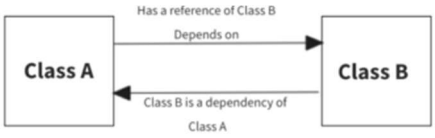
10 – Dependency injection

Tecnica generale che permette ad una classe di avere una **dipendenza con un'altra classe**

- Causa molti problemi perché, se non ci si focalizza bene sulle dipendeze relative, si rischia di scrivere codice molto complesso, difficile da mantenere.



In generale, la regola base è: "quando una classe necessita di lavorare con una dipendenza, è necessario che quella dipendenza descrive la classe come un'interfaccia" → questo rende il codice più disaccoppiabile:

- Permette di creare test più facilmente
- Comodo che il model abbia un'interfaccia per il viewmodel

Come fornire dipendenze:

```
class MyRepository(private val context: Context) {          1
    val myModel = MyDatabaseImpl.getDb(context)             2
    val myApi = MyApiImpl("https://myhost.com/api/v1")      3
    //... more code
}
```

2 patterns:

- Se hai una dipendenza forniscila come parametro del tuo **constructor** → così chiunque vuole buildarti, deve fornirtela.
    - Repository deve avere context che è necessario per implementare db/API
- Storare dipendenze in una variabile **lateint** che deve essere inizializzata da qualcuno

Per ridurre class coupling:

```
interface MyDatabase { ... }

@Database
class MyDatabaseImpl(context: Context):
            RoomDatabase(context), MyDatabase { ... }
```

Nella maggior parte dei casi, Room fornisce la parte implementativa.

```
interface MyApi{ ... }

class MyApiImpl(baseURL: string): MyApi { ... }
```

MyRepository dipende su myDb e MyApi

```
class MyRepository(val myDb: MyDatabase, val myApi: MyApi) {

    //... more code
}
```

C'è un pezzo di codice che fa in modo che i parametri siano riempiti nel modo corretto. L'approccio di android è quello di **inserire** qualcosa **a compile time**

- Pre-compilation : trova dipendenze
- Noi capiamo cosa manca e forniamo la parte necessaria
    - In caso di debugging, possiamo vedere invocazioni che arrivano da sorgenti inaspettate
        - Arrivano da classi scritte in modo automatico dal dependency injection a compile time

Nota: Dependency injection si occupa di scoprire quale classe concreta dovrebbe essere fornita come parametro di costruzione a una classe client e come individuare o costruire istanze reali di quelle classi

Storia in android:

Several DI frameworks exist for the Java/Kotlin ecosystem, some specifically aimed at Android and other at a more general programming environment

- **Spring** — based on a central piece of software, the ApplicationContext, that is responsible for instantiating, configuring, and assembling objects at runtime, as well as managing their life cycles
- **Dagger** — operates at compile time and produces static code where dependencies are resolved and wired
- **Dagger2** — Build on top of Dagger, created by Google Team
- **Koin** — A runtime DI framework, completely written in Kotlin
- **Hilt** — build on top of Dagger2, completely focused on Android
    - Provando a nascondere le complessità di dagger
- Frameworks often use annotations to label classes and their dependencies
    - Both to express the role played by a class and to mark the place where the dependency should be injected

HILT

Hilt è una libreria DI che **genera codice in fase di compilazione** basandosi su una serie di annotazioni

- Richiede diverse dipendenze, sia nel file build.gradle del progetto che nel modulo dell'app.

project: build.gradle

```
buildscript { ...
  dependencies { ...
    classpath "com.google.dagger:hilt-android-gradle-plugin:2.44"
  }
}
plugins { ...
  id 'com.google.dagger.hilt.android' version '2.44' apply false
}
```

import GRADLE

app: build.gradle

```
plugins { ...
  id 'kotlin-kapt'
  id 'com.google.dagger.hilt.android'
}
dependencies { ...
  implementation 'com.google.dagger:hilt-android:2.44'
  kapt 'com.google.dagger:hilt-compiler:2.44'
}
// Allow references to generated code
kapt {
  correctErrorTypes true
}
```

Hilt capisce un insieme di annotazioni che indicano cosa è injectable e dove deve essere injected.

- si basa sul fatto che un'applicazione è creata da un singleton.

Dichiariamo la nostra application class con **@HiltAndroidApp**:

- Attiva la generazione del codice Hilt e include la classe base per l'applicazione che serve come contenitore di dipendenze a livello di applicazione
- Memorizzerà un riferimento a tutte le dipendenze singleton

```
@HiltAndroidApp
class MyHiltApp : Application()

@AndroidEntryPoint
class MainActivity : AppCompatActivity {
  @Inject lateinit var player: MusicPlayer
  //...
}
```

Il singolo component/activity che necessita di essere injected con contenuto, necessita una speciale annotaziione: **@AndroidEntryPoint.**

```
class MusicPlayer @Inject constructor() {

  fun play(id: string) {
    //...
  }
}
```

Inject per due obiettivi:

- Questa **lateinit var** necessità di essere popolata da injection
- Questa **classe** deve essere injected (musicPlayer inject constuctor)
    - Quando questa classe viene creata, deve essere usata per popolare il corrispondente lateinit

Questo va bene se si ha una dipendenza pura: MainActivity dipende da musicPlayer che è una classe concreta e MusicPlayer inject proprietà.

nota: annotazione sevre anche per marcare delle classi come injectable, inquesto caso MusicPlayer ha il primary constructor marcato come injected → quella classe è candidata per essere injected.

