Dependency Injection with Hilt

Download the full App Created in this Guide:

https://drive.google.com/file/d/1C2mejEMsaIzfdJZ8BtF3WTCUv7SCKFLj/view?usp=sharing

A dependency is an object that another object requires. In other words, the latter object depends on the former for it to function. For example, a Car class might need a reference to an Engine class.

There are three ways for a class to get an object it needs:
1. The class constructs the dependency it needs. In the example above, Car would create and initialize its own instance of Engine.
2. Grab it from somewhere else. Some Android APIs, such as Context getters and getSystemService(), work this way.
3. Have it supplied as a parameter. The app can provide these dependencies when the class is constructed or pass them in to the functions that need each dependency. In the example above, the Car constructor would receive Engine as a parameter - **this dependency injection!** With this approach you take the dependencies of a class and provide them rather than having the class instance obtain them herself.

Dependency Injection is whereby dependencies are provided to a class instead of the class having to create them itself. Hilt is a standardized way of enforcing dependency injection in an Android application.

In the first two options Car and Engine are tightly coupled - an instance of Car uses one type of Engine, **and no subclasses or alternative implementations can easily be used.** If the Car were to construct its own Engine, you would have to create two types of Car instead of just reusing the same Car for engines of type Gas and Electric. It also makes the tests much harder because **Car must have a real instance of engine thus preventing us from the ability to mock it.**

Without dependency injection:

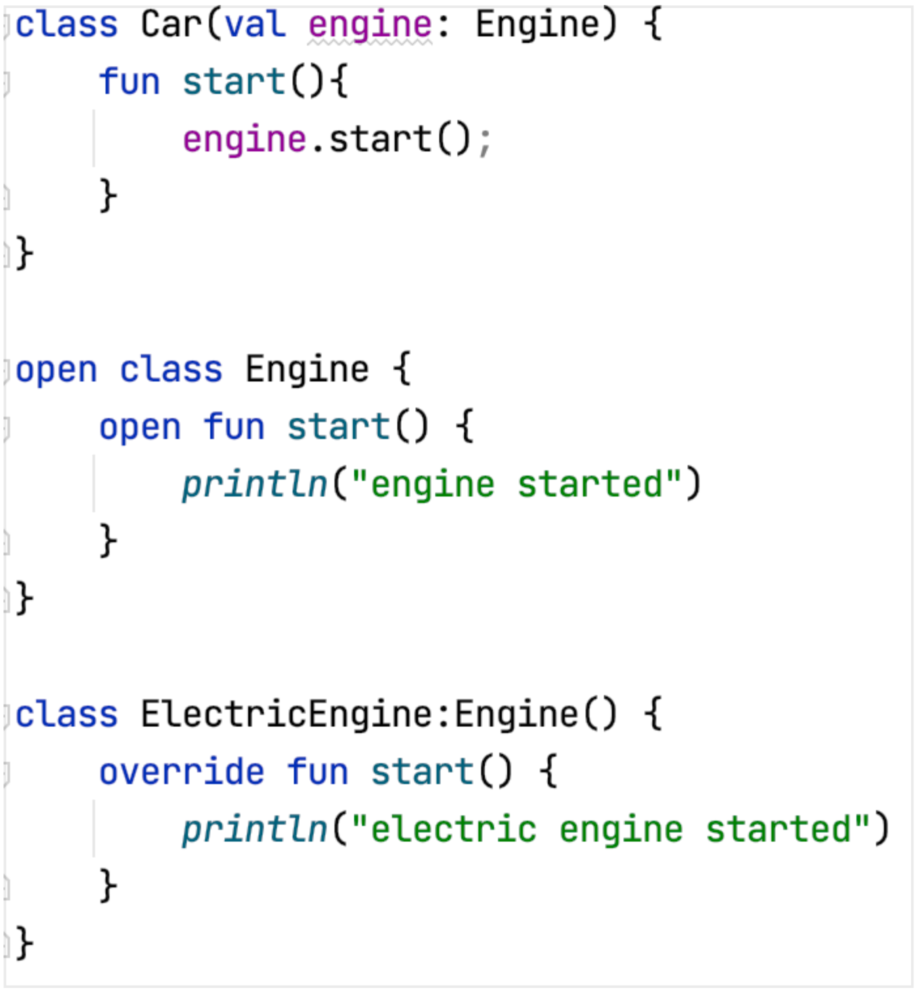
```kotlin
class Car {

    private val engine = Engine()

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.start()
}
```

With dependency injection

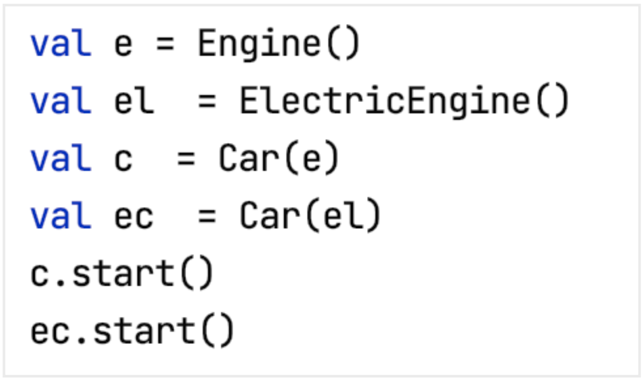
```kotlin
class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```

Write these classes in a new android studio project.

Lets look at the Reusability of Car:
You can pass in different implementations of Engine to Car. For example, you might define a new subclass of Engine called ElectricEngine that you want Car to use. If you use DI, all you need to do is pass in an instance of the updated ElectricEngine subclass, and Car still works without any further changes.



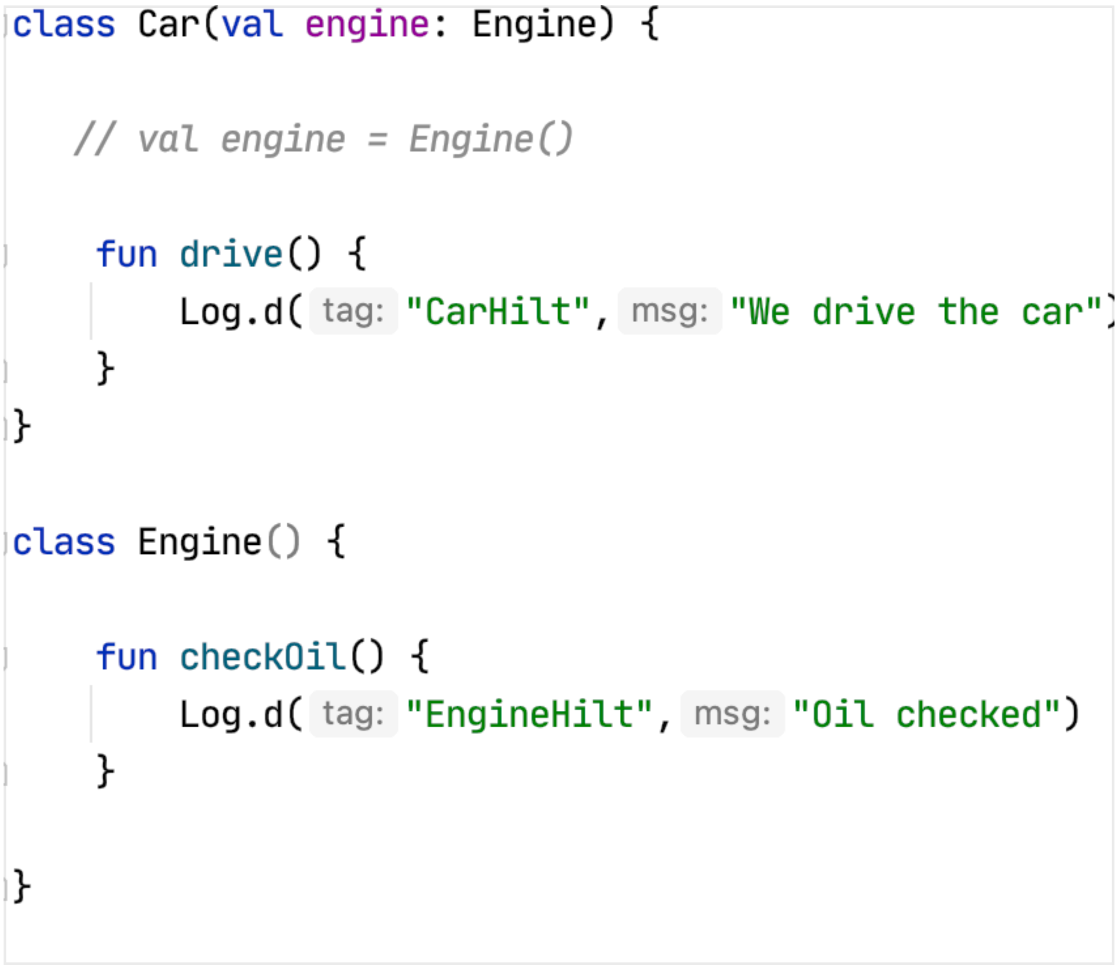
```kotlin
class Car(val engine: Engine) {
    fun start(){
        engine.start();
    }
}

open class Engine {
    open fun start() {
        println("engine started")
    }
}

class ElectricEngine:Engine() {
    override fun start() {
        println("electric engine started")
    }
}
```



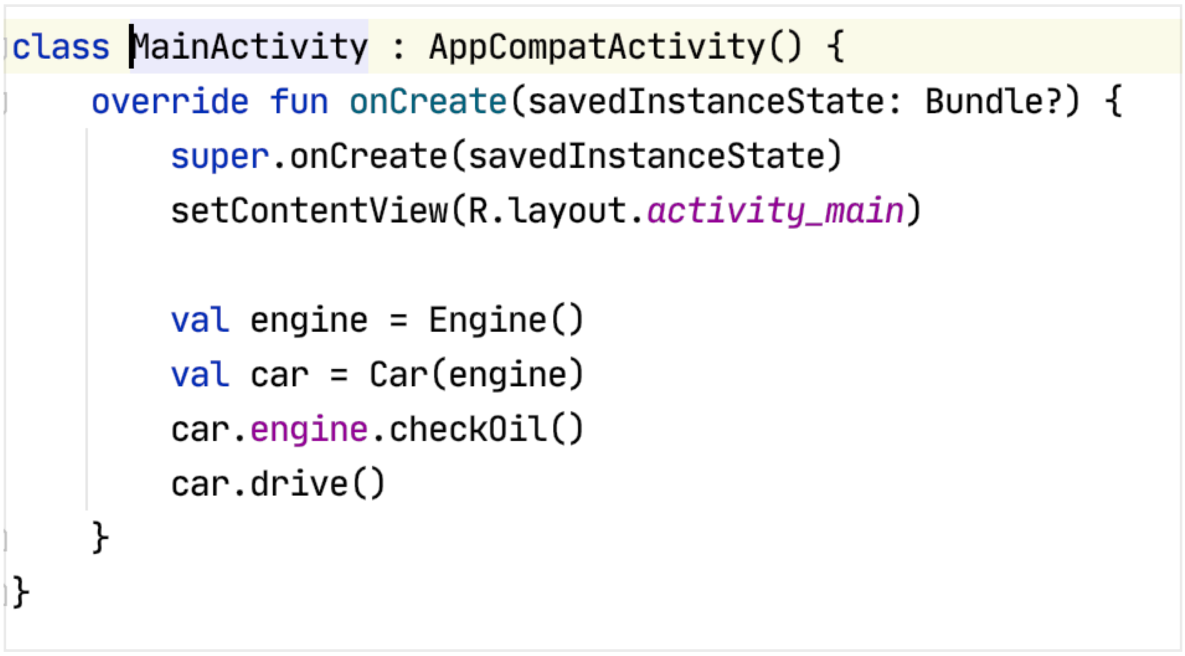
```kotlin
val e = Engine()
val el  = ElectricEngine()
val c  = Car(e)
val ec  = Car(el)
c.start()
ec.start()
```

There are two major ways to do dependency injection in Android:

- **Constructor Injection**. This is the way described above. You pass the dependencies of a class to its constructor.



