|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | dependency injection | <chare> | 1 | <pext> | Dependency injection is a technique  used to remove hard-coded dependencies between objects,  making it easier to change them later.  This is useful in situations where you want to be able to swap out  one implementation of a dependency for another,  without having to change the code that uses it. | </end> |
| <hitle> | dependency injection pros | <chare> | 2 | <pext> | 1. Reusability of classes and decoupling of dependencies:  It's easier to swap out implementations of a dependency.  Code reuse is improved because of inversion of control,  and classes no longer control how their dependencies are created,  but instead work with any configuration.  2. Ease of refactoring:  The dependencies become a verifiable part of the API surface,  so they can be checked at object-creation time or at compile time  rather than being hidden as implementation details.  3. Ease of testing: A class doesn't manage its dependencies,  so when you're testing it, you can pass in different implementations  to test all of your different cases. | </end> |
| <hitle> | Types of dependency Injection | <chare> | 2 | <pext> | 1. constructor injection  2. field injection | </end> |
| <hitle> | Dagger | <chare> | 2 | <pext> | Dagger 2 is a compile time dependency inject framework in Android  that is widely popular for reducing dependency in android development.  Basically Dagger 2 was introduced by Square  and now maintained by google.  Dagger 2 is annotation based library so we can learn basic annotations  and also will apply on the app development.  -----------@Module--------  1. @Module is annotation used to construct the objects  and provide the dependencies of the object  for example ApiClient object for retrofit is an object as module  and returning the instance of retrofit builder, HttpLogginInterceptor  for logging and OkHttpClient for adding  cache/interceptor/ timeout’s management.  The methods of @Module annotated class “provide dependencies”.  -----------@Provides--------  2. @Provides is used over the each method of Module class  that will return object of each method/feature.  It is said “tell Dagger how we want to build and present a dependency“  -----------@Inject----------  3. @Inject is used over the field, constructor and method  to get the instance of required class.  It is said “dependency is requested”.  -----------@Component--------  4. @Component is an interface which acts as bridge  between @Module and @Inject also it return the builder  for DaggerApp of Application.  It contains all the module of all associated with component.  It’s a bridge between @Inject and @Module.  -----------@Singleton--------  5. @Singleton is used to indicate only single instance  of particular dependency is being created.  -----------@Binds --------  6. @Binds is used when module is an abstract class.  Purpose of Binds is very similar with Provides but Binds is more concise.  -----------@ Qualifier--------  7. @Qualifier is used if different objects of the same type are necessary.  -----------@Scope--------------  8. @Scope enables to create global and local singletons  Note: It is recommended to use Dagger-Hilt.  It is easier to implement and saves you a lot of headaches. | </end> |
| <hitle> | Hilt | <chare> | 2 | <pext> | Hilt is a dependency injection library for Android that reduces  the boilerplate of doing manual dependency injection in the project.  Doing manual dependency injection requires to construct  every class and its dependencies by hand, and to use containers  to reuse and manage dependencies.  Hilt provides a standard way to use DI in your application  by providing containers for every Android class in your project  and managing their lifecycles automatically.  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.  This guide explains the basic concepts of Hilt and its generated containers. It also includes a demonstration of how to bootstrap an existing app to use Hilt. | </end> |
| <hitle> | Hilt application class | <chare> | 3 | <pext> | All apps that use Hilt must contain an Application class  that is annotated with @HiltAndroidApp.  @HiltAndroidApp triggers Hilt's code generation,  including a base class for the application that serves  as the application-level dependency container.  @HiltAndroidApp  class ExampleApplication : Application() { ... }  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. | </end> |