Nel caso in cui si usi l'approccio classico in cui si separa interfaccia e implementazione (molte attività si basano su musicPlayer che è un'interfaccia –non una classe--), non si può usare questo metodo *(impl implementa musicPlayer, non sono uguali).*

Dependency injection possono accadere in due modi:

- **Constructor injection:** dipendenza fornita al costruttore come parametero
- **Property injection**: quando una proprietà lateinit è dichiarata e non inizializzata

To inject a dependency, the destination must be labelled with `@Inject`

- In case of constructor injection, it is the constructor that must be annotated, independently of how many parameters it declares

Hilt selects what value to store in these variables using different criteria

- If the variable has a **concrete type**, and the type has a **no-arg constructor**, it is possible to decorate the primary constructor of the dependency with the `@Inject` annotation (again!)
- Otherwise, a `@Module` class need to be defined

Quando creiamo un modulo, abbiamo la possibilità di specificare cosa il component deve fornire

- Possiamo avere altri component a cui vogliamo ridurre il lyfecycle a quello di un'attività (quando c'è activity → vogliamo che qualcosa sia injected)
    - Sometimes, an app need that the same dependency is used in all places where an object of that kind is required
        - This can easily be achieved adding an extra annotation to the dependency class: `@Singleton`
    - It is also possible to constrain creating different instances of a dependency per each high-level Android component (e.g, activity or service)
        - Via `@ActivityRetainedScoped`, `@ViewModelScoped`, `@ActivityScoped`, `@ServiceScoped`
    - Or to limit the scope of a dependency to each view or fragment instances
        - Via `@ViewScoped`, `@FragmentScoped`
    - If no scope is indicated, a new instance of the dependency will be created per each requesting client

- **Lifecycle dei componenti:**

- `@Singleton` - Application::onCreate() → Application::onTerminate() → vive con applicazione
- `@ActivityRetainedScoped` - Activity::onCreate(...) → Activity::onDestroy() *(objects survive the destructions due to configuration changes)*
- `@ViewModelScoped` - ViewModel::<init> → ViewModel::onCleared()
- `@ActivityScoped` - Activity::onCreate(...) → Activity::onDestroy()
- `@FragmentScoped` - Fragment::onAttach(...) → Fragment::onDetach()
- `@ViewScoped` - View construction → View destruction
- `@ServiceScoped` - Service::onCreate() → Service::onDestroy()

MODULO HILT

Per creare un modulo hilt, bisogna annotare una classe come **@Module**

- Metodi annotati con **@Provides** sono usati dal framewor per fornire istanze di un dato tipo
- I parametri della funzione vengono utilizzati per fornire le dipendenze transitive necessarie all'istanza da fornire

```
class MusicPlayer @Inject constructor(
  private val db: MusicDatabase//this is an interface! It prevents direct injection
) {
  fun play(id: string) {  … }
}
```

```
@Module
@InstallIn(SingletonComponent::class)
class MusicModule {
  @Singleton // any class needing this dependency will get the same instance
  @Provides
  fun provideMusicDB(@ApplicationContext context: Context): MusicDatabase {
    return Room.databaseBuilder(context, MusicDatabase::class.java, "music.db")
                            .build()
  }
}
```

Module mi serve per iniettare in MusicPlayer, come parte dell'applicazione (quindi @Singleton):

- Al suo interno dichiaro dei metodi
- @Provides: deve fornire qualcosa
- Dobbiamo fornire un contesto (insieme di informazioni , folder, etc…) → forniamo come parametro.

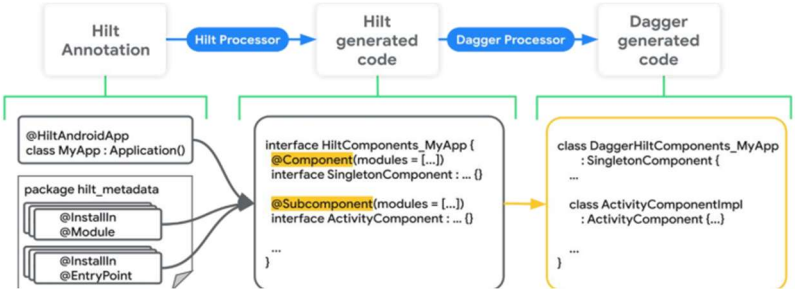
JETPACK INTEGRATION:

Composable function non possono far parte di injection → perché non sono object. L'unico punto in cui possiamo inserire cose è il viewModel (se lo annotiamo come **@HiltViewModel**)

```
@HiltViewModel
class MyViewModel @Inject constructor(
    private val adapter: AnalyticsAdapter,
    private val state: SavedStateHandle
): ViewModel() {
   //...
}
```

Generazione codice, sotto coperta:

- Tutto succede a compile time
  - Creiamo application
  - Creiamo modules
    - Definiamo vari metodi per fornire cosa necessario
  - Hilt processor Riscrive hilt annotation in classi / interfacce
  - Dagger Processor genera codice per classi
- Hilt ha riscritto il viewModel in un altro viewModel
  - Classi diverse
  - Righe completamente diverse (se c'è un errore a riga 150, probabilmente non è veramente 150)



Conviene usare Hilt solo se è tutto ok e si sta consegnando, non in fase di debugging