```
class Car(val engine: Engine) {

    // val engine = Engine()

    fun drive() {
        Log.d( tag: "CarHilt", msg: "We drive the car"
    }
}

class Engine() {

    fun checkOil() {
        Log.d( tag: "EngineHilt", msg: "Oil checked")
    }

}
```



```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val engine = Engine()
        val car = Car(engine)
        car.engine.checkOil()
        car.drive()
    }
}
```

- **Field Injection (or Setter Injection)**. Certain Android framework classes such as activities and fragments are instantiated by the system, so constructor injection is not possible. With field injection, dependencies are instantiated after the class is created. The code would look like this:

```kotlin
class Car {
    lateinit var engine: Engine

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.engine = Engine()
    car.start()
}
```

In all the examples above we did the dependency injection manually

Automated dependency injection

In the previous example, you created, provided, and managed the dependencies of the different classes yourself, without relying on a library. This is called *dependency injection by hand*, or ***manual dependency injection*.** In the Car example, there was only one dependency, but more dependencies and classes can make manual injection of dependencies more tedious. Also When you're not able to construct dependencies before passing them in — for example when using lazy initializations — you need to write and maintain a custom container (or graph of dependencies) that manages the lifetimes of your dependencies in memory.

Look at this example to see fully manual dependency injection
https://developer.android.com/training/dependency-injection/manual

There are libraries that solve this problem by automating the process of creating and providing dependencies. They fit into two categories:

- Reflection-based solutions that connect dependencies at runtime. Example of this solution is using the **Guice library.**
- Static solutions that generate the code to connect dependencies at compile time. Example is using Dagger2 library which is now managed by Google and a jetpack composed library called **Hilt**.

Dagger is a popular dependency injection library for Java, Kotlin, and Android that is maintained by Google. Dagger facilitates using DI in your app by creating and managing the graph of dependencies for you. It provides fully static and compile-time dependencies

Use Hilt in your Android app

Hilt is Jetpack's recommended library for dependency injection in Android. Hilt defines a standard way to do DI in your application by **providing containers for every Android class in your project and managing their lifecycles automatically for you.**

Hilt is built on top of the popular DI library Dagger to benefit from the compile time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

Using Hilt

First, add the hilt-android-gradle-plugin plugin to your **project's root** build.gradle file:

```
 dependencies {
        ...
        classpath 'com.google.dagger:hilt-android-gradle-plugin:2.38.1'
    }
```

Then, apply the Gradle plugin and add these dependencies in your app/build.gradle file:

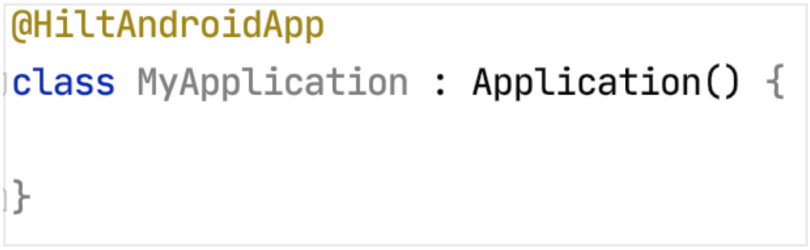
```
id 'kotlin-kapt'
id 'dagger.hilt.android.plugin'


dependencies {
    implementation "com.google.dagger:hilt-android:2.38.1"
    kapt "com.google.dagger:hilt-compiler:2.38.1"
}
```

Make sure Java 8 is enabled (Hilt uses Java 8) - in you app Gradle file

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

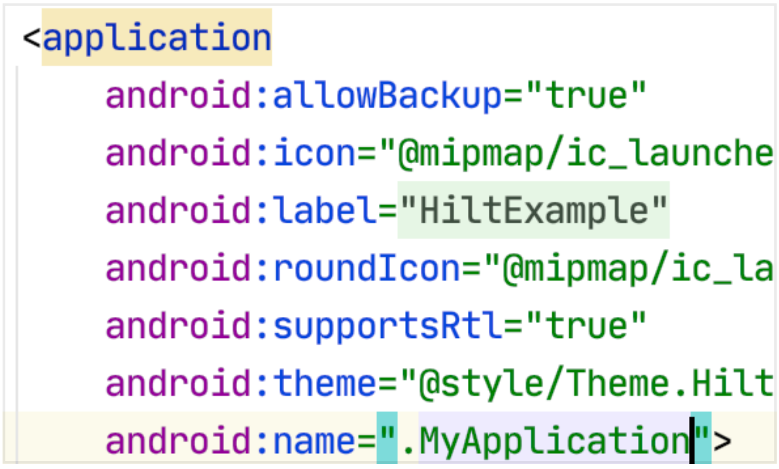
First step

Extend the Application class and annotate it with **@HiltAndroidApp** this triggers Hilt's code generation, including a base class for your application that serves as the application-level dependency container.



```
@HiltAndroidApp
class MyApplication : Application() {

}
```

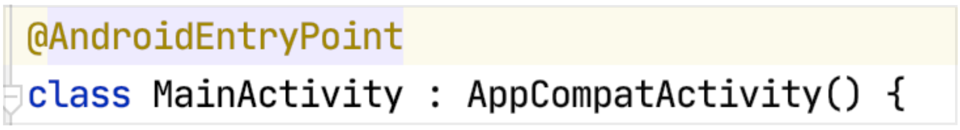
Don't forget to add the application name to the manifest file



```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launche
    android:label="HiltExample"
    android:roundIcon="@mipmap/ic_la
    android:supportsRtl="true"
    android:theme="@style/Theme.Hilt
    android:name=".MyApplication">
```

This generated Hilt component is attached to the Application object's lifecycle and provides dependencies to it. Additionally, it is the parent component of the app, which means that other components can access the dependencies that it provides.

Once Hilt is set up in your Application class and an application-level component is available, Hilt can provide dependencies to other Android classes that have the @AndroidEntryPoint annotation:



```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
```

Hilt currently supports the following Android classes:

- Application (by using @HiltAndroidApp)
- ViewModel (by using @HiltViewModel)
- Activity
- Fragment
- View
- Service
- BroadcastReceiver

If you annotate an Android class with @AndroidEntryPoint, then you also must annotate Android classes that depend on it. For example, if you annotate a fragment, then you must also annotate any activities where you use that fragment. Classes that Hilt injects can have other base classes that also use injection. Those classes don't need the @AndroidEntryPoint annotation if they're abstract.

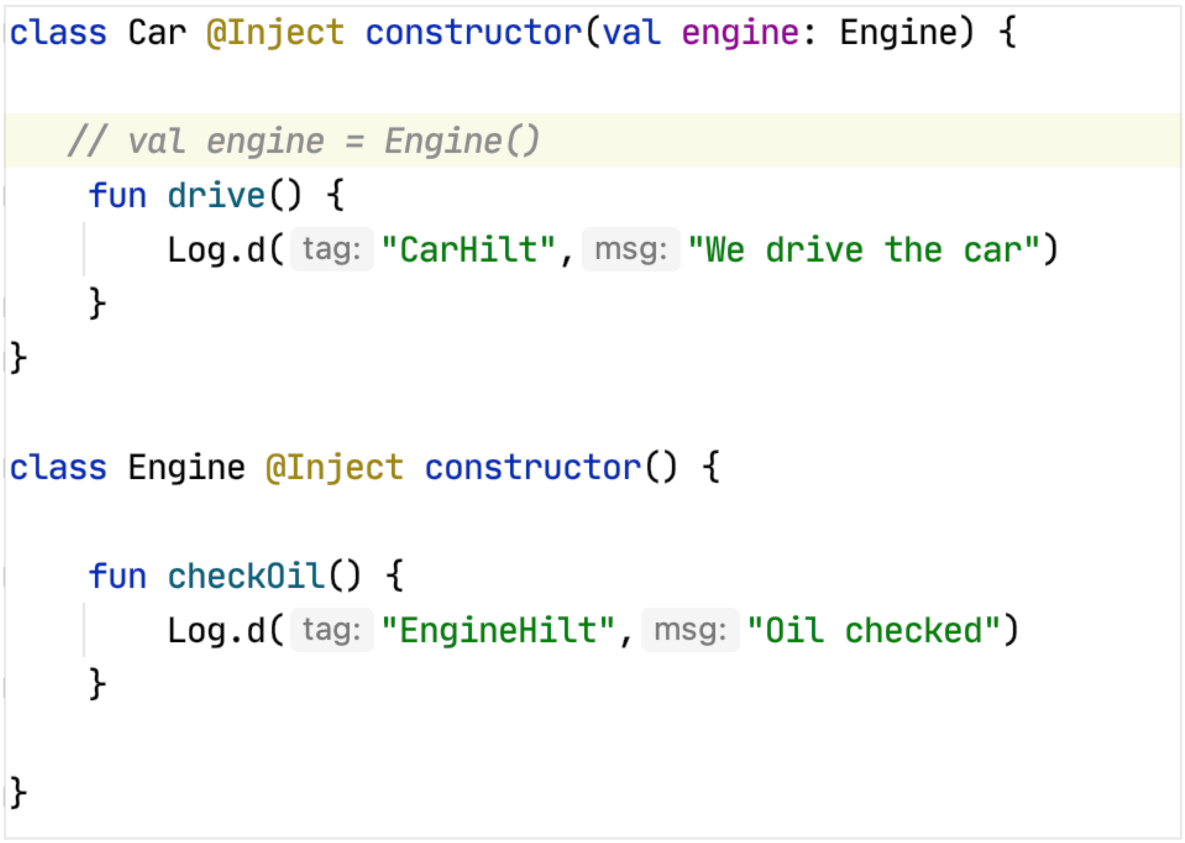
@AndroidEntryPoint generates an individual Hilt component for each Android class in your project. It turn them into dependency containers.

Define Hilt bindings

To perform field injection, Hilt needs to know how to provide instances of the necessary dependencies from the corresponding component. An **Hilt *binding*** contains the information necessary to provide instances of a type as a dependency.

One way to provide binding information to Hilt is *constructor injection*. Use the @Inject annotation on the constructor of a class to tell Hilt how to provide instances of that class.

Our full code will look like this now:

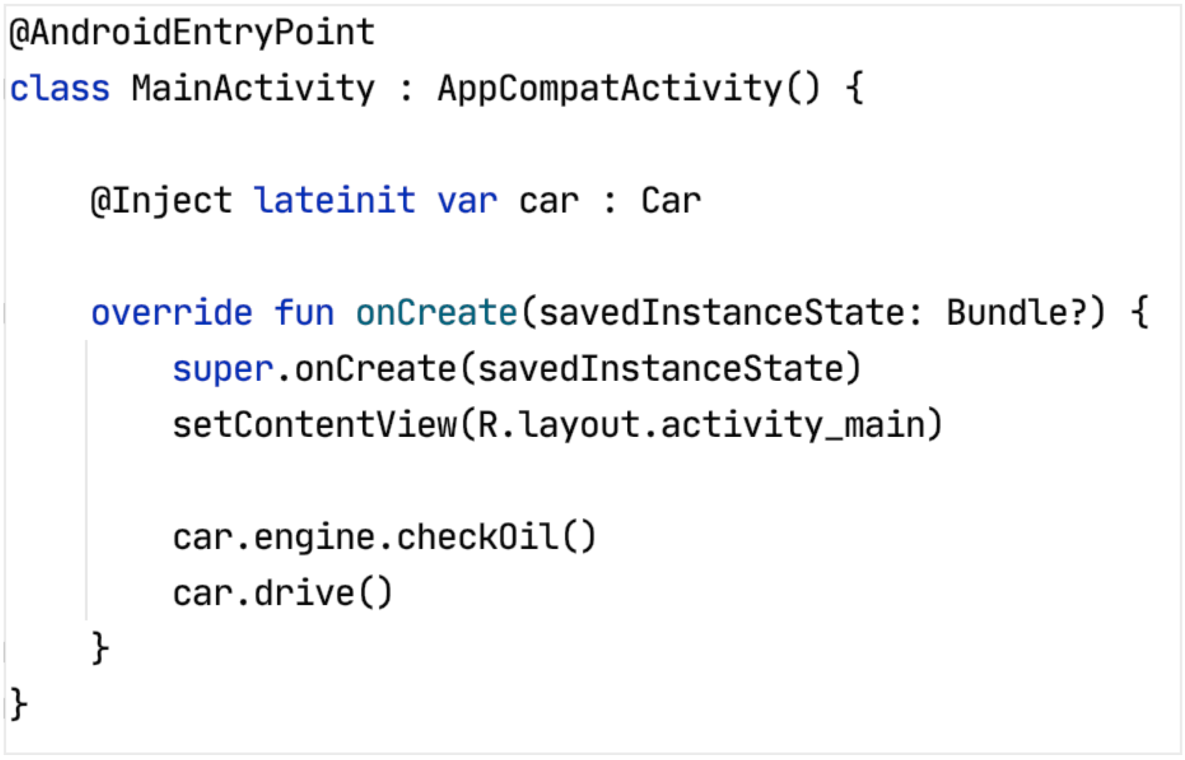


