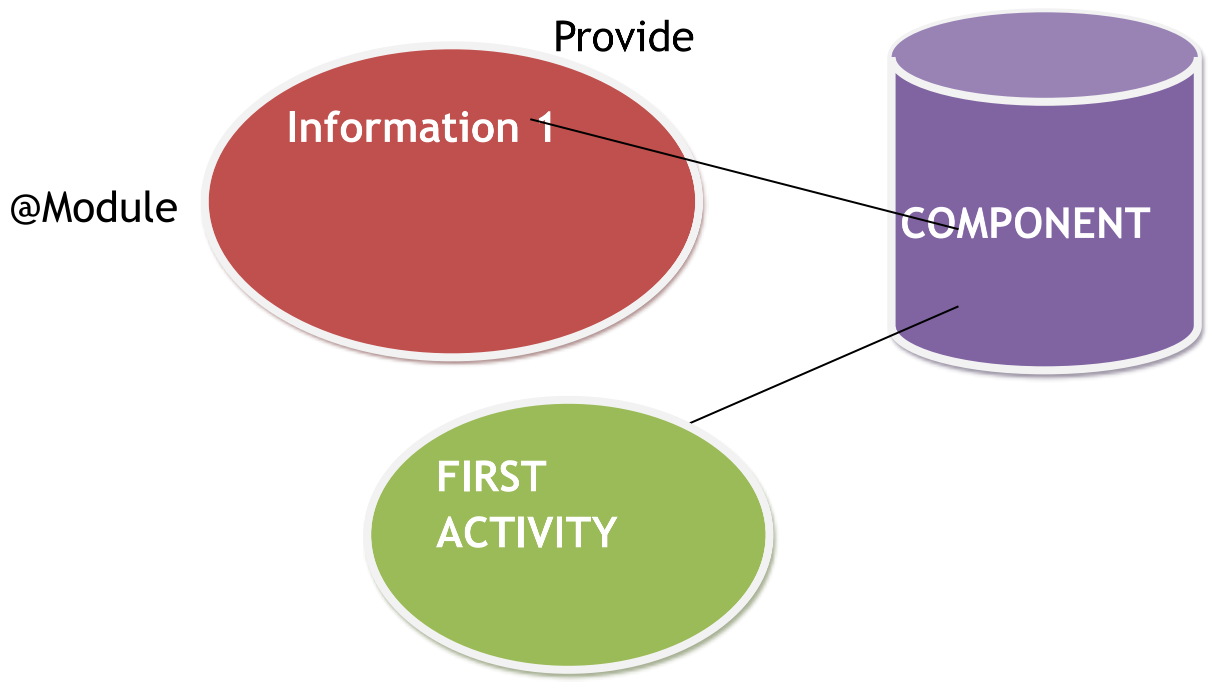
This is called multiple dependencies working for activities ...

First learn this practically

@Component , @Inject

If we want to provide different information then

Using @provides but how we can provide it we can use @Module for this

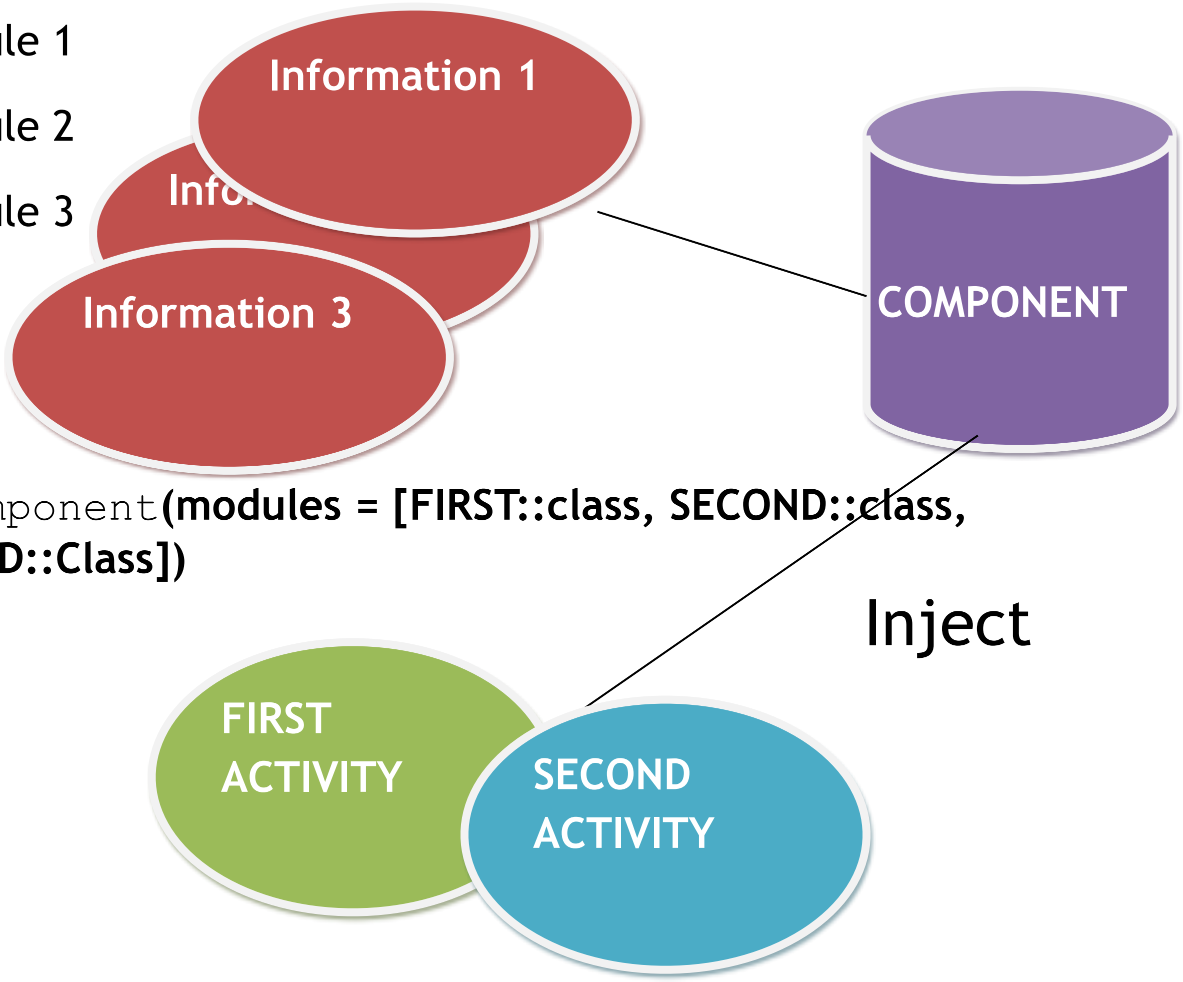


@Module works using @provides

Module 1

Module 2

Module 3



`@Component`**(modules = [FIRST::class, SECOND::class, THIRD::Class])**

But what happen if we provide two information using the same class .. we use @Named

and what about @singlton and how we can implement this with MVVM in kotlin

@Qualifier

@Documented

@Retention(RUNTIME)

properties declared as having a non-null type must be initialized in the constructor.properties can be initialized through dependency injection but you still want to avoid null checks when referencing the property inside the body of a class.Using lateinit, the initial value does not need to be assigned