Sources referred: Section 1
1. https://medium.com/@suhelchakraborty/dagger-2-modules-components-and-subcomponents-a-complete-story-part-i-1f484de3b15 2. https://iammert.medium.com/new-android-injector-with-dagger-2-part-1-8baa60152abe 3. https://iammert.medium.com/new-android-injector-with-dagger-2-part-2-4af05fd783d0

Dagger is a dependency injection framework for Android and also Java. In short it helps you forget the **new** keyword in favour of managing your dependencies through a series of interfaces and alike.
Note this: ***managing your dependencies through a series of interfaces and alike***

Component-Subcomponent relationship = parent-child relationship. What the parent has, the child has also

SubComponents are Components derived from another parent Component, getting their Modules in addition to its own Modules. One SubComponent can have only one parent Component.
I'm guessing that this statement from here is foundationally...the reason why our original approach of SHARING LendingOnboarding with Lending component & Onboarding component failed

Application class builds a graph using AppComponent.(AppComponent has @Component annotation top of its class)
When AppComponent is built with its modules, we have a graph with all provided instances in our graph.
For instance, If app module provides ApiService, we will have ApiService instance when we build component which has app module.
Draw this graph out

we need to tell ancestor about subcomponent info. So all subcomponents have to be known by its ancestor.
According to source 2

//ARE THESE OBSOLETE given we use factories now
Component.Builder: We might want to bind some instance to Component. In this case we can create an interface with @Component.Builder annotation and add whatever method we want to add to builder. In my case I wanted to add Application to my AppComponent.
Without even touching manual DI, I'm guessing that we could accomplish the same results as Builder without actually using a builder?
Bro honestly, WHAT IS A BUILDER AND A FACTORY FOR???
EY YO, IS BUILDER JUST A MEANS TO RECEIVE DEPENDENCIES FROM OUTSIDE THE MODULE???

```
DaggerAppComponent
        .builder()
        .application(this)
        .build()
        .inject(this);
```

WHY WOULD YOU WANT TO RECEIVE THE SAME THING TWICE THO

//ARE THESE OBSOLETE
We can think apps in three layer while using Dagger.

- Application Component
- Activity Components
- Fragment Components

How shall this be translated to in Jetpack compose?

//ARE THESE OBSOLETE

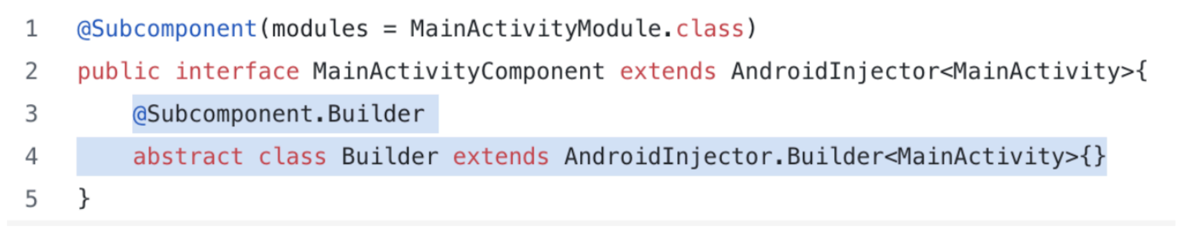
Application component is providing 3 module in our app.

- AndroidInjectionModule: We didn’t create this. It is an internal class in Dagger 2.10. Provides our activities and fragments with given module.
- ActivityBuilder: We created this module. This is a given module to dagger. We map all our activities here. And Dagger know our activities in compile time. In our app we have Main and Detail activity. So we map both activities here.

Obsolete ah? Since we're using Jetpack Compose

```
@Module(subcomponents = {
        MainActivityComponent.class,
        DetailActivityComponent.class})
public class AppModule {
    @Provides
    @Singleton
    Context provideContext(Application application) {
        return application;
    }
}
```

Let this be a reference



```
@Subcomponent(modules = MainActivityModule.class)
public interface MainActivityComponent extends AndroidInjector<MainActivity>{
    @Subcomponent.Builder
    abstract class Builder extends AndroidInjector.Builder<MainActivity>{}
}
```

- MainActivityComponent:This component is just a bridge to MainActivityModule.
- Here is an important change. **We don’t add inject() and build() method to this component.**
- MainActivityComponent has these methods from ancestor class.
- AndroidInjector class is new dagger-android class which exist in dagger-android framework. ***//ARE THESE OBSOLETE***

I’m guessing that this implies that for ALL subcomponents, we don’t need to use the `inject()` & `build()` function

I can’t find `AndroidInjector` in our codebase sia

```java
@Module
public class MainActivityModule {

    @Provides
    MainView provideMainView(MainActivity mainActivity){
        return mainActivity;
    }

    @Provides
    MainPresenter provideMainPresenter(MainView mainView, ApiService apiService){
        return new MainPresenterImpl(mainView, apiService);
    }
}
```

