

Dependency Injection

Torbjørn

Hva er en “dependency”?

En dependency er en klasse som en annen klasse er avhengig av

```
class StorageDrive {}  
  
class Computer {  
    val drive: StorageDrive = StorageDrive()  
}
```

Her er `StorageDrive` er en avhengighet (dependency) i klassen `Computer`

“Dependency” kan referere til både typer og instanser

Hvordan får en klasse sine avhengigheter?

1. Klassen sørger selv for tilordning

```
class Computer {  
    val drive: StorageDrive = StorageDrive()  
}
```

Fordel: Lite kompleksitet

Ulempe: Lite fleksibelt - begrenset mulighet til å lage datamaskiner med ulike typer sekundærminne

2. En annen del av programmet bestemmer

Dette er dependency injection!

Ulike typer dependency injection

Constructor Injection

Avhengigheter tilordnes gjennom konstruktøren

```
class Computer(  
    val drive: StorageDrive  
)  
  
fun main() {  
    val drive = StorageDrive()  
  
    val computer = Computer(drive)  
}
```

Field Injection (evt. setter injection)

Tilordning gjennom instansvariabel (evt. gjennom set-metode)

```
class Computer {  
    lateinit var drive: StorageDrive  
}  
  
fun main() {  
    val computer = Computer()  
  
    computer.drive = StorageDrive()  
}
```

Her brukes `lateinit` for å tillate initialisering etter oppretting av `Computer`-objektet

Hilt

Hilt

Bibliotek for automatisert dependency injection i Jetpack Compose

Statisk: Genererer kode under kompilering

I motsetning løsninger som kobler avhengigheter under kjøring

Anbefalt løsning for Jetpack Compose

```
@HiltAndroidApp
class ExampleApplication : Application()

@AndroidEntryPoint
class MainActivity : ComponentActivity() {}

class StorageDrive()

class Computer @Inject constructor(
    val drive: StorageDrive
)

@Module
@InstallIn(SingletonComponent::class)
object StorageDriveModule {

    @Provides
    @Singleton
    fun provideStorageDrive(): StorageDrive = StorageDrive()
}
```

Minimalt eksempel med Hilt

Minimalt eksempel med Hilt

```
@HiltAndroidApp  
class ExampleApplication : Application()
```

@HiltAndroidApp - iverksetter Hilt's kodegenerering

Må brukes på en subklasse av Application

Minimalt eksempel med Hilt

```
@AndroidEntryPoint  
class MainActivity : ComponentActivity() {}
```

@AndroidEntryPoint - inngangspunkt for Hilt's graf over avhengigheter

Hilt injiserer avhengigheter når et objekt med denne annoteringen opprettes

Minimalt eksempel med Hilt

```
class Computer @Inject constructor(  
    val drive: StorageDrive  
)
```

@Inject constructor(...) - spesifiserer constructor injection via Hilt for denne klassen

Minimalt eksempel med Hilt

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

    @Provides
    @Singleton
    fun provideStorageDrive(): StorageDrive = StorageDrive()
}
```

@Module - Objekter med denne annoteringen forteller Hilt hvordan avhengigheter skal genereres

Minimalt eksempel med Hilt

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

    @Provides
    @Singleton
    fun provideStorageDrive(): StorageDrive = StorageDrive()
}
```

`@InstallIn(SingletonComponent::class)` - spesifiserer en komponent (`SingletonComponent`)

Modulen med annoteringen `@InstallIn` bidrar til avhengighetsgrafen for denne komponenten

`SingletonComponent` sikrer at modulen bare opprettes en gang og eksisterer til appen stoppes

Minimalt eksempel med Hilt

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

    @Provides
    @Singleton
    fun provideStorageDrive(): StorageDrive = StorageDrive()
}
```

@Provides - forteller Hilt hvordan objekter av returtypen (StorageDrive) skal opprettes

@Singleton - forsikrer at den samme instansen injeseres hver gang returtypen blir etterspurt

Hilt med interfaces

Hva om en avhengighet er et grensesnitt?

```
interface StorageDrive {}  
  
class HardDiskDrive() : StorageDrive  
  
class Computer @Inject constructor(  
    val drive: StorageDrive  
)
```

Bruk @Binds

```
@Module
@InstallIn(SingletonComponent::class)
abstract class StorageDriveModule {

    @Binds
    @Singleton
    abstract fun bindStorageDrive(
        impl: HardDiskDrive
    ): StorageDrive = impl
}
```

```
interface StorageDrive {}

class HardDiskDrive() : StorageDrive

class Computer @Inject constructor(
    val drive: StorageDrive
)
```

Nå er modulen `StorageDriveModule` og funksjonen `bindStorageDrive` abstrakte
`@Binds` brukes på `bindStorageDrive`, med `HardDiskDrive` som parameter

Hilt + Room

```
@Dao
interface SomeDao {}

@Database( ... )
abstract class AppDataBase : RoomDatabase() {
    abstract fun someDao(): SomeDao
}
```

Dao + database

```
interface SomeRepository {}

@Singleton
class SomeRepositoryImpl @Inject constructor(
    private val someDao: SomeDao
) : SomeRepository

@HiltViewModel
class SomeViewModel @Inject constructor(
    private val repository: SomeRepository
) : ViewModel()
```

Repository + ViewModel

```
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {

    @Binds
    @Singleton
    abstract fun bindSomeRepository(
        impl: SomeRepositoryImpl
    ): SomeRepository
}
```

Hilt-modul for repository

```

@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(
        @ApplicationContext appContext: Context
    ): AppDatabase {
        return Room.databaseBuilder(
            appContext,
            AppDatabase::class.java,
            "app_db"
        ).build()
    }

    @Provides
    fun provideSomeDao(
        db: AppDatabase
    ): SomeDao = db.someDao()
}

```

Hilt-module for database