**Section 84 – Mastering Dependency Injection with Hilt**

**A. Key Concepts Taught**

**1. What is Dependency Injection (DI)**

* **Definition:** A design pattern where dependencies are provided from an external source instead of a class creating them internally.
* **Goal:** Separate *object creation* from *object usage* → leads to cleaner, more modular, and testable code.
* **Benefits:**
  + **Loose coupling** – classes are less dependent on specific implementations.
  + **Improved testability** – easier to mock or swap dependencies.
  + **Scalability** – easier to manage large projects with many dependencies.
  + **Readability** – code is cleaner and easier to follow.

**2. Manual Dependency Injection**

**Without DI:**

* Class directly creates its dependency → tightly coupled.
* Hard to replace dependencies for testing.

class Car {

private val engine = Engine() // tightly coupled

fun drive() = println(engine.start())

}

**With Manual Constructor Injection:**

class Car(private val engine: Engine) {

fun drive() = println(engine.start())

}

// Main

val engine = Engine()

val car = Car(engine) // injecting dependency externally

car.drive()

**3. Hilt – Dependency Injection Library**

* Built on **Dagger 2**, but designed specifically for Android.
* Reduces boilerplate and integrates with Android lifecycle.
* **Key concepts:**
  + **Component** – container that knows how to provide dependencies.
  + **Module** – defines how dependencies are provided.
  + **Scopes** – define the lifecycle of dependencies.

**4. Adding Hilt to Project**

1. **Add Plugins** in *project-level* build.gradle:

plugins {

id 'com.google.dagger.hilt.android' version 'x.x.x' apply false

id 'org.jetbrains.kotlin.kapt' version 'x.x.x' apply false

}

1. **App-level build.gradle:**

plugins {

id 'com.google.dagger.hilt.android'

id 'org.jetbrains.kotlin.kapt'

}

dependencies {

implementation "com.google.dagger:hilt-android:2.x"

kapt "com.google.dagger:hilt-compiler:2.x"

}

1. **Sync Project.**

**5. Constructor Injection with Hilt**

* Add @Inject to constructor → tells Hilt how to create it.

class Engine @Inject constructor() {

fun start() = "Engine started"

}

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

fun drive() = println(engine.start())

}

* Hilt automatically provides Engine to Car when needed.

**6. Hilt Modules**

* Define how to provide dependencies, especially:
  + When **constructor injection** is not possible (e.g., third-party classes, interfaces).

@Module

@InstallIn(SingletonComponent::class)

object AppModule {

@Provides

@Singleton

fun provideEngine(): Engine = Engine()

@Provides

@Singleton

fun provideCar(engine: Engine): Car = Car(engine)

}

* **@Singleton** ensures only one instance exists for entire app lifecycle.

**7. Hilt Components**

* **Bridge** between modules (providers) and classes that need dependencies.
* Example (pure Kotlin project, not Android):

@Singleton

@Component(modules = [AppModule::class])

interface AppComponent {

fun getCar(): Car

}

**8. Generated Code**

* Dagger/Hilt generates code at compile-time (e.g., DaggerAppComponent).
* Cleaning project removes generated code; rebuilding regenerates it.

**9. Field Injection**

* Inject directly into class properties (common in Activities/Fragments).
* Used when you **cannot control the constructor** (e.g., Android framework classes).

class Car {

@Inject lateinit var engine: Engine

fun drive() = println(engine.start())

}

@Component(modules = [AppModule::class])

interface AppComponent {

fun inject(car: Car) // injects into fields

}

**10. Method Injection**

* Inject dependencies into a method (rare, used when dependency is temporary).

class Car {

private lateinit var engine: Engine

@Inject

fun installEngine(engine: Engine) {

this.engine = engine

}

fun drive() = println(engine.start())

}

**11. Important Hilt/Dagger Annotations**

**Application Level**

* @HiltAndroidApp – marks Application class for Hilt setup.

**Injection**

* @Inject – marks constructor, field, or method for injection.
* @Named / @Qualifier – distinguish between multiple same-type dependencies.

**Modules**

* @Module – marks a class as a provider of dependencies.
* @Provides – function returns a dependency.
* @Binds – bind interface to implementation (abstract method).

**Components**

* @InstallIn(SingletonComponent::class) – defines scope of module.