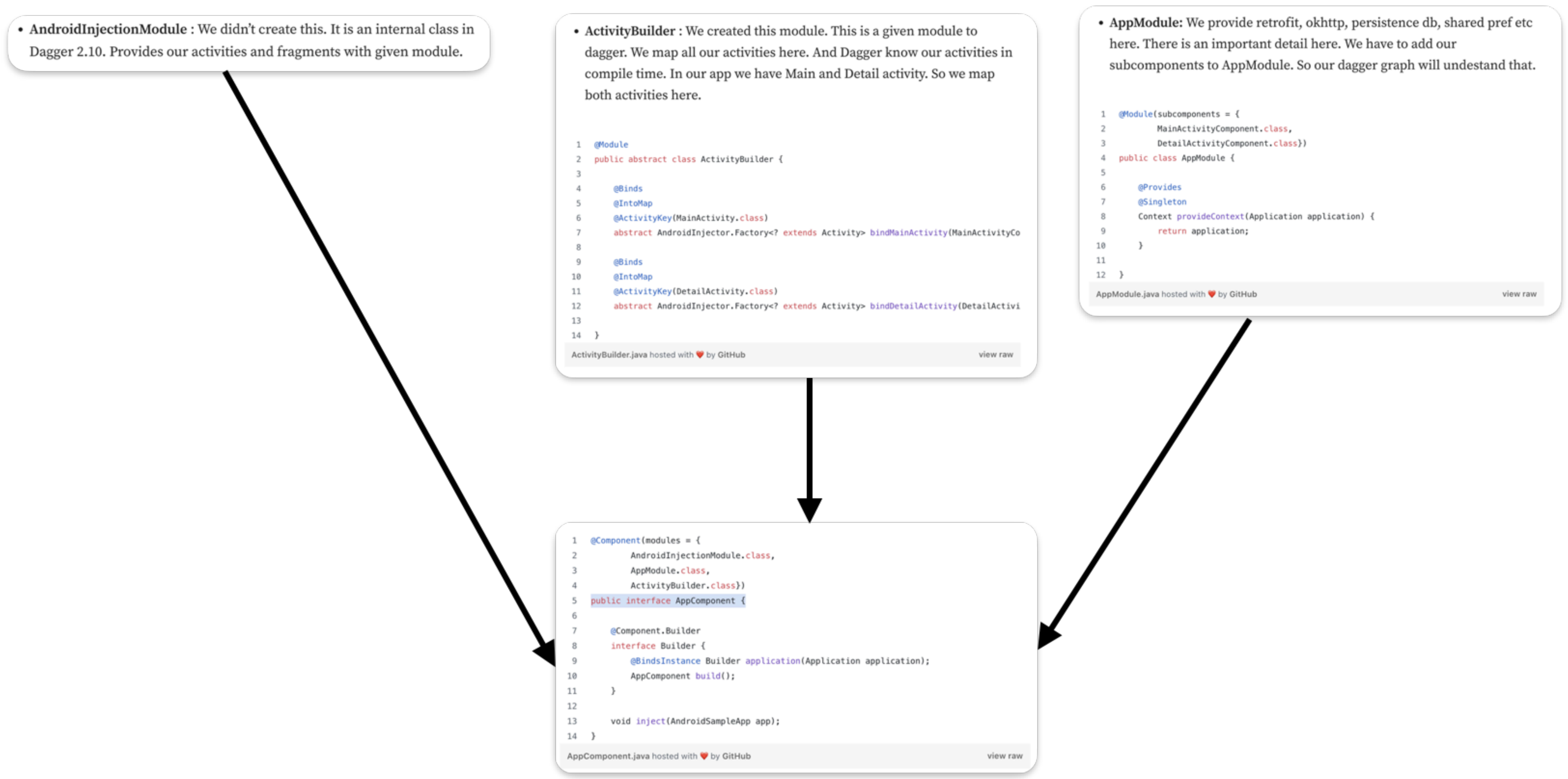
- MainActivityModule: This module provides main activity related instances(eg. MainActivityPresenter).
- Notice that `provideMainView()` method takes MainActivity as parameter.
- We create our MainActivityComponent with our <MainActivity> class.
- So dagger will attach our activity to its graph. So we can use it because it is on the graph.

Fragment Components...seems straight forward ***//ARE THESE OBSOLETE***
DispatchingAndroidInjector<T>...skipping this, not sure if it's needed ***//ARE THESE OBSOLETE***

AndroidInjection.inject(this) ***//ARE THESE OBSOLETE***

- Will I be needing this now? Is this still relevant?
- Anyway, this feels like the alternative to the AppComponent's `Builder` functions that insert the invoker into the graph/modules

@ContributesAndroidInjector ***//ARE THESE OBSOLETE***
Will I...need this???



- **AndroidInjectionModule :** We didn't create this. It is an internal class in Dagger 2.10. Provides our activities and fragments with given module.

- **ActivityBuilder :** We created this module. This is a given module to dagger. We map all our activities here. And Dagger know our activities in compile time. In our app we have Main and Detail activity. So we map both activities here.

```java
@Module
public abstract class ActivityBuilder {

    @Binds
    @IntoMap
    @ActivityKey(MainActivity.class)
    abstract AndroidInjector.Factory<? extends Activity> bindMainActivity(MainActivityCo

    @Binds
    @IntoMap
    @ActivityKey(DetailActivity.class)
    abstract AndroidInjector.Factory<? extends Activity> bindDetailActivity(DetailActivi

}
```

ActivityBuilder.java hosted with ❤ by GitHub view raw

- **AppModule:** We provide retrofit, okhttp, persistence db, shared pref etc here. There is an important detail here. We have to add our subcomponents to AppModule. So our dagger graph will undestand that.

```java
@Module(subcomponents = {
        MainActivityComponent.class,
        DetailActivityComponent.class})
public class AppModule {

    @Provides
    @Singleton
    Context provideContext(Application application) {
        return application;
    }

}
```

AppModule.java hosted with ❤ by GitHub view raw

```java
@Component(modules = {
        AndroidInjectionModule.class,
        AppModule.class,
        ActivityBuilder.class})
public interface AppComponent {

    @Component.Builder
    interface Builder {
        @BindsInstance Builder application(Application application);
        AppComponent build();
    }

    void inject(AndroidSampleApp app);
}
```

AppComponent.java hosted with ❤ by GitHub view raw

Notice that the Subcomponents are announced into the parent's modules

//ARE THESE OBSOLETE

Do we want to touch on part 3? Looks super legacy: https://medium.com/android-news/new-android-injector-with-dagger-2-part-3-fe3924df6a89

Sources referred: Section 2
1. https://janisharali.com/blog/introduction-to-dagger-2-using-dependency-injection-in-android-part-1-223289c2a01b 2. https://janisharali.com/blog/introduction-to-dagger-2-using-dependency-injection-in-android-part-2-b55857911bcd

Inversion of Control

- a class should get its dependencies from outside.
- No class should instantiate another class but should get the instances from a configuration class.
- E.g. 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 and is called a hard dependency.
- benefits of providing the dependencies from outside the class
    - The most important advantage is that it increases the possibility of reusing the class + being able to test them independent of other classes.