| <hitle> | Inject dependencies Android classes | <chare> | 3 | <pext> | Once Hilt is set up in the Application class  and an application-level component is available,  Hilt can provide dependencies to other Android classes  that have the @AndroidEntryPoint annotation.  @AndroidEntryPoint  class ExampleActivity : AppCompatActivity() { ... }  Hilt currently supports some of Android classes:  1. Application (by using @HiltAndroidApp)  2. ViewModel (by using @HiltViewModel)  3. Activity  4. Fragment  5. View  6. Service  7. BroadcastReceiver  If we annotate an Android class with @AndroidEntryPoint,  then we also must annotate Android classes that depend on it.  For example, if we annotate a fragment, then we must also annotate  any activities where we use that fragment.  Hilt support for Android classes:  1. Hilt only supports activities that extend ComponentActivity,  such as AppCompatActivity.  2. Hilt only supports fragments that extend androidx.Fragment.  3. Hilt does not support retained fragments.  @AndroidEntryPoint generates an individual Hilt component  for each Android class in your project.  These components can receive dependencies  from their respective parent classes  as described in Component hierarchy.  To obtain dependencies from a component,  we can use the @Inject annotation to perform field injection.  @AndroidEntryPoint  class ExampleActivity : AppCompatActivity() {  @Inject lateinit var analytics: AnalyticsAdapter  ...  }  Note: Fields injected by Hilt cannot be private.  Attempting to inject a private field with Hilt results  in a compilation error.  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. | </end> |
| <hitle> | Hilt bindings | <chare> | 2 | <pext> | To perform field injection, Hilt needs to know how to provide instances  of the necessary dependencies from the corresponding component.  A 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.  We can use the @Inject annotation on the constructor of a class  to tell Hilt how to provide instances of that class.  The parameters of an annotated constructor of a class  are the dependencies of that class.  class AnalyticsAdapter @Inject constructor(  private val service: AnalyticsService  ) { ... }  (\*AnalyticsAdapter has AnalyticsService as a dependency. Therefore, Hilt must also know how to provide instances of AnalyticsService.\*)  Note: At build time, Hilt generates Dagger components  for Android classes.  Then, Dagger walks through the code and performs two steps.  1. It builds and validates dependency graphs, ensuring that  there are no unsatisfied dependencies and no dependency cycles.  2. It generates the classes that it uses at runtime to create  the actual objects and their dependencies. | </end> |
| <hitle> | Hilt modules | <chare> | 2 | <pext> | Sometimes a type cannot be constructor-injected.  This can happen for multiple reasons.  For example, we cannot constructor-inject an interface.  We also cannot constructor-inject a type that we do not own,  such as a class from an external library.  In these cases, we can provide Hilt with binding information  by using Hilt modules.  A Hilt module is a class that is annotated with @Module.  Like a Dagger module, it informs Hilt  how to provide instances of certain types.  Unlike Dagger modules, we must annotate Hilt modules  with @InstallIn to tell Hilt which Android class each module  will be used or installed in.  Note: Hilt modules are different from Gradle modules.  Dependencies that we provide in Hilt modules are available  in all generated components that are associated with the Android class  where we install the Hilt module.  Note: Because Hilt's code generation needs access to  all of the Gradle modules that use Hilt,  the Gradle module that compiles the Application class  also needs to have all of your Hilt modules  and constructor-injected classes in its transitive dependencies. | </end> |
| <hitle> | Inject interface instances with @Binds | <chare> | 3 | <pext> | Let’s say the AnalyticsService example.  If AnalyticsService is an interface, then we cannot constructor-inject it.  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.  @Binds annotated function provides two informations to Hilt:  1. The function return type tells Hilt  what interface the function provides instances of.  2. The function parameter tells Hilt which implementation to provide.  interface AnalyticsService {  fun analyticsMethods()  }  // Constructor-injected, because Hilt needs to know how to  // provide instances of AnalyticsServiceImpl, too.  class AnalyticsServiceImpl @Inject constructor(  ...  ) : AnalyticsService { ... }  @Module  @InstallIn(ActivityComponent::class)  abstract class AnalyticsModule {  @Binds  abstract fun bindAnalyticsService(  analyticsServiceImpl: AnalyticsServiceImpl  ): AnalyticsService  }  (\*The Hilt module AnalyticsModule is annotated with @InstallIn(ActivityComponent::class) because you want Hilt to inject that dependency into ExampleActivity. This annotation means that all of the dependencies in AnalyticsModule are available in all of the app's activities.\*) | </end> |