**Scopes**

* @Singleton – single instance for entire app lifecycle.
* @ActivityScoped, @FragmentScoped – lifecycle-specific.

**Testing**

* @HiltAndroidTest – enables Hilt in tests.
* @UninstallModules – remove specific modules during tests.

**B. Clear Steps for Future Project Implementation**

**1. Setup Hilt in Gradle**

* **Project-level build.gradle:**

plugins {

id 'com.google.dagger.hilt.android' version '2.x' apply false

id 'org.jetbrains.kotlin.kapt' version '1.x' apply false

}

* **App-level build.gradle:**

plugins {

id 'com.google.dagger.hilt.android'

id 'org.jetbrains.kotlin.kapt'

}

dependencies {

implementation "com.google.dagger:hilt-android:2.x"

kapt "com.google.dagger:hilt-compiler:2.x"

}

* **Why:** This sets up Hilt so it can generate dependency injection code at compile time.

**2. Annotate the Application Class**

@HiltAndroidApp

class MyApp : Application()

* **Why:** This is the entry point for Hilt. Without it, injection into Android components won’t work.

**3. Decide the Injection Type**

* **Constructor Injection** → Preferred for most classes (immutable, easy to test).
* **Field Injection** → For Android components (Activities, Fragments) that you don’t instantiate yourself.
* **Method Injection** → For dependencies only needed temporarily inside a method.

**4. Add @Inject in the Right Place**

* **Constructor example:**

class Engine @Inject constructor() { ... }

* **Field example:**

@Inject lateinit var engine: Engine

* **Method example:**

@Inject fun installEngine(engine: Engine) { this.engine = engine }

**5. Create Modules for Special Cases**

* **When?**
  + Third-party library classes
  + Interfaces
  + Need extra creation logic
* **Example:**

@Module

@InstallIn(SingletonComponent::class)

object AppModule {

@Provides

fun provideRetrofit(): Retrofit { ... }

}

**6. Scope Dependencies Properly**

* @Singleton → Live for the entire app lifecycle.
* @ActivityScoped → Live for the lifecycle of a single Activity.
* **Tip:** Don’t mark everything @Singleton — only what’s truly global.

**7. Use Entry Point Annotations in Android Components**

@AndroidEntryPoint

class MainActivity : AppCompatActivity() { ... }

* **Why:** Tells Hilt this component should receive injected dependencies.

**8. Testing with Hilt**

* Use @HiltAndroidTest for test classes.
* Use @UninstallModules to replace real modules with test versions that return fake/mocked data.

**Part B – Additional Important Knowledge**

**1. Choosing Injection Type**

| **Feature** | **Constructor Injection ✅ (Preferred)** | **Field Injection ⚠** | **Method Injection ⚠** |
| --- | --- | --- | --- |
| Immutability | Strong | Weak | Weak |
| Best For | Most classes | Activities/Fragments | Temporary needs |
| Testability | Easiest | Requires setup | Requires setup |
| Visibility of deps | Clear in constructor | Hidden in fields | Hidden in methods |

**2. Why Constructor Injection is Preferred**

* All dependencies are available when the object is created.
* Promotes immutability (val properties).
* Makes dependencies explicit (easier for new developers to read).
* Works perfectly with unit testing and mocking.

**3. When to Use Modules**

* Class is from a library and cannot be annotated with @Inject.
* You need dynamic logic (e.g., deciding which implementation to return).
* You’re binding an interface to its implementation:

@Binds

abstract fun bindRepo(repoImpl: RepoImpl): Repo

**4. Best Practices**

* Keep modules small — one responsibility per module.
* Avoid over-scoping (don’t make everything @Singleton).
* Minimize field injection; only use it where constructor injection isn’t possible.
* Prefer @Binds for interfaces — it’s faster and cleaner than @Provides.

**5. Latest Industry Approach**

* **Architecture:** MVVM + Repository pattern + Coroutines/Flow.
* **Hilt Usage Pattern:**
  + Inject repositories into @HiltViewModel classes.
  + Inject API services/DAOs into repositories.
  + Scope dependencies appropriately.
* **Example ViewModel Injection:**

@HiltViewModel

class MyViewModel @Inject constructor(

private val repo: MyRepository

) : ViewModel()