Annotation Processor: a way to read the compiled files during build time to generate source code files to be used in the project.

Notes:

- The standard java annotations for describing the dependencies of a class are defined in the Java Specification Request 330 (JSR 330)
- The order in which the methods or fields annotated with @Inject are called is not defined by JSR330
- You cannot assume that the methods or fields are called in the order of their declaration in the class.
    - As fields and method parameters are injected after the constructor is called, you cannot use injected member variables in the constructor

visualize the dependency injection process with Dagger as follows:
A **dependency consumer** asks for the **dependency(Object)** from a **dependency provider** through a **connector**.

- **Dependency provider**:
    - Classes annotated with @Module
    - responsible for providing objects which can be injected.
    - Such classes define methods annotated with @Provides.
    - The returned objects from these methods are available for dependency injection.
- Dependency consumer:
    - The @Inject annotation is used to define a dependency.
- Connecting consumer and producer:
    - A @Component annotated interface
    - **Defines the connection between** the **provider of objects (modules)** and the **objects which express a dependency**.
    - The class for this connection is generated by the Dagger.

Limitations of Dagger :

- Dagger does not inject fields automatically.
- It cannot inject private fields.
- If you want to use field injection you have to define a method in your @Component annotated interface
    - Takes the instance of the class into which you want to inject the member variable.

@Qualifier annotation

- Provided by javax inject package
- Used to qualify the dependency.
- Used to distinguish between objects of the same type but with different instances.
- For example,
    - a class can ask for BOTH an Application Context and an Activity Context.
    - But both these Objects will be of type Context.
    - So, for Dagger to figure out which variable is to be provided with what, we have to explicitly specify the identifier for it.

@Named annotation

- An alternative to @Qualifier
- provided by Dagger.
- @Named itself is annotated with @Qualifier.
- With @Named we have to provide a string identifier for similar class objects
- this identifier is used to map the dependency of a class.

@Scope

- used to specify the scope in which a dependency object persists.
- If a class is getting dependencies, have members injected with classes annotated with a scope
    - then each instance of that class asking for dependencies will get its own set of member variables.

@Singleton
ensure a single instance of a class globally.
It will be provided with the same instance that is maintained in the Dagger's dependency graph.

@Inject - instructs the Dagger to accumulate all the parameter dependencies when the class is being constructed.
@ApplicationContext Qualifier - facilitates Recepient to get the context object of the application from dagger's dependency graph
@DatabaseInfo qualifier - helps the dagger to distinguish between String and Integer Dependencies from existing same types in the dependency graph.

Step 10 contains some stuff that I don't know if it's obsolete already or not

```
@PerActivity
@Component(dependencies = ApplicationComponent.class, modules = ActivityModule.class
public interface ActivityComponent {

    void inject(MainActivity mainActivity);

}
```

What does dependencies look like in the graph??? Shouldn't it be Subcomponent?

Random pieces

- ActivityComponent specify ApplicationComponent and ActivityModule.
- ApplicationComponent is added to use the graph that has already been generated in the previous step + already exists because the DemoApplication class persists till the application is running(terminated?).
- @PerActivity is a scope
    - used to tell the Dagger that the Context and Activity provided by the ActivityModule will be instantiated each time an Activity is created.
    - So, these objects persist till that activity lives(dies?) and each activity has its own set of these.
- We may ask that the DataManager will then be created with each Activity.
    - But that is not true because we have annotated the DataManager class with @Singleton.
    - Which brings the class in the global scope and thus is accessed whenever a dependency is expressed.

Now let's revisit the DemoApplication class and MainActivity class.
These classes don't have a constructor and Android System is responsible for instantiating these.
To get the dependency we use the OnCreate method as it is called once when they are instantiated.

```
applicationComponent = DaggerApplicationComponent
                            .builder()
                            .applicationModule(new ApplicationModule(this))
                            .build();
applicationComponent.inject(this);
```

- We provide the ApplicationModule class that is used to construct the dependencies.
- We have also called the inject method of applicationComponent and passed the instance of the DemoApplication class.
    - This is done to use it for providing the DataManager.ApplicationComponent instance is retained so as to access all the classes that are available in the dependency graph and is express for access.

As we mentioned about @Named("string")annotation, we just have to replace @ApplicationContext and @ActivityContext with something like @Named("application_context") and @Named("activity_context") everywhere.
But personally I don’t like this implementation. We are required to maintain a String tag.

Note: If due to some reason we have to inject dependencies through field injection for classes other that android constructed classes then

- define a component interface for that class
- Call it’s inject method in class’s constructor.

I would suggest to figure out a way not to do this but try to use the constructor with @Inject for the injection.

Sources referred: Section 3
• https://medium.com/android-news/dagger2-component-factory-and-subcomponent-factory-b181ec96b213 • https://medium.com/@zekromvishwa56789/understanding-dagger-2-custom-scopes-and-subcomponents-in-android-1af073a3283e

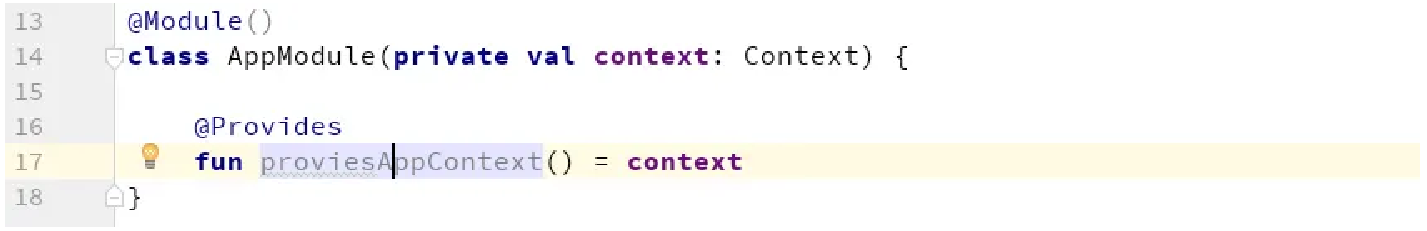
When we want to pass dependency to our modules from outside in order to facilitate injection in other classes the. e.g.