| <hitle> | Inject instances with @Provides | <chare> | 3 | <pext> | Interfaces are not the only case  where we cannot constructor-inject a type.  Constructor injection is also not possible if we don't own the class  because it comes from an external library like Retrofit, OkHttpClient,  or Room databases, or if instances must be created  with the builder pattern.  We can tell Hilt how to provide instances of the type  by creating a function inside a Hilt module and annotating that function  with @Provides.  @Provides annotated function supplies three information to Hilt:  1. The function return type tells Hilt  what type the function provides instances of.  2. The function parameters tell Hilt  the dependencies of the corresponding type.  3. 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.  @Module  @InstallIn(ActivityComponent::class)  object AnalyticsModule {  @Provides  fun provideAnalyticsService(  // Potential dependencies of this type  ): AnalyticsService {  return Retrofit.Builder()  .baseUrl("https://example.com")  .build()  .create(AnalyticsService::class.java)  }  } | </end> |
| <hitle> | multiple bindings for the same type | <chare> | 3 | <pext> | In cases where we need Hilt to provide different implementations  of the same type as dependencies,  we must provide Hilt with multiple bindings.  We can define multiple bindings for the same type with qualifiers.  A qualifier is an annotation that we use to identify a specific binding  for a type when that type has multiple bindings defined.  We can define the qualifiers that we will use to annotate  the @Binds or @Provides methods.  @Qualifier  @Retention(AnnotationRetention.BINARY)  annotation class AuthInterceptorOkHttpClient  @Qualifier  @Retention(AnnotationRetention.BINARY)  annotation class OtherInterceptorOkHttpClient  (\*If you need to intercept calls to AnalyticsService, we could use an OkHttpClient object with an interceptor. For other services, we might need to intercept calls in a different way. In that case, we need to tell Hilt how to provide two different implementations of OkHttpClient.\*)  Then, Hilt needs to know how to provide an instance  of the type that corresponds with each qualifier.  In this case, we could use a Hilt module with @Provides.  Both methods have the same return type,  but the qualifiers label them as two different bindings:  @Module  @InstallIn(SingletonComponent::class)  object NetworkModule {  @AuthInterceptorOkHttpClient  @Provides  fun provideAuthInterceptorOkHttpClient(  authInterceptor: AuthInterceptor  ): OkHttpClient {  return OkHttpClient.Builder()  .addInterceptor(authInterceptor)  .build()  }  @OtherInterceptorOkHttpClient  @Provides  fun provideOtherInterceptorOkHttpClient(  otherInterceptor: OtherInterceptor  ): OkHttpClient {  return OkHttpClient.Builder()  .addInterceptor(otherInterceptor)  .build()  }  }  We can inject the specific type that we need by annotating the field  or parameter with the corresponding qualifier.  // As a dependency of another class.  @Module  @InstallIn(ActivityComponent::class)  object AnalyticsModule {  @Provides  fun provideAnalyticsService(  @AuthInterceptorOkHttpClient okHttpClient: OkHttpClient  ): AnalyticsService {  return Retrofit.Builder()  .baseUrl("https://example.com")  .client(okHttpClient)  .build()  .create(AnalyticsService::class.java)  }  }  // As a dependency of a constructor-injected class.  class ExampleServiceImpl @Inject constructor(  @AuthInterceptorOkHttpClient private val okHttpClient: OkHttpClient  ) : ...  // At field injection.  @AndroidEntryPoint  class ExampleActivity: AppCompatActivity() {  @AuthInterceptorOkHttpClient  @Inject lateinit var okHttpClient: OkHttpClient  }  As a best practice, if we add a qualifier to a type,  add qualifiers to all the possible ways to provide that dependency.  Leaving the base or common implementation without a qualifier  is error-prone and could result in Hilt injecting the wrong dependency. | </end> |
| <hitle> | Predefined qualifiers in Hilt | <chare> | 3 | <pext> | Hilt provides some predefined qualifiers.  For example, as we might need the Context class  from either the application or the activity, Hilt provides  the @ApplicationContext and @ActivityContext qualifiers.  class AnalyticsAdapter @Inject constructor(  @ActivityContext private val context: Context,  private val service: AnalyticsService  ) { ... }  (\*The AnalyticsAdapter class needs the context of the activity.  It demonstrates how to provide the activity context to AnalyticsAdapter.\*) | </end> |
