Inversion of Control
Dependency injection

Inversion of control. Dependency Injection

Inversion of control (IOC) - это подход к написанию программ, при котором приложение не реализует весь поток выполнения приложения. Вместо этого приложение реализует только специфические, для себя, функции. Обязанность по управление потоком выполнения приложения ложиться, обычно, на какой-либо фреймворк. Фреймворк вызывает функции, реализуемого приложения, в соответствии с требованием потока выполнения.

Dependency injection (DI) - это частный случай **IOC**. При **DI**, приложение не управляет явно своими зависимостями, а лишь описывает их. Зависимости предоставляются приложению DI фреймворком, в соответствии с описанием, на этапе компиляции или в процессе выполнения приложения

Dependency Injection

IOC и **DI**, в частности, решают 4 основные задачи

- разделение кода по зонам ответственности
- отделение выполнения приложения от реализации
- понижение связанности кода. Внесение в код контрактов
- упрощение модификации и переиспользования кода

Примером **IOC** может служить контейнер сервлетов. Реализуя код контроллера для обработки запросов, программист не заботиться о создании соединения, роутинге запросов и сериализации\десериализации данных. Вместо этого, код контроллера будет вызван в нужный момент, в соответствии с потоком обработки запросов

Dependency Injection

Способы реализации **DI** на scala, можно разделить на 2 большие категории

- с применением DI фреймворка (DI контейнера)
  - MacWire
  - AirFrame
  - Guice
  - ScalaDi
- реализация **DI**, средствами языка
  - Functions and methods
  - Reader monad
  - Cake pattern

Dependency Injection

Зачем нужен **DI**

Основная причина появления DI в проекте - желание разработчиков избавится от ответственности за зависимости. С ростом количества зависимостей и с усложнением их взаимозависимостей управление ими становится все сложнее. Код загромождается параметрами, которые нужны только для того, чтобы передать их на следующий уровень иерархии приложения. Для решения этих проблем управление зависимостями передают контейнеру зависимостей или применяют техники, снижающие сложность, связанную с зависимостями.

Dependency Injection

Применение **DI** в любом приложении состоит из 3-х шагов

- Первый шаг - отделение зависимостей от реализации логики приложения. На этом шаге очень полезными оказываются принципы **SOLID** и простой здравый смысл.
- После того, как зависимости определены, наступает время выбрать способ реализации **DI**. На этом шаге большую роль играет объем и сложность приложения и, в немалой степени, привычки и опыт конкретного разработчика
- Последний этап - реализация задуманного)

Dependency Injection

В пакете классе **lectures.di.naive.NaiveUserServiceProgram** реализована маленькая программа которую мы будем препарировать. Мы пройдем все три основные шага внедрения **Di**, а именно :

- разделим приложение на компоненты и выделим зависимости
- выберем несколько подходящих **DI** методик
- и по очереди их реализуем

Dependency Injection

Самый простой способ думать о DI - это представлять приложение, как функции, а DI, как передачу в нее параметров.

Благодаря мощному синтаксису SCALA мы получаем возможность комбинировать компоненты приложения с помощью методов функций **compose**, **andThan**, **curried** и т.д.

К сожалению, эти функции позволяют нам комбинировать только функции, возвращающие значения, но не функции, возвращающие другие функции.

Для решения задачи композиции функций высших порядков была изобретена **reader monad**

Dependency Injection

Reader Monad (RM). В этом разделе мы подробно остановимся на том, как **RM** помогает решить задачи **DI**. Что такое монада и почему Reader, это тоже монада, мы рассмотрим в дальнейших разделах.
Возьмем упрощенный синтаксис Reader из библиотеки Cats

```
type Id[A] = A

type ReaderT[F[_], A, B] = Kleisli[F, A, B]

object Reader {
 def apply[A, B](f: A => B): Reader[A, B] = ReaderT[Id, A, B](f)
}

final case class Kleisli[F[_], A, B](run: A => F[B]) {
 def map[C](f: B => C)(implicit F: Functor[F]): Kleisli[F, A, C] = ???
 def mapF[N[_], C](f: F[B] => N[C]): Kleisli[N, A, C] = ???
 def flatMap[C](f: B => Kleisli[F, A, C])(implicit F: FlatMap[F]): Kleisli[F, A, C] = ???
  ...
```

Dependency Injection

По сути, **RM** - это контейнер для функции из **A => B**. Метод **map** можно рассматривать как аналог **andThen** для **A => B**, а **flatMap** - это метод композиции функций **A => B** и **B => A => C,** который возвращает функцию **A => C**

Рассмотрим, как **RM** может пригодиться для реализации DI на примере **lectures.di.reader.ReaderMonadProgram**

Dependency Injection

Cake Pattern (CP) Суть метода состоит в том, что приложение разделяется на “слои”. Каждый такой слой представляет собой атомарную часть приложения, реализованный в виде trait-а или композиции трейтов. Каждый слой описывает свою зависимость от других слоев, через наследование от трейтов соответствующих слоев. Таким образом приложение, спроектированное с помощью **CP** представляет собой класс или трейт наследующий от набора слоев, которые в свою очередь могут быть составлены из других слоев. Непосредственно **DI** происходит в момент создания инстанса, синглтон класса, наследующего конкретные реализации трейтов-слоев.

Dependency Injection

В **CP** для обозначения зависимости одного слоя (trait-а) от другого, вместо наследования с помощью ключевых слов extends и with используют композицию с помощью **self type annotation**
Почему self annotation предпочитают наследованию мы рассмотрим на примере: **lectures.di.cake.SelfAnnotationExplained**
CP, очень гибкий метод, при грамотном использовании, он позволяет создавать приложения практически неограниченной сложности. Тем не менее, в его применении есть ограничение, связанное с порядком инициализации зависимостей. Поскольку, вследствии линеизации, члены трейтов инициализируются в определенной порядке, необходимо следить, чтобы ни один член приложения не использовался до своей инициализации. Иначе можно столкнуться с NullPointerException.
Рассмотрим такой случай на примере **lectures.di.cake.CakePatternCaveat**

Dependency Injection

Развернутый пример **CP** находиться в **lectures.di.cake.CakeUserProgram.** В этом классе мы реализуем уже знакомое нам приложение с помощью cake pattern dependency injection

Dependency Injection

MacWire (MW)- это пример легковесного DI контейнера, реализованного с помощью scala macros.
MW - это open source проект с развитым community и документацией**.** На github проекта можно найти все необходимые подробности о том, как применять библиотеку и, о DI в общем.
К преимуществам macwire, можно отнести

- отсутствие проблем с инициализацией зависимостей
- простой способ описания зависимостей

Небольшим недостатком можно считать

- зависимость на стороннюю библиотеку
- чуть большее время компиляции

Пример реализации уже знакомого нам приложения с применением **MacWire lectures.di.macwire.MacWireProgram**

Dependency Injection

Домашнее задание находится в **lectures.di.domain.scala**