- passing application context to appmodule
- passing user id to User Detail Screen for which we need User Module to have User Id from outside.

Previously there were 2 ways to solve this problem but Dagger 2.22 introduced another way. Lets analyze all three approaches to apply the best solution.

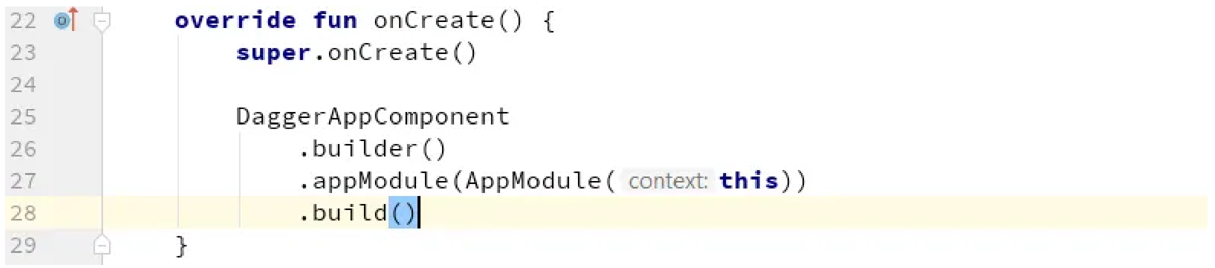
Approach 1: Modules with constructor arguments:

- This approach simply says “pass constructor argument to the module”.



```
@Module()
class AppModule(private val context: Context) {

    @Provides
    fun proviesAppContext() = context
}
```

- Pass the context to the module simply while creating component.



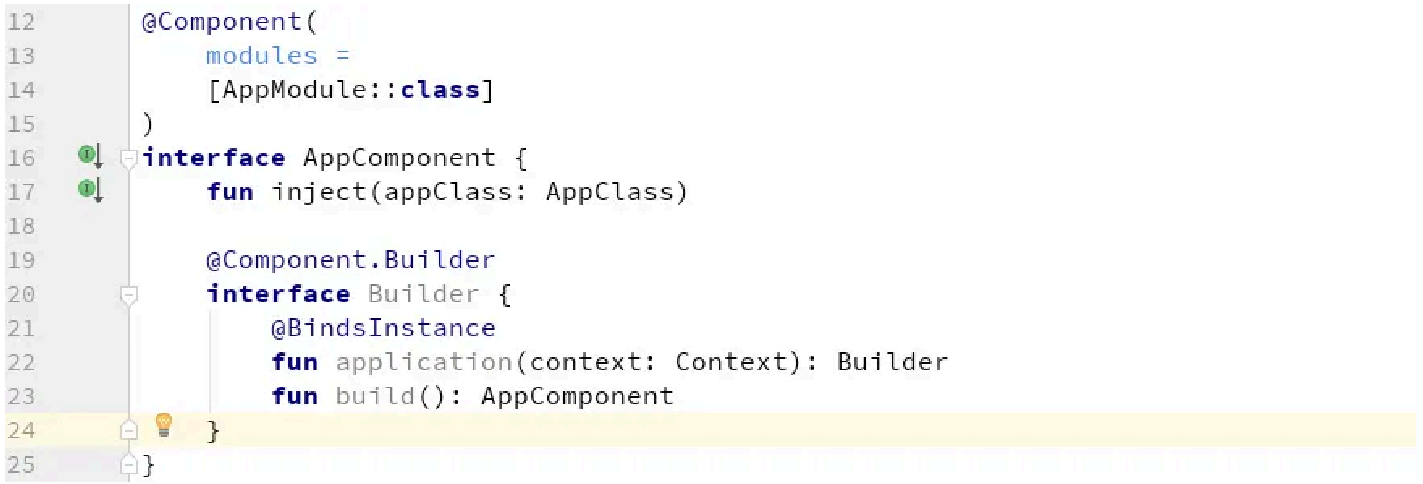
```
    override fun onCreate() {
        super.onCreate()

        DaggerAppComponent
            .builder()
            .appModule(AppModule( context: this))
            .build()
    }
```

Problem: This looks quite straight forward but the problem with this approach is that

- **We can’t make our module abstract(which is recommended as modules should not have states + modules methods should be statically available to dagger for better performance)**
- Because of that we have to pass the instance of module to our component.

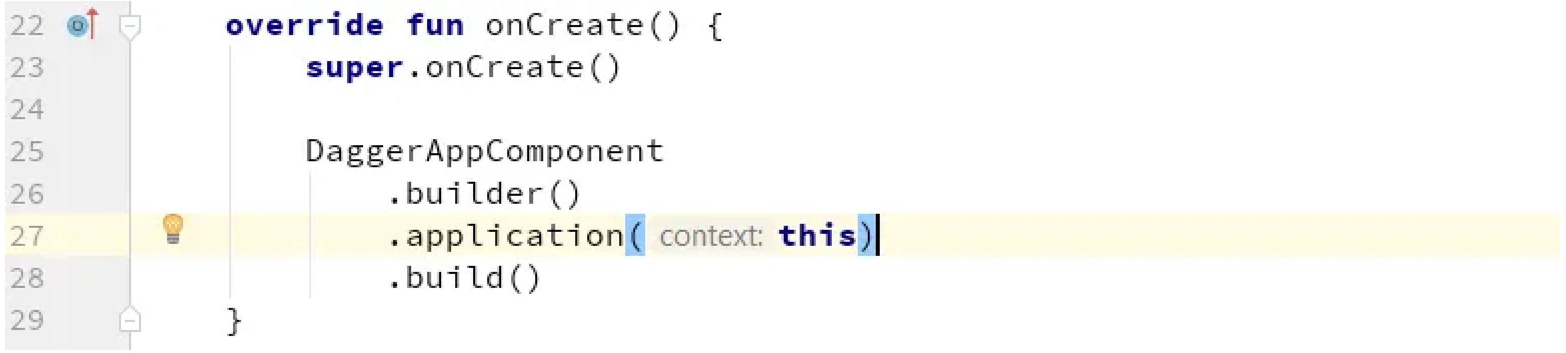
Approach 2: @Component.Builder:

In Dagger2.12 we got two new annotations @Component.Builder and @BindsInstances for doing the same thing we previously were doing by passing arguments to the constructor. We don’t have provide method in our module anymore.