```kotlin
class Car @Inject constructor(val engine: Engine) {

    // val engine = Engine()
    fun drive() {
        Log.d( tag: "CarHilt", msg: "We drive the car")
    }
}

class Engine @Inject constructor() {

    fun checkOil() {
        Log.d( tag: "EngineHilt", msg: "Oil checked")
    }

}
```

Here @Inject gives Hilt access to the the necessary constructors meaning it can generate instances of both Car and Engine. If Engine is a parameter of the Injected constructor then Hilt must also know how to create instances of it (the Engine class).

Instances that Hilt knows how to create go by the name bindings. So Car and Engine are bindings.

And the Activity which is holding the Injectable Fields will eventually look like this:



```kotlin
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject lateinit var car : Car

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        car.engine.checkOil()
        car.drive()
    }
}
```

To obtain dependencies from a component, use the **@Inject** annotation to perform field injection.
Here the @Inject annotation goes by a different meaning. Here it means the car field is injectable field. Injectable means that Hilt can supply the instantiated dependencies to it.
Please note that Fields injected by Hilt cannot be private. Attempting to inject a private field with Hilt results in a compilation error.

Hilt modules

Sometimes a type cannot be constructor-injected. This can happen for multiple reasons. For example, you cannot constructor-inject an interface. You also cannot constructor-inject a type that you do not own, such as a class from an external library. In these cases, you can provide Hilt with binding information by using *Hilt modules*.

A Hilt module is a class that is annotated with **@Module** it informs Hilt how to provide instances of certain types. you must annotate Hilt modules with @InstallIn to tell Hilt which Android class each module will be used or installed in. This determine the dependency lifetime scope.

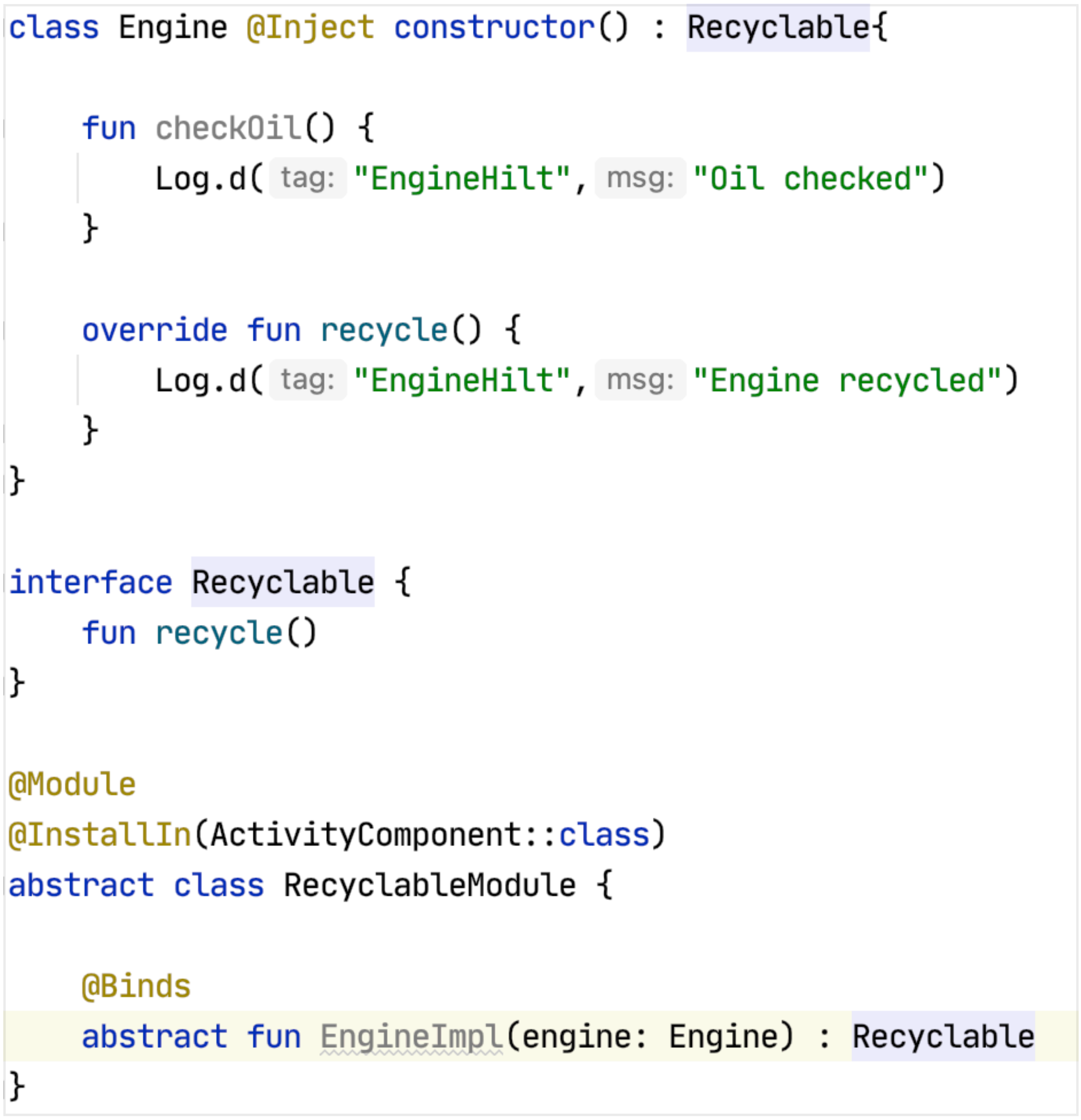
If you want the dependency to exist in all of your app activities use **@InstallIn(ActivityComponent::class)**. Later we will see all the available scopes.

Hilt can't generate a constructor for an interface. Instead, provide Hilt with the binding information by creating an abstract function annotated with **@Binds** inside a Hilt module.

The @Binds annotation tells Hilt which implementation to use when it needs to provide an instance of an interface.
The annotated function provides the following information to Hilt:

- The function return type tells Hilt what interface the function provides instances of.
- The function parameter tells Hilt which implementation to provide.

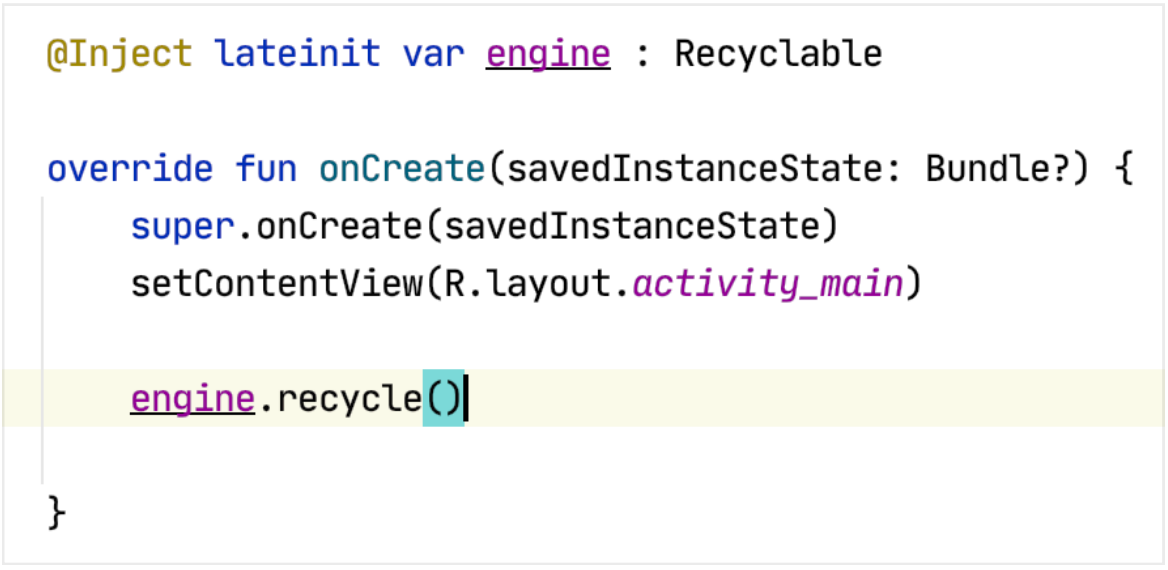
Therefor our code will look like this:



```
class Engine @Inject constructor() : Recyclable{

    fun checkOil() {
        Log.d( tag: "EngineHilt", msg: "Oil checked")
    }

    override fun recycle() {
        Log.d( tag: "EngineHilt", msg: "Engine recycled")
    }
}

interface Recyclable {
    fun recycle()
}

@Module
@InstallIn(ActivityComponent::class)
abstract class RecyclableModule {

    @Binds
    abstract fun EngineImpl(engine: Engine) : Recyclable
}
```

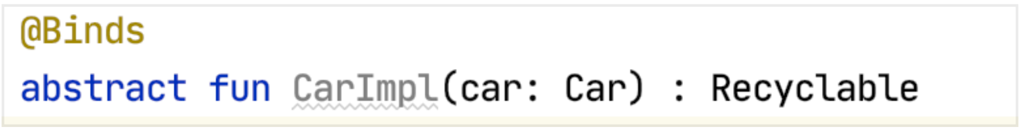
In our example when someone requires a Recyclable then we will return an Engine

And in the Main Activity we change the code like that:



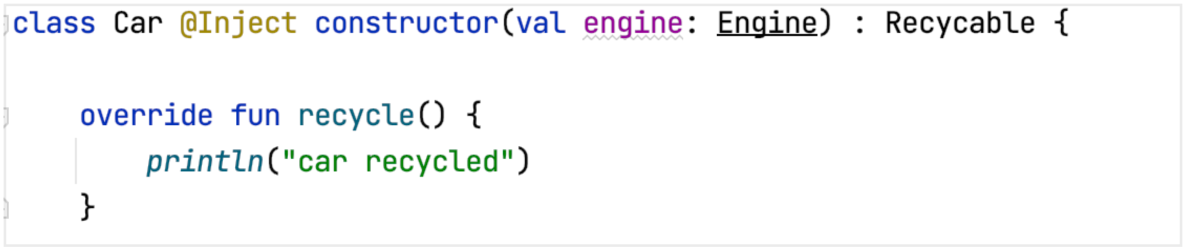
```kotlin
@Inject lateinit var engine : Recyclable

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()

}
```

Now what happens if we want different implementation for car and for engine. If we try to add the following function to out Module and run your code.



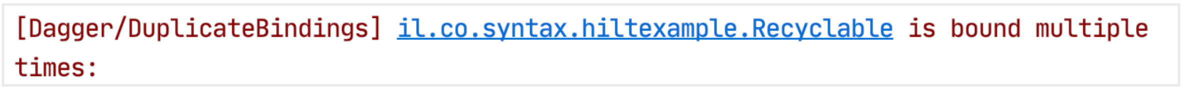
```kotlin
@Binds
abstract fun CarImpl(car: Car) : Recyclable
```

And of course make car also implement the Recyclable interface



```kotlin
class Car @Inject constructor(val engine: Engine) : Recycable {

    override fun recycle() {
        println("car recycled")
    }
```

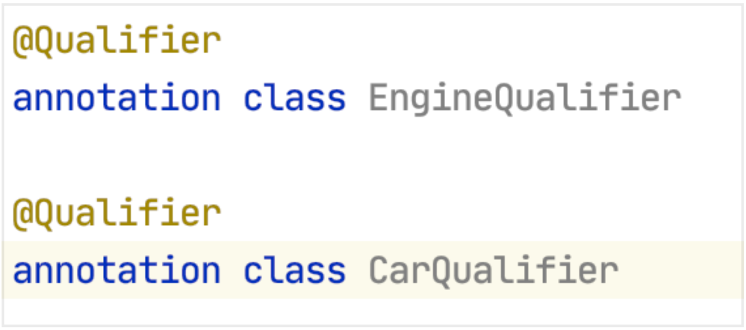
We will get the following error when we try to execute our code:



```
[Dagger/DuplicateBindings] il.co.syntax.hiltexample.Recyclable is bound multiple
times:
```

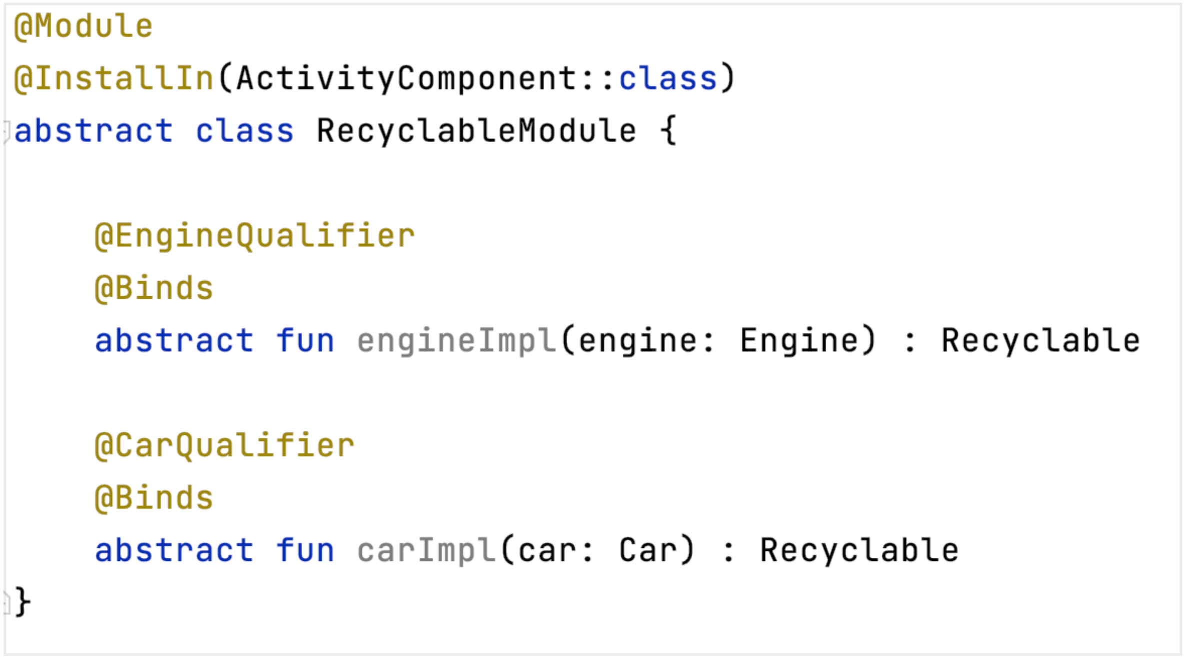
Hilt doesn’t know which implementation to use and We need to differentiate them somehow.
This is why we have the @Qualifier Annotation

Create the following Annotation Quailifiers next to you Module they will use you to differentiate the implementations



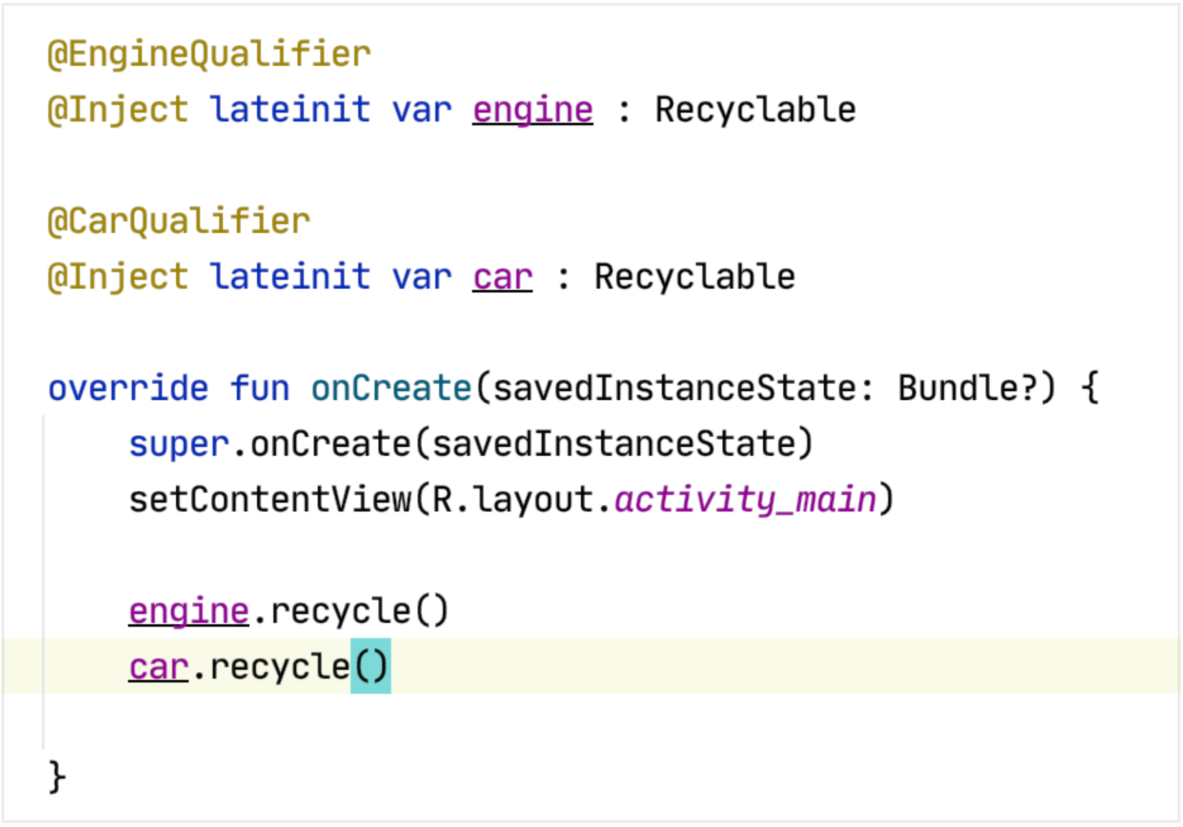
```kotlin
@Qualifier
annotation class EngineQualifier

@Qualifier
annotation class CarQualifier
```

And add the Qualifier next to the implementations like this:



```kotlin
@Module
@InstallIn(ActivityComponent::class)
abstract class RecyclableModule {

    @EngineQualifier
    @Binds
    abstract fun engineImpl(engine: Engine) : Recyclable

    @CarQualifier
    @Binds
    abstract fun carImpl(car: Car) : Recyclable
}
```

Also add them next to the reference definition in you MainActivity file



```kotlin
@EngineQualifier
@Inject lateinit var engine : Recyclable

@CarQualifier
@Inject lateinit var car : Recyclable

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()
    car.recycle()

}
```

Interfaces are not the only case where you cannot constructor-inject a type. Constructor injection is also not possible if you don't own the class because it comes from an external library (classes like Retrofit or Room databases), or if instances must be created with the builder pattern.

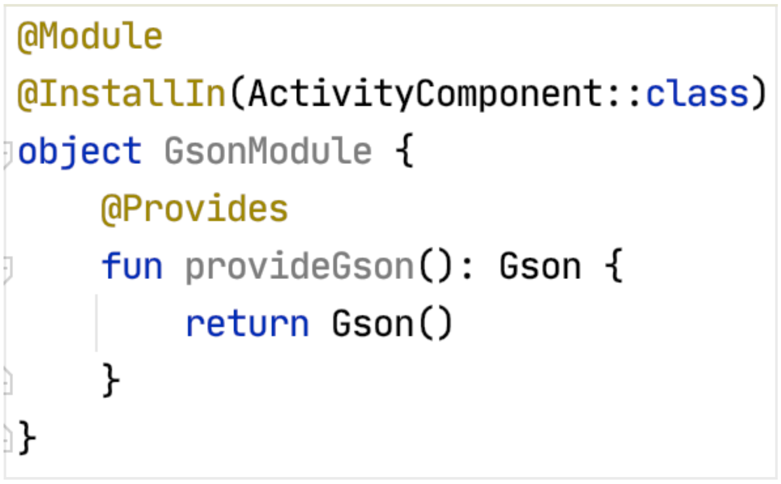
We can tell Hilt how to provide instances of a type by creating a function inside a Hilt module and annotating that function with **@Provides**.

Lets take for example a simple library called Gson to covert string to json and vice versa

Add the following dependency to your project

