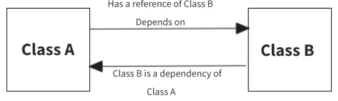
# 10 – Dependency injection

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

### Immagine che contiene testo, schermata, Carattere, numero Descrizione generata automaticamente**Come fornire dipendenze:**

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:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

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

MyRepository dipende su myDb e MyApi

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.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

**Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteimport GRADLE**

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.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteDichiariamo 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

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

**Inject per due obiettivi**:

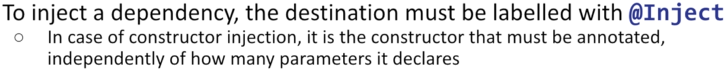
* Questa **lateinit var** necessità di essere popolata da injection
* Questa **classe** deve essere injected (musicPlayer inject constuctor)
  + Quando questa classe viene creata, deve essere usata per popolare il corrispondente lateinit

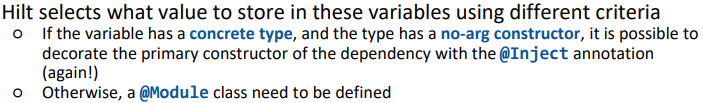
Questo va bene se si ha una dipendenza pura: MainActivity dipende da musicPlayer che è una classe concreta e MusicPlayer inject proprietà.  
**nota**: annotazione sevre anche per marcare delle classi come injectable, inquesto caso MusicPlayer ha il primary constructor marcato come injected → quella classe 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:

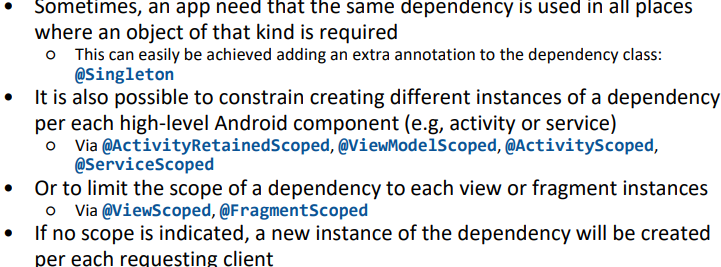
* **Constructor injection:** dipendenza fornita al costruttore come parametero
* **Property injection**: quando una proprietà lateinit è dichiarata e non inizializzata



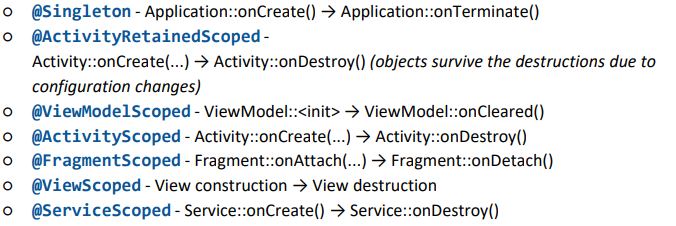


Quando creiamo un modulo, abbiamo la possibilità di specificare cosa il component deve fornire

* Possiamo avere altri component a cui vogliamo ridurre il lyfecycle a quello di un’attività (quando c’è activity → vogliamo che qualcosa sia injected)



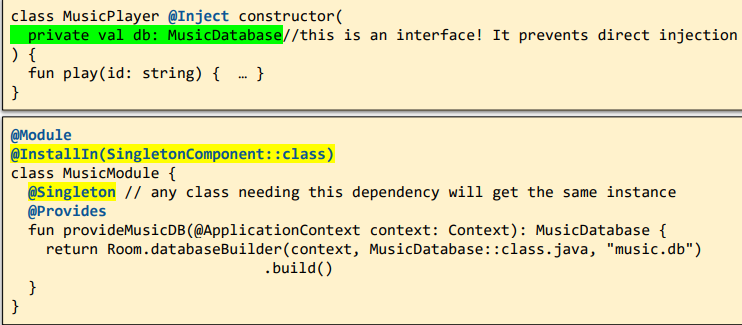
* **Lifecycle dei componenti:**

→ vive con applicazione

## 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



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

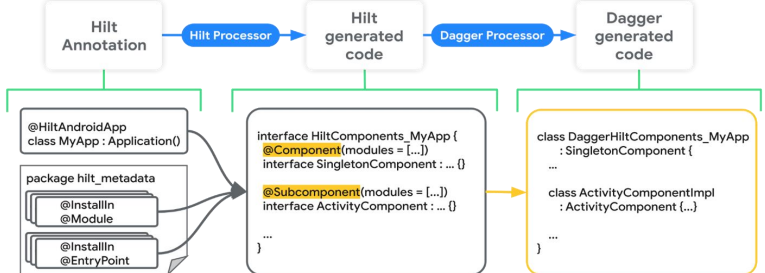
* Al suo interno dichiaro dei metodi
* @Provides: deve fornire qualcosa
* Dobbiamo fornire un contesto (insieme di informazioni , folder, etc...) → forniamo come parametro.

#### JETPACK INTEGRATION:

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamenteComposable 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**)

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*