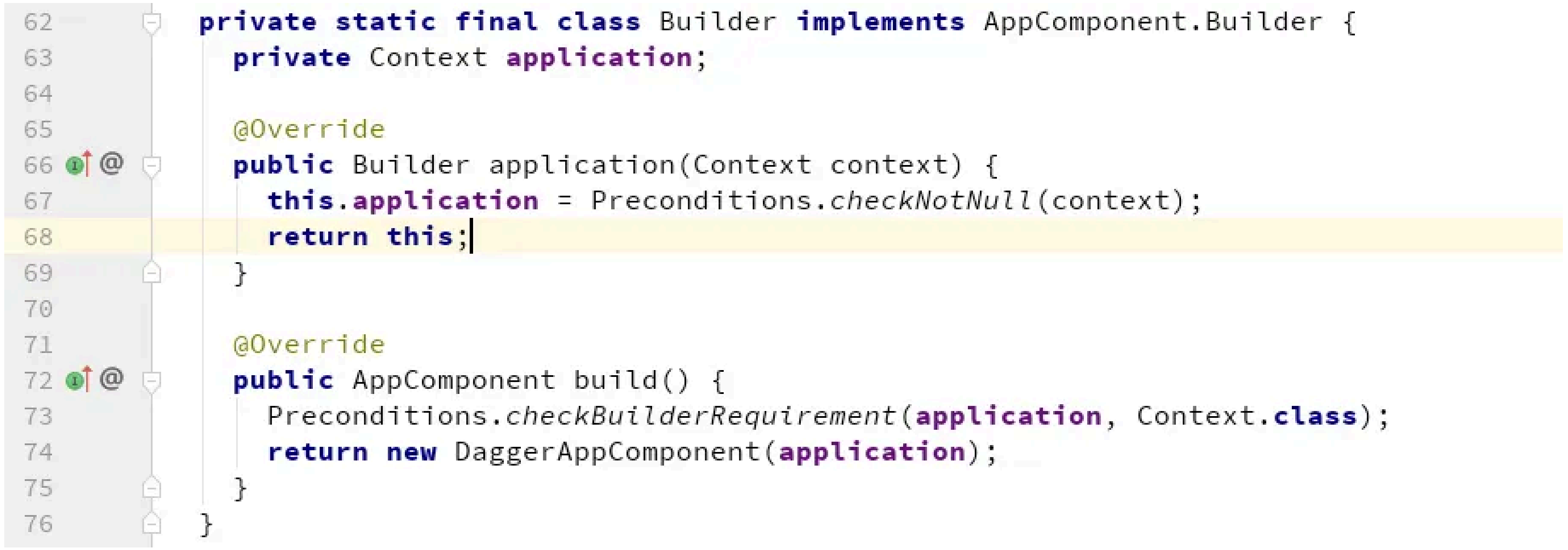
```kotlin
@Component(
    modules =
    [AppModule::class]
)
interface AppComponent {
    fun inject(appClass: AppClass)

    @Component.Builder
    interface Builder {
        @BindsInstance
        fun application(context: Context): Builder
        fun build(): AppComponent
    }
}
```

- As the name suggests @BindsInstances binds the instance to the component. By doing this we added the context to our dagger graph and now we can get context anywhere like before.
- There are some rules that your builder interface should follow.
  - Builder must have at least one method which returns the Component or super type of the Component, in our example it is the build() function.
  - Methods other than build method that are used to bind instances should return the Builder type.
  - Methods used for binding dependencies must not have more than one parameter, if you want to bind more dependencies then create different methods for each dependency.
- This is how our component will be created.



```kotlin
    override fun onCreate() {
        super.onCreate()

        DaggerAppComponent
            .builder()
            .application( context: this)
            .build()
    }
```

- To understand how @Component.Builder works we have to look at the generated code.
- Our component interface name is AppComponent so dagger generated a class named DaggerAppComponent which is providing the implementation of our component.
- Dagger uses Builder pattern to create components, so inside our DaggerAppComponent class we have got static final class called Builder.
- Now because we have provided the builder interface in our AppComponent -> the inner Builder class present inside DaggerAppComponent implements that Builder interface created by us.

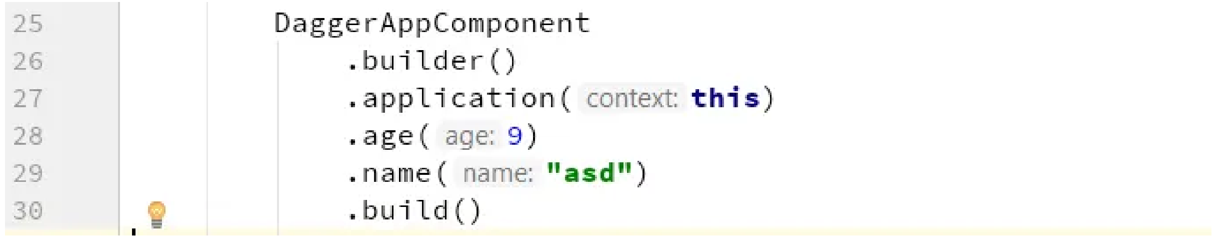


```java
private static final class Builder implements AppComponent.Builder {
  private Context application;

  @Override
  public Builder application(Context context) {
    this.application = Preconditions.checkNotNull(context);
    return this;
  }

  @Override
  public AppComponent build() {
    Preconditions.checkBuilderRequirement(application, Context.class);
    return new DaggerAppComponent(application);
  }
}
```

On calling build, dagger passes the context passed by us to the component.
How is this solution better than the previous one?
Well we don't have to pass constructor arguments any more so our modules can be stateless and can contain all static methods. Thats a success!