```
implementation 'com.google.code.gson:gson:2.8.6'
```

And create the following Gson module:

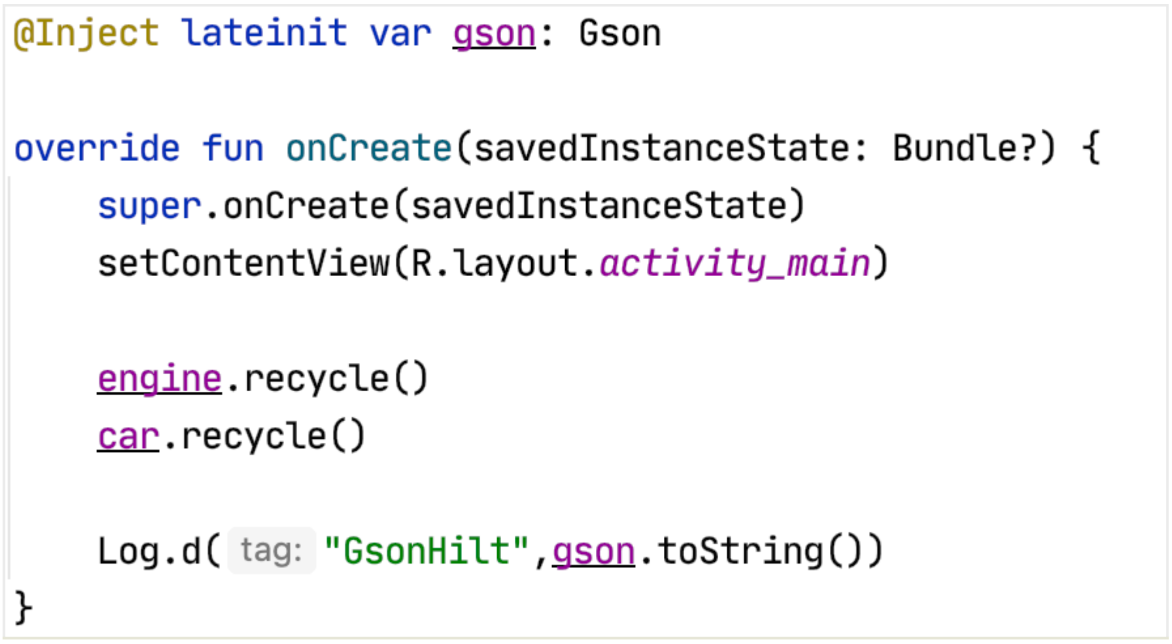


```kotlin
@Module
@InstallIn(ActivityComponent::class)
object GsonModule {
    @Provides
    fun provideGson(): Gson {
        return Gson()
    }
}
```

Through @Provides, the annotated function gives Hilt the following information:

- The return type tells Hilt what type the function provides instances of.
- The parameters tell Hilt the dependencies required to provide the type. In our case, there are none.
- The function body tells Hilt how to provide an instance of the corresponding type. Hilt executes the function body every time it needs to provide an instance of that type.

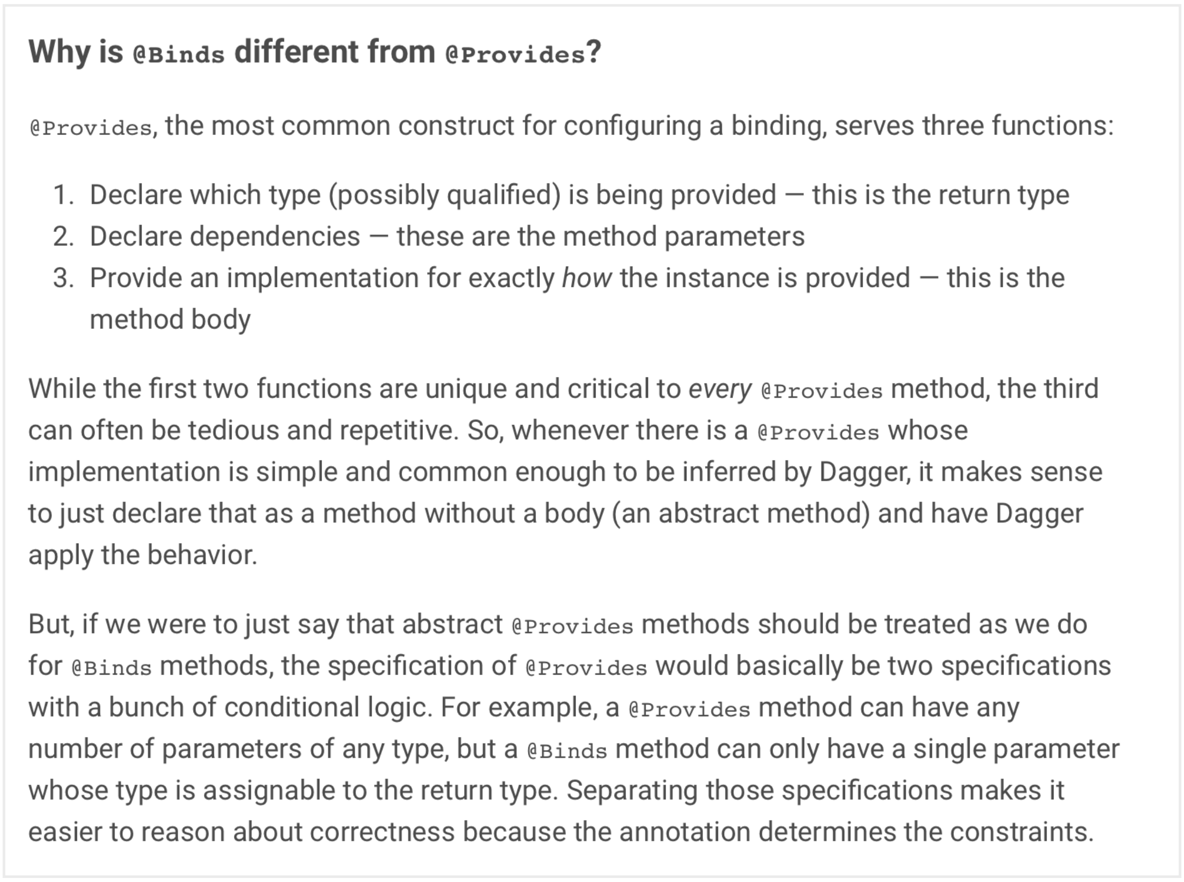
In the MainActivity add the following code



```kotlin
@Inject lateinit var gson: Gson

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()
    car.recycle()

