Впровадження залежностей - це спосіб обробки залежностей поза залежного класу, коли залежному класу не потрібно нічого робити.

Оскільки визначення впровадження залежності нічого не говорить про те, де повинна відбуватися робота з залежностями (крім як поза залежного класу), розробник повинен вибрати відповідне місце для впровадження залежностей.

Давайте уявимо, що існує хтось інший, хто міг би обробляти залежності, внаслідок чого ні залежний, ні для користувача класи не виконували б цю роботу. Цей спосіб дозволяє працювати з залежностями в додатку безпосередньо.

Впровадження залежностей - це не технологія, фреймворк, бібліотека або щось подібне. Це просто ідея. Ідея працювати з залежностями поза залежного класу (бажано в спеціально виділеній частині). Можна застосовувати дану ідею, не використовуючи ніякі бібліотеки або фреймворки. Проте, краще звертатися до фреймворків для впровадження залежностей, тому що це спрощує роботу і дозволяє уникнути написання шаблонного коду. Будь-який фреймворк впровадження залежностей має дві невід'ємні характеристики.

По-перше, дані фреймворки пропонують спосіб визначення полів (об'єктів), які повинні бути впроваджені. Деякі фреймворки здійснюють це за допомогою анотування поля або конструктора за допомогою анотації @Inject, але існують і інші методи. Наприклад, Koin використовує вбудовані мовні особливості Kotlin для визначення впровадження. Під Inject мається на увазі, що залежність повинна оброблятися DI-фреймворком. Код буде виглядати приблизно так:

class ClassA {

var classB: ClassB

@Inject constructor(classB: ClassB){

this.classB = classB

}

}

class ClassB {

var classC: ClassC

@Inject constructor(classC: ClassC){

this.classC = classC

}

}

class ClassC {

@Inject constructor(){

}

}

По-друге, фреймворки дозволяють визначити, як потрібно надати кожну залежність, і це відбувається в окремому файлі (файлах). Приблизно це виглядає так:

class OurThirdPartyGuy {

fun provideClassC(){

return ClassC() //just creating an instance of the object and return it.

}

fun provideClassB(classC: ClassC){

return ClassB(classC)

}

fun provideClassA(classB: ClassB){

return ClassA(classB)

}

}

Отже, як можна бачити, кожна функція відповідає за обробку однієї залежності. Тому якщо нам десь в додатку потрібно використовувати ClassA, то відбудеться наступне: наш DI-фреймворк створює один екземпляр класу ClassC, викликавши provideClassC, передавши його в provideClassB і отримавши примірник ClassB, який передається в provideClassA, і в результаті створюється ClassA.

Переваги такого методу.

* Все максимально просто. І залежний клас, і клас, що надає залежності, зрозумілі і прості.
* Класи слабо пов'язані і легко замінні іншими класами. Припустимо, ми хочемо замінити ClassC на AssumeClassC, який є підкласом ClassC. Для цього потрібно лише змінити код провайдера наступним способом, і всюди, де використовується ClassC, тепер автоматично буде використовуватися нова версія:

fun provideClassC(){

return AssumeClassC()

}

* Чудова можливість для тестування. Можна легко замінити залежності тестовими версіями під час тестування.
* Поліпшення структури коду, тому що в додатку є окреме місце для обробки залежностей. В результаті, інші частини програми можуть виконувати виключно свої функції і не перетинатися з залежностями.

Недоліки методу.

* У DI-фреймворків є певний поріг входження, тому команда проекту повинна витратити час і вивчити його, перш ніж ефективно використовувати.