Problem

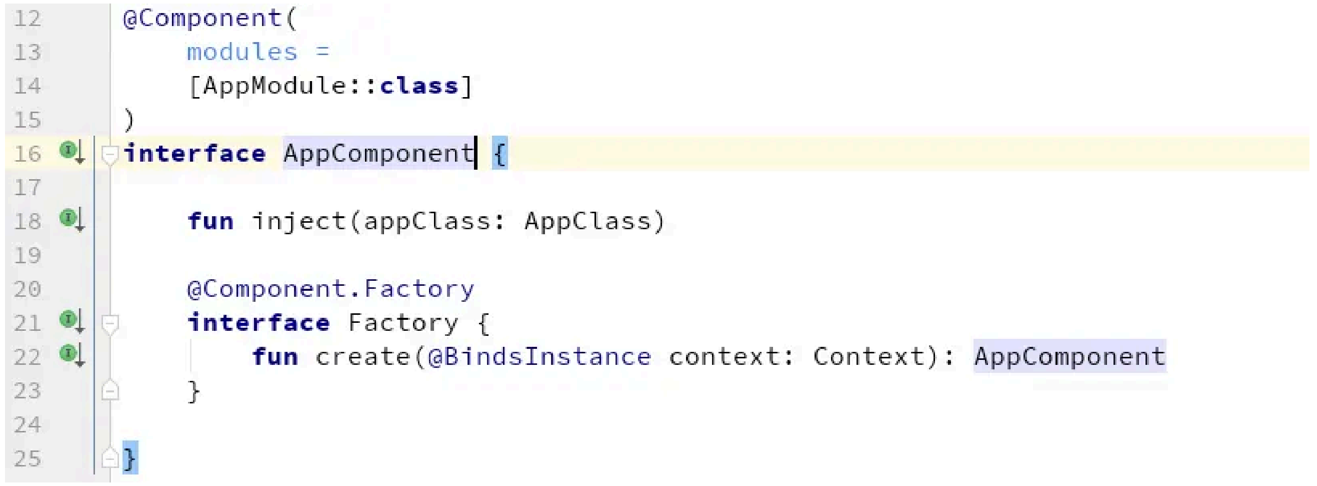
It is not quite a performance related problem but if we want to bind many instances to the component then it will create a long chain and if we forgot to call any method in the chain it will result in a run time exception. So that's the problem remaining to address.



```java
DaggerAppComponent
    .builder()
    .application( context: this)
    .age( age: 9)
    .name( name: "asd")
    .build()
```

Approach 3: The winner @Component.Factory

With dagger 2.22 we have got yet another amazing annotation named @Component.Factory to address the problems that were introduced as the result of using @Component.Factory. With Dagger 2.22 now we can use @BindsInstances with each parameter.

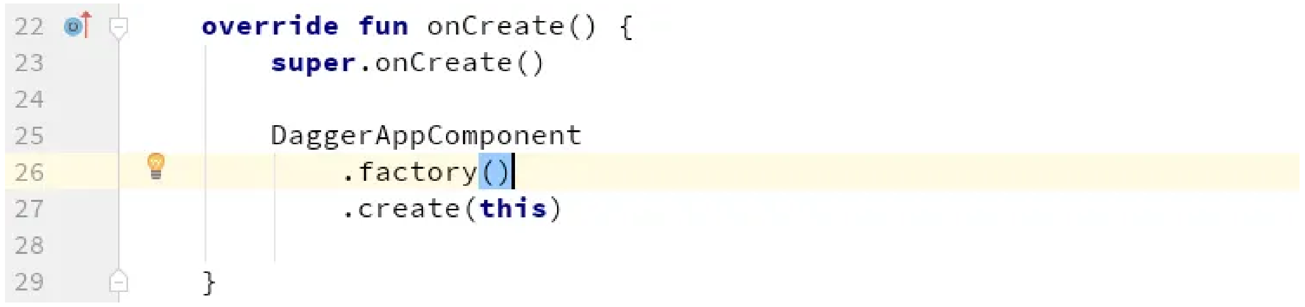


```
@Component(
    modules =
    [AppModule::class]
)
interface AppComponent {

    fun inject(appClass: AppClass)

    @Component.Factory
    interface Factory {
        fun create(@BindsInstance context: Context): AppComponent
    }

}
```

As precise as you like it, you can see how much lines of codes we have saved.

- Like @Component.Builder there are some rules that your Factory interface should adhere.
- Your factory can not have more than 1 method.
    - If you want to bind many dependencies then instead of creating method for each dependency(as we did with @Component.Builder)
        - you can simply add a new parameter for each dependency.
    - Your method must return the type of Component or super type of Component.

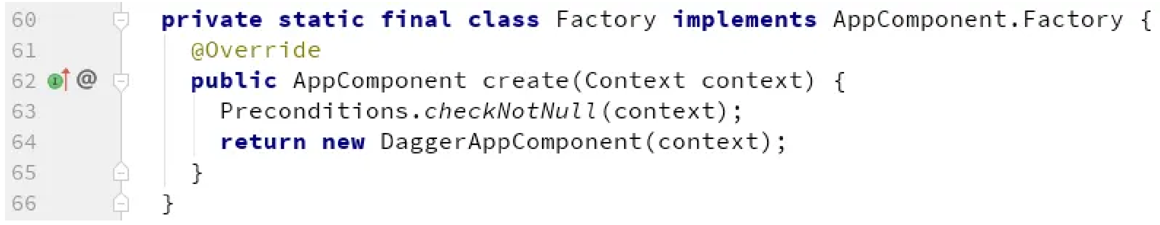
And now our component will be created like this.



```
override fun onCreate() {
    super.onCreate()

    DaggerAppComponent
        .factory()
        .create(this)

}
```

To understand how @Component.Factory works we have to look at the generated code.

Inside DaggerAppComponent instead of Builder class now we get Factory class which is implementing our created Factory interface.



```
private static final class Factory implements AppComponent.Factory {
  @Override
  public AppComponent create(Context context) {
    Preconditions.checkNotNull(context);
    return new DaggerAppComponent(context);
  }
}
```

How this solution is better than the previous one?