    Log.d( tag: "GsonHilt",gson.toString())
}
```

We have successfully injected 3rd library dependency!

A bit more about @Binds and @Provides :



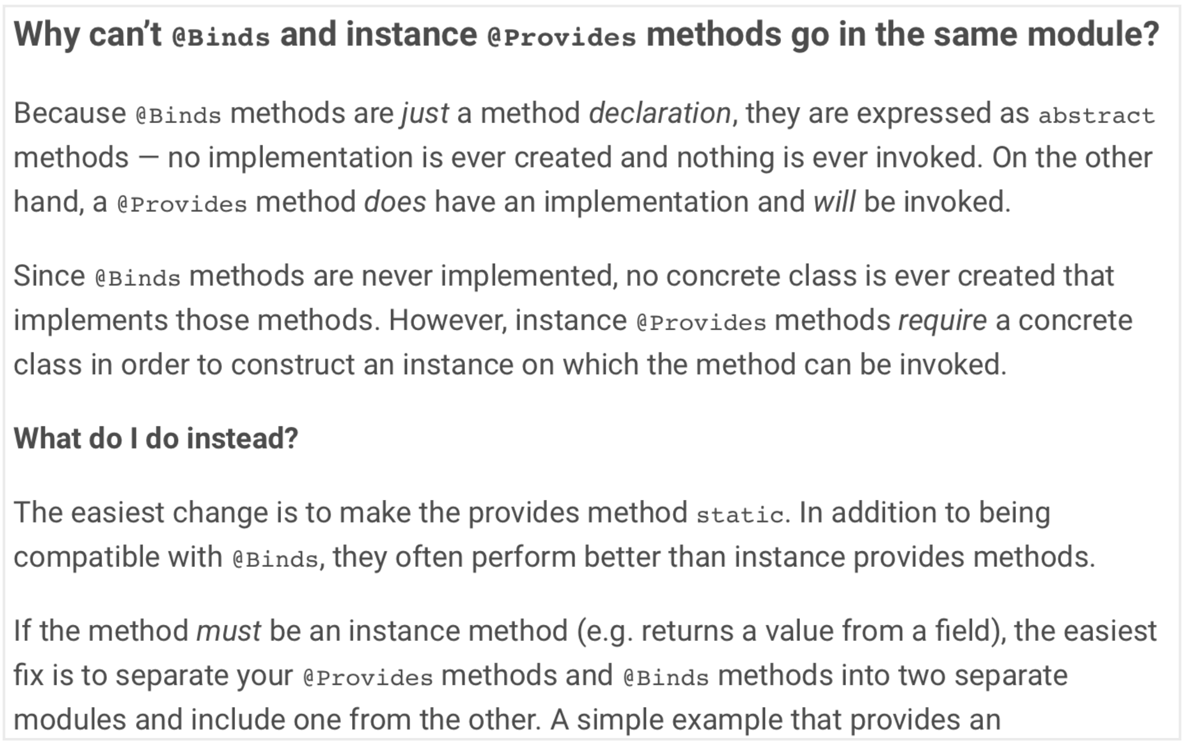
Why is `@Binds` different from `@Provides`?

`@Provides`, the most common construct for configuring a binding, serves three functions:

1. Declare which type (possibly qualified) is being provided — this is the return type
2. Declare dependencies — these are the method parameters
3. Provide an implementation for exactly *how* the instance is provided — this is the method body

While the first two functions are unique and critical to *every* `@Provides` method, the third can often be tedious and repetitive. So, whenever there is a `@Provides` whose implementation is simple and common enough to be inferred by Dagger, it makes sense to just declare that as a method without a body (an abstract method) and have Dagger apply the behavior.

But, if we were to just say that abstract `@Provides` methods should be treated as we do for `@Binds` methods, the specification of `@Provides` would basically be two specifications with a bunch of conditional logic. For example, a `@Provides` method can have any number of parameters of any type, but a `@Binds` method can only have a single parameter whose type is assignable to the return type. Separating those specifications makes it easier to reason about correctness because the annotation determines the constraints.



Why can't `@Binds` and instance `@Provides` methods go in the same module?

Because `@Binds` methods are *just* a method *declaration*, they are expressed as `abstract` methods — no implementation is ever created and nothing is ever invoked. On the other hand, a `@Provides` method *does* have an implementation and *will* be invoked.

Since `@Binds` methods are never implemented, no concrete class is ever created that implements those methods. However, instance `@Provides` methods *require* a concrete class in order to construct an instance on which the method can be invoked.

What do I do instead?

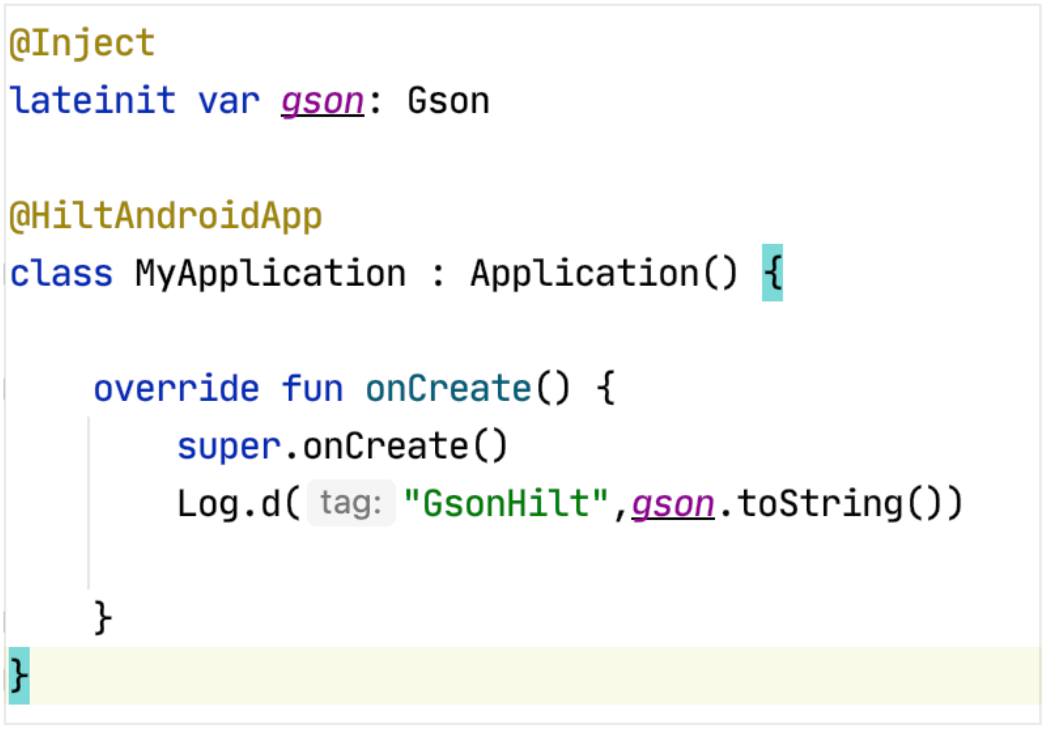
The easiest change is to make the provides method `static`. In addition to being compatible with `@Binds`, they often perform better than instance provides methods.

If the method *must* be an instance method (e.g. returns a value from a field), the easiest fix is to separate your `@Provides` methods and `@Binds` methods into two separate modules and include one from the other. A simple example that provides an

As you can see @Binds functions are abstract while @Provides functions have a body. With @Binds the implementation is obvious. @Binds method can only have a single parameter whose type is assignable to the return type. @Provides method can have any number of parameters of any type.

More on @InstallIn

Now let’s look at the scope again, try writing the same lines in you Application class. You will get an error!!

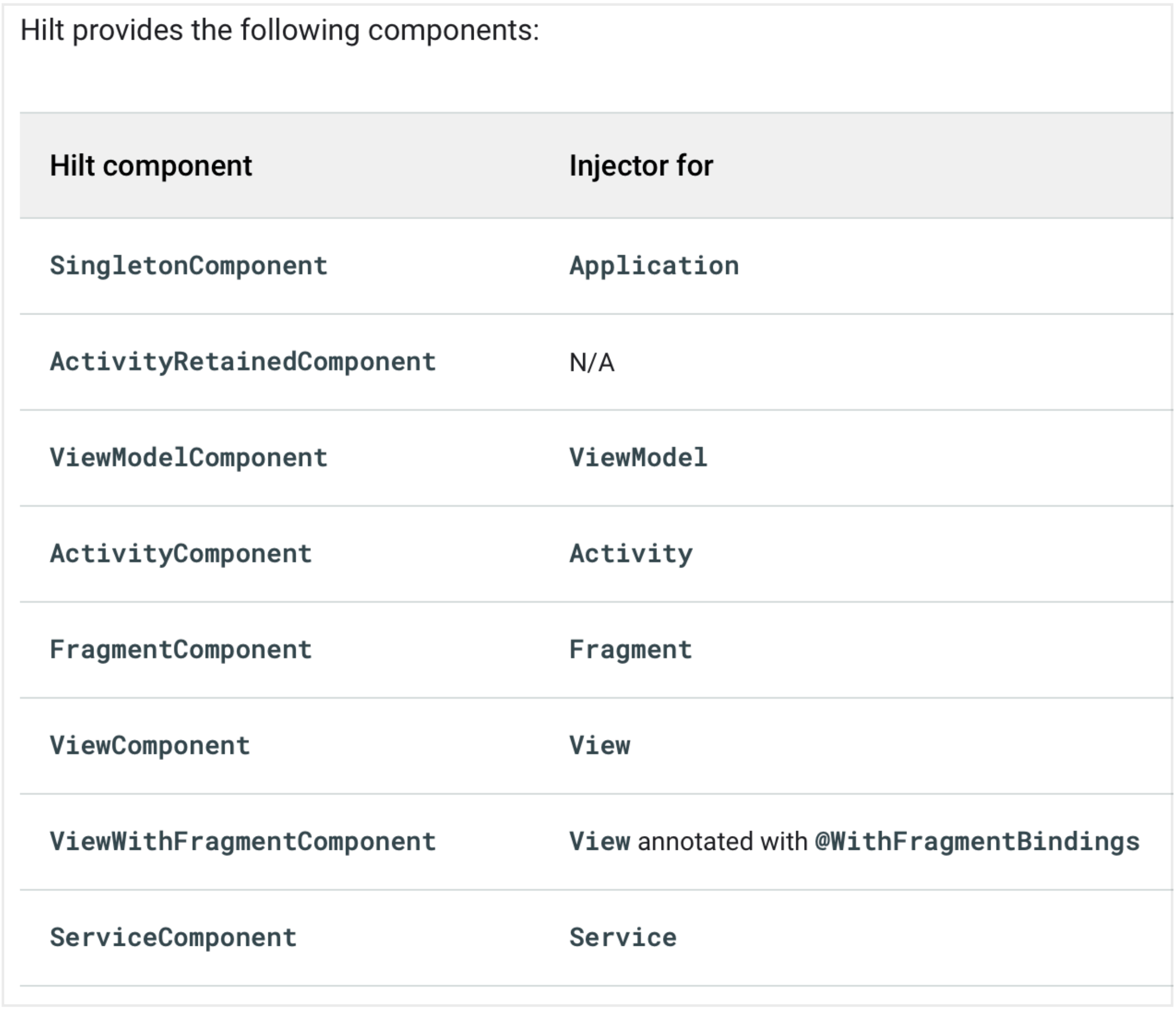


```kotlin
@Inject
lateinit var gson: Gson

@HiltAndroidApp
class MyApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        Log.d( tag: "GsonHilt",gson.toString())

    }
}
```

Do you remember the discussion on components? If you look at the GsonModule component, it is installed in the ActivityComponent.class. Therefore, it is only available during the lifetime of an activity rather than that of the entire application.

For each Android class in which you can perform field injection, there's an associated Hilt component that you can refer to in the @InstallIn annotation. Each Hilt component is responsible for injecting its bindings into the corresponding Android class.

The previous examples demonstrated the use of ActivityComponent in Hilt modules.



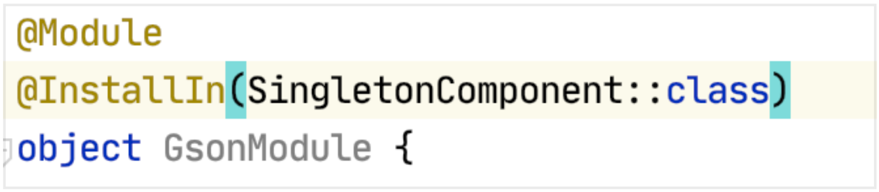
Hilt provides the following components:

Hilt component	Injector for
`SingletonComponent`	`Application`
`ActivityRetainedComponent`	N/A
`ViewModelComponent`	`ViewModel`
`ActivityComponent`	`Activity`
`FragmentComponent`	`Fragment`
`ViewComponent`	`View`
`ViewWithFragmentComponent`	`View` annotated with `@WithFragmentBindings`
`ServiceComponent`	`Service`

Hilt automatically creates and destroys instances of generated component classes following the lifecycle of the corresponding Android classes(for example, all the activity components will be destroyed by hilt in the activity onDestroy() method)

When it comes to classes such as Gson, Retrofit and Room database, we may need to make them available to the entire application.

To correct this error, change the ActivityComponent.class to SingletonComponent.class - The error is gone



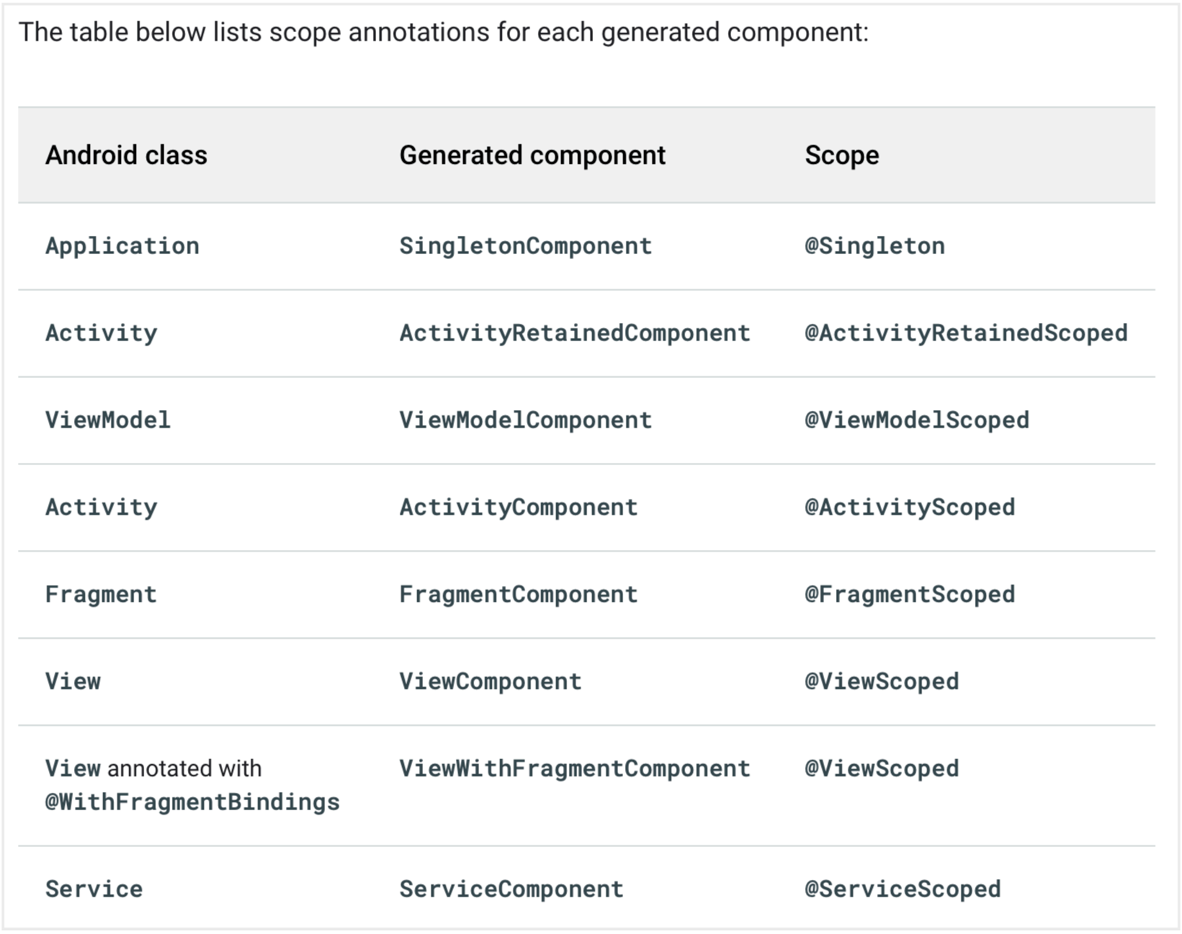
```
@Module
@InstallIn(SingletonComponent::class)
object GsonModule {
```

But is the Gson object the same in MyApplication and MainActivity? No.

Scoping

Bindings in Hilt are naturally **unscoped**. This means that each time your app requests the binding (the dependency), **Hilt creates a new instance of the needed type.**

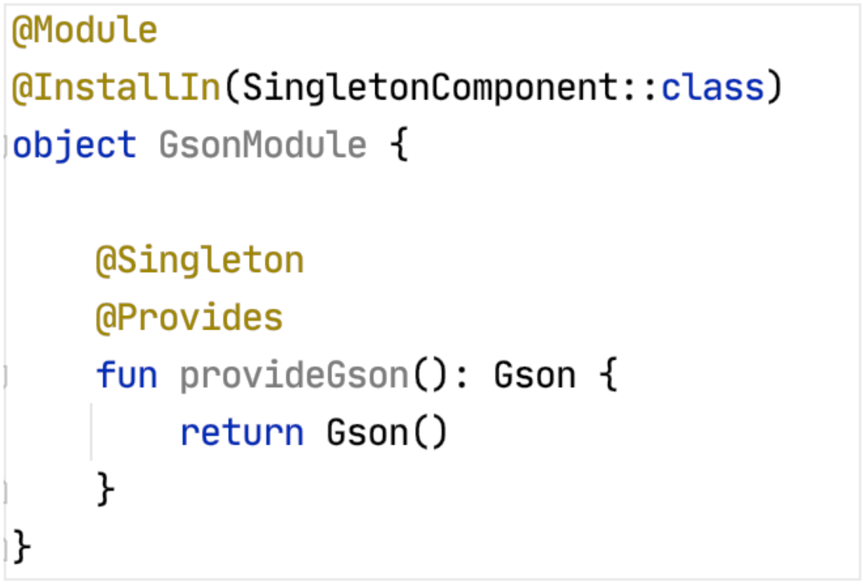
However, Hilt also allows a binding to be scoped to a particular component. Hilt only creates a scoped binding once per instance of the component that the binding is scoped to, and all requests for that binding share the same instance.



The table below lists scope annotations for each generated component:

Android class	Generated component	Scope
`Application`	`SingletonComponent`	`@Singleton`
`Activity`	`ActivityRetainedComponent`	`@ActivityRetainedScoped`
`ViewModel`	`ViewModelComponent`	`@ViewModelScoped`
`Activity`	`ActivityComponent`	`@ActivityScoped`
`Fragment`	`FragmentComponent`	`@FragmentScoped`
`View`	`ViewComponent`	`@ViewScoped`
`View` annotated with `@WithFragmentBindings`	`ViewWithFragmentComponent`	`@ViewScoped`
`Service`	`ServiceComponent`	`@ServiceScoped`

To ensure only one instance of Gson is available at a time, modify GsonModule and add @Singleton annotation used to ensure that the generated instance is the only one throughout the application's lifecycle.