| <hitle> | Generated Hilt components for Android classes | <chare> | 2 | <pext> | For each Android class in which we can perform field injection,  there's an associated Hilt component that we can refer to  in the @InstallIn annotation.  Each Hilt component is responsible for injecting its bindings  into the corresponding Android class.  Generated components for Android classes are:  1. SingletonComponent for Application  2. ActivityRetainedComponent  3. ViewModelComponent forViewModel  4. ActivityComponent forActivity  5. FragmentComponent for Fragment  6. ViewComponent for View  7. ViewWithFragmentComponent for View annotated with @WithFragmentBindings  8. ServiceComponent for Service  Note: Hilt doesn't generate a component for broadcast receivers  because Hilt injects broadcast receivers directly  from SingletonComponent. | </end> |
| <hitle> | Hilt Component lifetimes | <chare> | 2 | <pext> | Hilt automatically creates and destroys instances  of generated component classes.  1. SingletonComponent  created at Application#onCreate(), destroyed at Application destroyed  2. ActivityRetainedComponent  Activity#onCreate() --- Activity#onDestroy()  3. ViewModelComponent  ViewModel created --- ViewModel destroyed  4. ActivityComponent  Activity#onCreate() --- Activity#onDestroy()  5. FragmentComponent  Fragment#onAttach() --- Fragment#onDestroy()  6. ViewComponent  View#super() --- View destroyed  7. ViewWithFragmentComponent  View#super() --- View destroyed  8. ServiceComponent  Service#onCreate() --- Service#onDestroy()  Note: ActivityRetainedComponent lives across configuration changes,  so it is created at the first Activity#onCreate()  and destroyed at the last Activity#onDestroy(). | </end> |
| <hitle> | Hilt vs Dagger | <chare> | 2 | <pext> | Hilt is built on top of the Dagger dependency injection library,  providing a standard way to incorporate Dagger  into an Android application.  The goals of Hilt are:  1. To simplify Dagger-related infrastructure for Android apps.  2. To create a standard set of components and scopes to ease setup,  readability, and code sharing between apps.  3. To provide an easy way to provision different bindings  to various build types, such as testing, debug, or release.  Because the Android operating system  instantiates many of its own framework classes,  using Dagger in an Android app requires to write  a substantial amount of boilerplate.  Hilt reduces the boilerplate code that is involved in using Dagger  in an Android application.  Hilt automatically generates and provides four things:  1. Components for integrating Android framework classes with Dagger  that you would otherwise need to create by hand.  2. Scope annotations to use with the components  that Hilt generates automatically.  3. Predefined bindings to represent Android classes  such as Application or Activity.  4. Predefined qualifiers to represent  @ApplicationContext and @ActivityContext.  Dagger and Hilt code can coexist in the same codebase.  However, in most cases it is best to use Hilt to manage  all of your usage of Dagger on Android. | </end> |
| <hitle> | Dagger | <chare> | 2 | <pext> | Dagger is a Dependency Injection framework  that will generate a lot of boilerplate code  for you to achieve the goal of Dependency injection in Android development. | </end> |
| <hitle> | Dagger pros | <chare> | 2 | <pext> | 1. Generating the AppContainer code that we manually implemented  in the manual DI section.  2. Creating factories for the classes available in the application graph.  This is how dependencies are satisfied internally.  3. Deciding whether to reuse a dependency or create a new instance  through the use of scopes.  4. Creating containers for specific flows using Dagger subcomponents.  This improves the app's performance by releasing objects in memory  when they're no longer needed. | </end> |
| <hitle> | Dependency Injection Dagger Hilt Scope Scoping Component tree | <chare> | 2 | <pext> | The root is SingletonComponent which is activity scoped.  1. SingletonComponent, @Singletone, @ActivityScoped  1-1. ActivityRetainedComponent, @ActivityRetainedScoped  1-1-1. ActivityComponent, @ActivityScoped  1-1-1-1. FragmentComponent, @FragmentScoped  1-1-1-2. ViewComponent, @ViewScoped  1-1-2. ViewModelComponent, @ViewModelScoped  1-2. ServiceComponent, @ServiceScoped  //For Activity + ViewModel  @Module  @InstallIn(ActivityRetainedComponent::class)  object ActivityLevelModule {  }  //For Activity  @Module  @InstallIn(ActivityComponent::class)  object OnlyActivityLevelModule {  }  //For ViewModel  @Module  @InstallIn(ViewModelComponent::class)  object OnlyViewModelLevelModule {  }  //For Fragment  @Module  @InstallIn(FragmentComponent::class)  object FragmentLevelModule{  }  //For Service  @Module  @InstallIn(ServiceComponent::class)  object ServiceLevelModule {  } | </end> |