- Since we are not adding method for each dependency there will be no long chains and method count will remain same in every case.
- Each dependency will add a parameter in a function and if we forgot to pass any dependency we will get a compile time error instead of run time exception

Like @Component.Builder and @Component.Factory there are also subcomponent version for both these annotations

- @SubComponent.Builder - https://dagger.dev/api/2.11/dagger/Subcomponent.Builder.html
- @SubComponent.Factory - https://dagger.dev/api/2.22/dagger/Subcomponent.Factory.html

Sample Project Overview
It'll include:

- Application-wide dependencies
- Activity-specific dependencies
- Fragment-specific dependencies

To manage these dependencies effectively, we will define custom scopes and use subcomponents.

Custom Scopes

- Custom scopes help manage the lifespan of dependencies. Let's define scopes for activities and fragments.
- Defining Custom Scopes

```
@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class ActivityScope

@Scope
@Retention(AnnotationRetention.RUNTIME)
annotation class FragmentScope
```

Modules and Components - we'll define three modules and three components to manage dependencies:

- AppModule for application-wide dependencies: ***we provide the application context, which can be used across the entire application.***

```
@Module
class AppModule(private val context: Context) {
    @Provides
    @Singleton
    fun provideContext(): Context {
        return context
    }
}
```

- ActivityModule for activity-specific dependencies: ***we provide the activity instance, which will be scoped to the activity lifecycle.***

```
@Module
class ActivityModule(private val activity: AppCompatActivity) {
    @Provides
    @ActivityScope
    fun provideActivity(): AppCompatActivity = activity
}
```

- FragmentModule for fragment-specific dependencies: ***we provide the fragment instance, which will be scoped to the fragment lifecycle.***

```
@Module
class FragmentModule(private val fragment: Fragment) {
    @Provides
    @FragmentScope
    fun provideFragment(): Fragment = fragment
}
```

Components and Subcomponents

AppComponent - AppComponent is the root component that provides application-wide dependencies.

```
@Singleton
@Component(modules = [AppModule::class])
interface AppComponent {
    fun inject(application: MyApplication)
    fun activityComponent(activityModule: ActivityModule): ActivityComponent
}
```

ActivityComponent - ActivityComponent is a subcomponent of AppComponent and provides activity-specific dependencies.

```
@ActivityScope
@Subcomponent(modules = [ActivityModule::class])
interface ActivityComponent {
    fun inject(activity: HomeActivity)
    fun fragmentComponent(fragmentModule: FragmentModule): FragmentComponent
}
```

FragmentComponent - FragmentComponent is a subcomponent of ActivityComponent and provides fragment-specific dependencies.

```
@FragmentScope
@Subcomponent(modules = [FragmentModule::class])
interface FragmentComponent {
    fun inject(fragment: HomeFragment)
}
```

Application Class - In the Application class, we initialize the AppComponent.

```
class MyApplication : Application() {

    lateinit var appComponent: AppComponent

    override fun onCreate() {
        super.onCreate()
        appComponent = DaggerAppComponent.builder()
             .appModule(AppModule(this))
             .build()
    }
}
```

Activity Setup - In HomeActivity, we create the ActivityComponent from AppComponent.

```
@ActivityScope
class HomeActivity : AppCompatActivity() {

    lateinit var activityComponent: ActivityComponent

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        activityComponent = (application as MyApplication).appComponent
             .activityComponent(ActivityModule(this))
        activityComponent.inject(this)
    }
}
```

Fragment Setup - In HomeFragment, we create the FragmentComponent from ActivityComponent.

```
@FragmentScope
class HomeFragment : Fragment() {

    lateinit var fragmentComponent: FragmentComponent

    override fun onAttach(context: Context) {
        super.onAttach(context)
        fragmentComponent = (activity as HomeActivity).activityComponent
             .fragmentComponent(FragmentModule(this))
        fragmentComponent.inject(this)
    }
}
```

Accessing Dependencies Across Scopes

One of the main benefits of using subcomponents and custom scopes is the ability to access dependencies across different scopes. For example:

- Application-wide dependencies:
  - These are available throughout the entire app.
  - E.g. you might need access to the application context or a singleton service class

```
class MySingletonService @Inject constructor(private val context: Context) {
   // Singleton service code
}
```

- Activity-specific dependencies: These are available within a specific activity. For example, you might have a presenter or view model that should only live as long as the activity.

```
class HomeActivity : AppCompatActivity() {

   @Inject
   lateinit var mySingletonService: MySingletonService // Application scoped dependency

   @Inject
   lateinit var activityPresenter: ActivityPresenter // Activity scoped dependency

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       activityComponent = (application as MyApplication).appComponent
           .activityComponent(ActivityModule(this))
       activityComponent.inject(this)

       // Use mySingletonService and activityPresenter
   }
}
```

- Fragment-specific dependencies: These are available within a specific fragment. For example, you might have a fragment presenter or view model that should only live as long as the fragment.

```
class HomeFragment : Fragment() {

   @Inject
   lateinit var mySingletonService: MySingletonService // Application scoped dependency

   @Inject
   lateinit var activityPresenter: ActivityPresenter // Activity scoped dependency

   @Inject
   lateinit var fragmentPresenter: FragmentPresenter // Fragment scoped dependency

   override fun onAttach(context: Context) {
       super.onAttach(context)
       fragmentComponent = (activity as HomeActivity).activityComponent
           .fragmentComponent(FragmentModule(this))
       fragmentComponent.inject(this)
```

```
        // Use mySingletonService, activityPresenter, and fragmentPresenter
    }
}
```

In this example, HomeFragment can access dependencies from the application scope (mySingletonService), activity scope (activityPresenter), and its own fragment scope (fragmentPresenter). This demonstrates the flexibility and power of using custom scopes and subcomponents in Dagger 2.

Understanding the Lifespans and Scopes
@Singleton:

- Lifespan: Lives as long as the application is alive.
- Usage: For application-wide dependencies such as Context.

@ActivityScope:

- Lifespan: Lives as long as the activity is alive.
- Usage: For dependencies that should exist only within an activity’s lifecycle, such as activity-specific presenters or view models.