```kotlin
@Module
@InstallIn(SingletonComponent::class)
object GsonModule {

    @Singleton
    @Provides
    fun provideGson(): Gson {
        return Gson()
    }
}
```

Because we scoped the GsonModule to the SingletonComponent using @Singleton Hilt provides the same instance of GsonModule throughout the life of the entire application.

ActivityScoped ensures that the instance is the same throughout the activity. Same if we would have scoped any other adapter or module to the ActivityComponent using @ActivityScoped, then Hilt would have provided the same instance of that module throughout the life of the corresponding activity.

```
@ActivityScoped
class AnalyticsAdapter @Inject constructor(
  private val service: AnalyticsService
) { ... }
```

Please note that Scoping a binding to a component can be costly because the provided object stays in memory until that component is destroyed.

To summarize the interfaces and the 3rd library in oppose to interfaces or class we own but can't call their constructor - this it how to obtain a single instance of the AnalyticSercive:

```
// If AnalyticsService is an interface.
@Module
@InstallIn(SingletonComponent::class)
abstract class AnalyticsModule {

  @Singleton
  @Binds
  abstract fun bindAnalyticsService(
    analyticsServiceImpl: AnalyticsServiceImpl
  ): AnalyticsService
}

// If you don't own AnalyticsService.
@Module
@InstallIn(SingletonComponent::class)
object AnalyticsModule {

  @Singleton
  @Provides
  fun provideAnalyticsService(): AnalyticsService {
      return Retrofit.Builder()
               .baseUrl("https://example.com")
               .build()
               .create(AnalyticsService::class.java)
  }
}
```

Scoping and ViewModels

Originally if no scoping is done then activity retain a new instance upon each configuration change. Like this:

```
class ExampleActivity : AppCompatActivity() {
  private val analyticsAdapter = AnalyticsAdapter()
  ...
}
```

In Hilt this will look like this:

```kotlin
@ActivityScoped
class AnalyticsAdapter @Inject constructor() { ... }


@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() {


  @Inject lateinit var analyticsAdapter: AnalyticsAdapter


}
```

The AnalyticsAdapter a scoped here to the Activity. When a new instance of ExampleActivity is created (e.g. the activity goes through a configuration change), a new instance of AnalyticsAdapter will be created.

To get the same instance we can achieve that through view models or with Hilt (with or without ViewModels)

With View Models

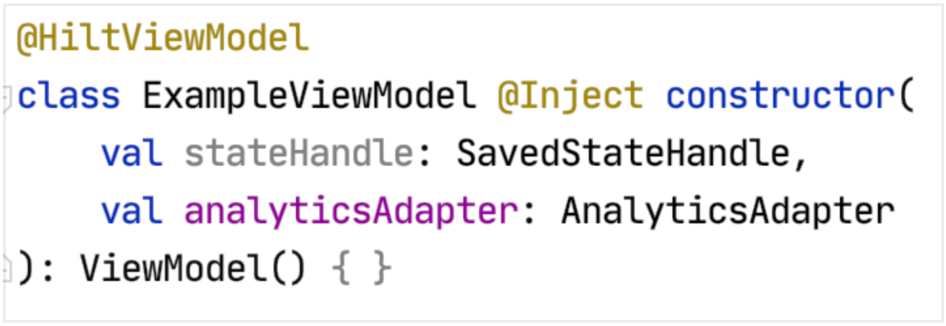
```kotlin
class AnalyticsAdapter() { ... }


class ExampleViewModel() : ViewModel() {
  val analyticsAdapter = AnalyticsAdapter()
}


class ExampleActivity : AppCompatActivity() {


  private val viewModel: ExampleViewModel by viewModels()
  private val analyticsAdapter = viewModel.analyticsAdapter


}
```

With Hilt (No ViewModels) we use **@ActivityRetainedScoped** that scope AnalyticsAdapter to the **ActivityRetainedComponent which also survives configuration changes**

```
@ActivityRetainedScoped
class AnalyticsAdapter @Inject constructor() { ... }

@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() {

  @Inject lateinit var analyticsAdapter: AnalyticsAdapter

}
```

With Hilt and View Models - There is one major difference:

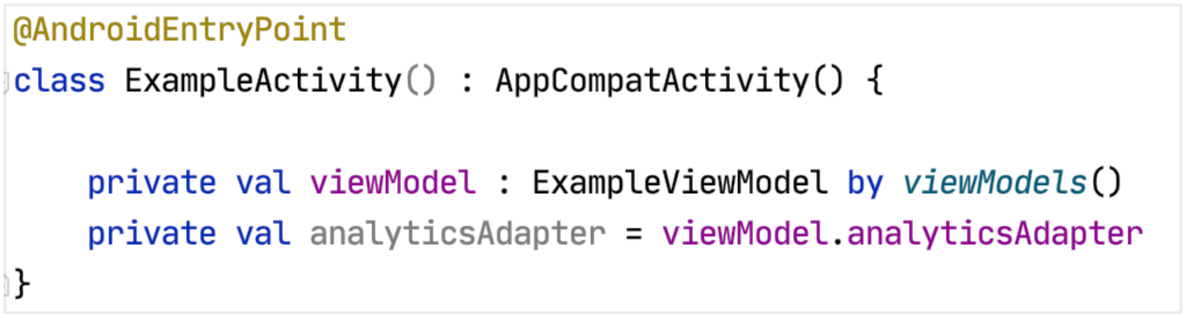
First, A Hilt View Model is a Jetpack ViewModel which his constructor injected by Hilt. To enable injection of a ViewModel by Hilt use the @HiltViewModel annotation:



```
@HiltViewModel
class ExampleViewModel @Inject constructor(
    val stateHandle: SavedStateHandle,
    val analyticsAdapter: AnalyticsAdapter
): ViewModel() { }
```

SavedStateHandle is a default binding available to all Hilt View Models (more on default bindings later on), while AnalyticsAdapter is a dependency which want to provide to the View Model. **This way of passing parameters to the view model is the preferred way over the Factory methods.**
But before we discuss this dependency scope let's look at how the activity or fragment retain an instance of that ViewModel.

The activity or fragments annotated with @AndroidEntryPoint can get the ViewModel instance as normal using ViewModelProvider or the by viewModels() KTX extension:

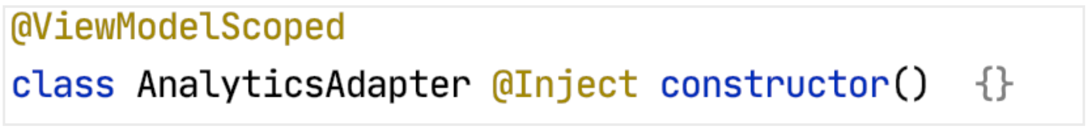


```
@AndroidEntryPoint
class ExampleActivity() : AppCompatActivity() {

    private val viewModel : ExampleViewModel by viewModels()
    private val analyticsAdapter = viewModel.analyticsAdapter
}
```

Only dependencies from the ViewModelComponent and its parent components

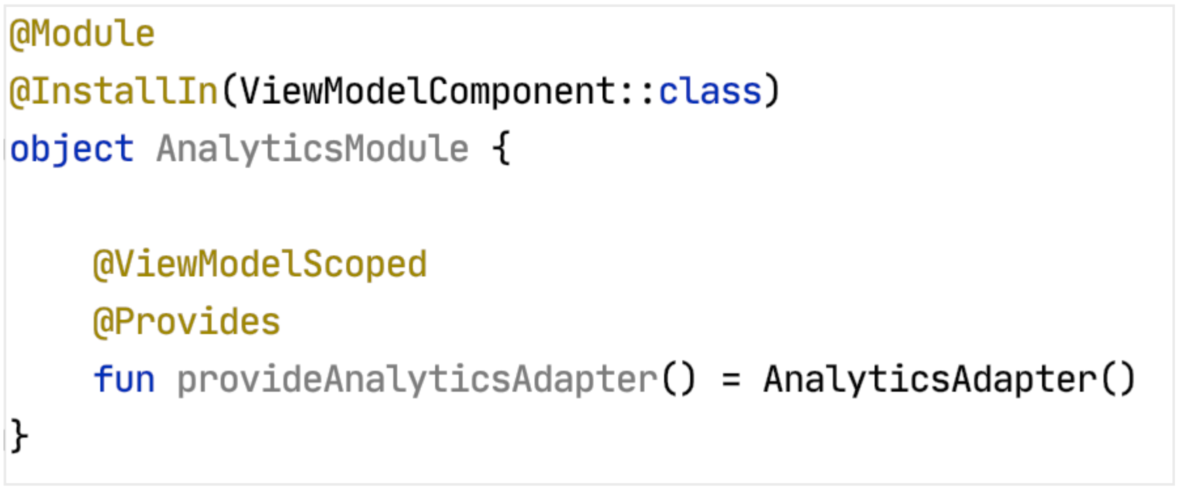
can be provided into the ViewModel.
All Hilt View Models are provided by the ViewModelComponent which follows the same lifecycle as a ViewModel, i.e. it survives configuration changes. To scope a dependency to a ViewModel use the @ViewModelScoped annotation.

If we own the class it will look like this:



```
@ViewModelScoped
class AnalyticsAdapter @Inject constructor()  {}
```

If it is from a library then it will probably look like this:



```
@Module
@InstallIn(ViewModelComponent::class)
object AnalyticsModule {

    @ViewModelScoped
    @Provides
    fun provideAnalyticsAdapter() = AnalyticsAdapter()
}
```

A @ViewModelScoped type will make it so that a single instance of the scoped type is provided across all dependencies injected into the Hilt View Model. **Other instances of a ViewModel that requests the scoped instance will receive a different instance.**

If a single instance needs to be shared across various View Models then it should be scoped using either @ActivityRetainedScoped or @Singleton.

For example, we can scope a dependency to be shared within a single ViewModel as such:

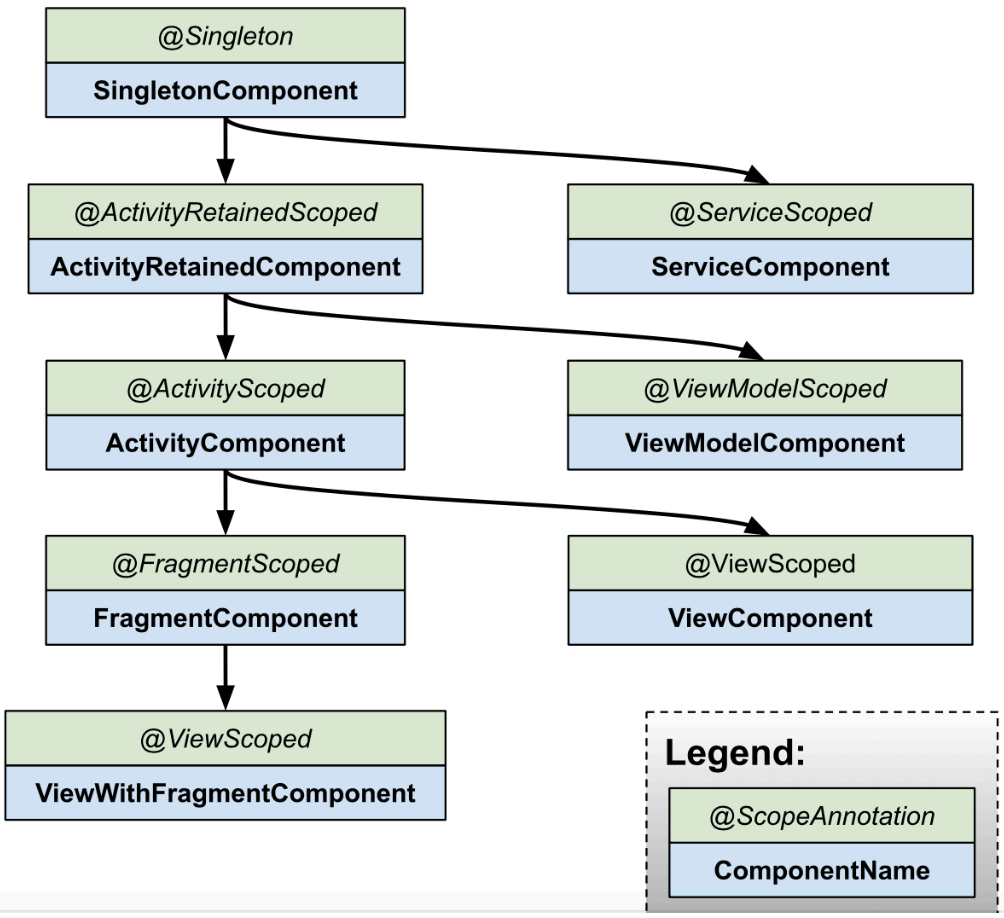
```
@Module
@InstallIn(ViewModelComponent::class)
internal object ViewModelMovieModule {
  @Provides
  @ViewModelScoped
  fun provideRepo(handle: SavedStateHandle) =
      MovieRepository(handle.getString("movie-id"));
}

class MovieDetailFetcher @Inject constructor(
  val movieRepo: MovieRepository
)

class MoviePosterFetcher @Inject constructor(
  val movieRepo: MovieRepository
)

@HiltViewModel
class MovieViewModel @Inject constructor(
  val detailFetcher: MovieDetailFetcher,
  val posterFetcher: MoviePosterFetcher
) : ViewModel {
  init {
    // Both detailFetcher and posterFetcher will contain the same instance of
    // the MovieRepository.
  }
}
```

Or another example:

```
@ViewModelScoped // Scopes type to the ViewModel
class UserInputAuthData(
  private val handle: SavedStateHandle // Default binding in ViewModelComponent
) { /* Cached data and logic here */ }

class RegistrationViewModel(
  private val userInputAuthData: UserInputAuthData,
  private val validateUsernameUseCase: ValidateUsernameUseCase,
  private val validatePasswordUseCase: ValidatePasswordUseCase
) : ViewModel() { /* ... */ }

class LoginViewModel(
  private val userInputAuthData: UserInputAuthData,
  private val validateUsernameUseCase: ValidateUsernameUseCase,
  private val validatePasswordUseCase: ValidatePasswordUseCase
) : ViewModel() { /* ... */ }

class ValidateUsernameUseCase(
  private val userInputAuthData: UserInputAuthData,
  private val repository: UserRepository
) { /* ... */ }

class ValidatePasswordUseCase(
  private val userInputAuthData: UserInputAuthData,
  private val repository: UserRepository
) { /* ... */ }
```

Since UserInputAuthData is scoped to the ViewModel, RegistrationViewModel and LoginViewModel will receive a *different instance* of UserInputAuthData. However, the UseCase dependencies of each ViewModel use the *same instance* that its ViewModel uses.

Installing a module into a component allows its bindings to be accessed as a dependency of other bindings in that component or in any child component below it in the component hierarchy:



Predefined qualifiers in Hilt

Hilt provides some predefined qualifiers. For example, as you might need the Context class from either the application or the activity, Hilt provides the @ApplicationContext and @ActivityContext qualifiers.

```
class AnalyticsServiceImpl @Inject constructor(
  @ApplicationContext context: Context
) : AnalyticsService { ... }

// The Application binding is available without qualifiers.
class AnalyticsServiceImpl @Inject constructor(
  application: Application
) : AnalyticsService { ... }
```

```
class AnalyticsAdapter @Inject constructor(
  @ActivityContext context: Context
) { ... }

// The Activity binding is available without qualifiers.
class AnalyticsAdapter @Inject constructor(
  activity: FragmentActivity
) { ... }
```

This is because Each Hilt component comes with a set of default bindings that Hilt can inject as dependencies into your own custom bindings.

Android component	Default bindings
`SingletonComponent`	`Application`
`ActivityRetainedComponent`	`Application`
`ViewModelComponent`	`SavedStateHandle`
`ActivityComponent`	`Application`, `Activity`
`FragmentComponent`	`Application`, `Activity`, `Fragment`
`ViewComponent`	`Application`, `Activity`, `View`
`ViewWithFragmentComponent`	`Application`, `Activity`, `Fragment`, `View`
`ServiceComponent`	`Application`, `Service`

Integration with the Jetpack navigation library

Add the following additional dependencies to your app Gradle file:

```
implementation("androidx.hilt:hilt-navigation-fragment:1.0.0")
```

If your ViewModel is scoped to the navigation graph, use the hiltNavGraphViewModels function that works with fragments that are annotated with @AndroidEntryPoint.

```
val viewModel: ExampleViewModel by hiltNavGraphViewModels(R.id.my_graph)
```

See a nice example https://stackoverflow.com/questions/66497047/hilt-doesnt-inject-a-scoped-viewmodel

Inject dependencies in classes not supported by Hilt

Hilt comes with support for the most common Android classes. However, you might need to perform field injection in classes that Hilt doesn't support.

In those cases, you can create an entry point using the @EntryPoint annotation. An entry point is the boundary between code that is managed by Hilt and code that is not. It is the point where code first enters into the graph of objects that Hilt manages. Entry points allow Hilt to use code that Hilt does not manage to provide dependencies within the dependency graph.

or example, Hilt doesn't directly support content providers. If you want a content provider to use Hilt to get some dependencies, you need to define an interface that is annotated with **@EntryPoint** for each binding type that you want and include qualifiers. Then add @InstallIn to specify the component in which to install the entry point as follows:

```
class ExampleContentProvider : ContentProvider() {

  @EntryPoint
  @InstallIn(SingletonComponent::class)
  interface ExampleContentProviderEntryPoint {
    fun analyticsService(): AnalyticsService
  }

  ...
}
```

To access an entry point, use the appropriate static method from EntryPointAccessors. The parameter should be either the component instance or the @AndroidEntryPoint object that acts as the component holder. Make sure that the component you pass as a parameter and the EntryPointAccessors static method both match the Android class in the @InstallIn annotation on the @EntryPoint interface:

```
class ExampleContentProvider: ContentProvider() {
    ...

  override fun query(...): Cursor {
    val appContext = context?.applicationContext ?: throw IllegalStateException()
    val hiltEntryPoint =
      EntryPointAccessors.fromApplication(appContext, ExampleContentProviderEntryPoint::class.java)

    val analyticsService = hiltEntryPoint.analyticsService()
    ...
  }
}
```

In this example, you must use the ApplicationContext to retrieve the entry point because the entry point is installed in SingletonComponent. If the binding that you wanted to retrieve were in the ActivityComponent, you would instead use the ActivityContext.

Unit tests
Hilt isn't necessary for unit tests, since when testing a class that uses constructor injection, you don't need to use Hilt to instantiate that class. Instead, you can directly call a class constructor by passing in fake or mock dependencies, just as you would if the constructor wasn't annotated:

```
@ActivityScoped
class AnalyticsAdapter @Inject constructor(
  private val service: AnalyticsService
) { ... }

class AnalyticsAdapterTest {

  @Test
  fun `Happy path`() {
    // You don't need Hilt to create an instance of AnalyticsAdapter.
    // You can pass a fake or mock AnalyticsService.
    val adapter = AnalyticsAdapter(fakeAnalyticsService)
    assertEquals(...)
  }
}
```

Hilt testing guide
One of the benefits of using dependency injection frameworks like Hilt is that it makes testing your code easier.
You can read more about testing with hilt here
https://developer.android.com/training/dependency-injection/hilt-testing

https://developer.android.com/training/dependency-injection
https://developer.android.com/training/dependency-injection/hilt-android

https://medium.com/androiddevelopers/using-hilts-viewmodelcomponent-53b46515c4f4
https://medium.com/androiddevelopers/scoping-in-android-and-hilt-c2e5222317c0
https://dagger.dev/hilt/view-model.html