@FragmentScope:

- Lifespan: Lives as long as the fragment is alive.
- Usage: For dependencies that should exist only within a fragment’s lifecycle, such as fragment-specific presenters or view models.

How Subcomponents Work
allow for a hierarchical structure where dependencies can be scoped to specific parts of your application. Here’s how they interact:

- AppComponent: Provides dependencies that live as long as the application.
- ActivityComponent: A subcomponent of AppComponent that provides dependencies specific to an activity's lifecycle.
- FragmentComponent: A subcomponent of ActivityComponent that provides dependencies specific to a fragment's lifecycle.

This hierarchical structure ensures that dependencies are appropriately scoped and managed, preventing memory leaks and ensuring efficient resource usage.

Sources referred
● https://medium.com/tompee/android-dependency-injection-using-dagger-2-530aa21961b4 ● https://medium.com/tompee/dagger-2-scopes-and-subcomponents-d54d58511781

Dependency injection

- A pattern in which the dependencies are provided to the object instead of finding and building it on its own.
- It is part of the umbrella concept Inversion of Control.
- Offloading dependency creation and management from your application and delegating it to a "service".
- think of it as a service that contains all of my dependency.
  - During injection, dagger will look for your dependency, and create it.
  - If your dependencies have other dependencies, dagger will create those dependencies first.
  - Notice that instantiation is deferred until actual injection.

It is not ideal for us to do DI on our own since we will encounter a lot of problems such as

- Complicated dependency graphs
- Cpu cycle and memory overhead when instantiating all our dependencies at once(even without actually needing it)

A good dependency injection allows you to

- build your dependency graph
- instantiate the dependencies lazily
- manage their lifecycles automatically.

When instantiating a component, you need to use the factory and provide all the module dependencies of the component.
E.g.
a project that depends on the AppModule is why we need to instantiate an AppModule using our context and set it to the factory.
Since we passed MyApplication context in AppModule, every time the context or application instance is requested from it, it will always provide the MyApplication context.

The component is

- the interface between the service and your application.
- an interface that dagger will implement.
- interface/abstract class
- the dependency graph.
- the container.
- It has a lot of functions.
- If you want any dependency from it, you should define it here.

this will not work yet. dagger does not really know (yet) where to get your dependencies, or how to provide them for you.
To do that, you have to link modules(Your component must contain the modules that satisfy all of its exposed objects)

Modules are

- what provide your dependencies.

- concrete classes.
- IF the module requires an instance of something, e.g. the application context
    - ask the user of this module to provide it for us(It is not wise to create a new context, since you are interested in the actual context of your application)
    - build our project first because dagger generates the code at compile time.
    - After building, dagger will generate a new class from your component prepended with ‘Dagger’.
    - Use this to build an instance of your component.(Go use factories for this)

@Provides & @Binds - instructs dagger that this method returns an object that can be used by the component to satisfy other internal or external dependencies.

The inject annotation
- instructs dagger that you need this dependency.
- Field injection example
    - Activities are created and maintained by the OS so it is difficult to do constructor injection.
    - Instead we will use field injection.
    - For field injection, we need to make the field accessible, annotate it with @Inject and manually trigger injection.
- There is an important thing to note about @Inject and constructor injection.
    - When you annotate a constructor with @Inject, dagger automatically adds it in the dependency graph.
    - This means there is no need for you to create a @Provides method for it in the modules.
    - **HOWEVER It's recommended to create @Provides methods as they are more intuitive and they are the way to go when annotation is not possible (such as 3rd party code).**

Singleton annotation
- This annotation instructs dagger that it should only create one instance of it.
- Although in a wider sense, singleton refers to application scope.
    - It means the scope and lifecycle of your object is tied to the application.

A separation of concern approach makes horizontal levels fairly easy to recognize. We can take advantage of two things:
- Dependency scopes can be localized
- Lifecycle can be localized

We need localization because we do not want all our dependencies to live as long as the application(there are cases when we want our dependencies to not share the same state by being the same object ///???)

Dagger 2 provides @Scope as a mechanism to handle scoping.
- Scoping allows you to “preserve” the object instance and provide it as a “local singleton” for the duration of the scoped component.
- The claim that "@Singleton allows you to define a component or a dependency as a global object" IS NOT ENTIRELY TRUE.
    - @Singleton, is just another scope. It was just there as the default scoping annotation.
    - It provides your dependencies as a singleton as long as you are using the same component.

This right here is similar to the singleton scope
@Scope
@Retention
annotation class ApplicationScope

THE ENTIRETY OF THE INTRO PLUS THE EXAMPLE GIVEN IS PHENOMENAL, REFER TO THAT DIRECTLY FROM SITE

A component can establish a parent-child dependency between itself and other components. For this to work, the parent components should expose their child's dependency. Let's take a look at an example.

I think this here is evident to how our GXB codebase is structured?

```
@WarriorScreenScope
@Component(modules = [WarriorScreenModule::class],
        [[AppComponent::class])
interface WarriorScreenComponent {
    fun inject(warriorActivity: WarriorActivity)
}
```

Funny how we didn't do try this out in our refactoring attempt where we assigned a value to our Component's "dependencies" annotation argument(Or did we?)
Bring this up to the Android Devs to discuss

First, instantiate the parent components -> pass their instances in our component.
the best approach as discussed previously is to save the instance of your global components + using them in you child components.
This will allow you to use the scoped objects in those components.

Subcomponent

- another way of building component relationships.
- ***This can be thought of as something similar in concept to inner/outer class relationship between classes in OOP.***
- A component can only have 1 parent BUT parent can be depended on by multiple subcomponents.
    - you cannot specify your parent BUT Your parent (or the ancestors) should ensure that it has all its child's dependency (aside from the modules of course).
    - You can get the subcomponents from the parent components (and provide the parameters of course).

How to use subcomponents.

- The parent should provide a method to get their child components.
- The code example exposes the WarriorScreenComponent from the AppComponent.
- This will allow us to instantiate WarriorScreenComponent from AppComponent.