DI (Dependency Injection)

허가받지 않은 복제(복사), 전송, 수정 및 배포를 금합니다.

▼ 의존 관계란? (Dependency)

▼ A가 B를 의존한다.

B가 변하면, A에 영향을 미친다.

바리스타가 아메리카노 커피를 만든다.
커피를 만들기 위해서는 레시피가 필요하다.

👉 **바리스타는 커피 레시피에 의존한다.**

```
class Barista {
  final AmericanoRecipe recipe;

  Barista(this.recipe);

  void makeCoffee() {
    print(recipe.info);
  }
}
```

만약, 카페라떼로 바꾼다면?

클래스를 바꿔줘야함.

```
class Barista {
  final CafeLatte recipe;
  /// 생략
}
```

▼ 이걸 해결 할 수 없을까? 🤔

추상화 클래스 `abstract class` **(인터페이스)**

```
Barista(AmericanoRecipe()).makeCoffee();

class Barista {
  final CoffeeRecipe recipe;

  Barista(this.recipe);

  void makeCoffee() {
    print(recipe.info);
  }
}

abstract class CoffeeRecipe {
  String get info;
}

class AmericanoRecipe implements CoffeeRecipe {
  @override
  String get info => '2 espresso shot + water';
}

class CafeLatteRecipe implements CoffeeRecipe {
  @override
  String get info => 'milk + 1 espresso shot + water';
}
```

▼ 의존성 주입이란? (Dependency Injection)

위 예시에서 어떤 Coffee 를 제조 할 지 **내부** 코드에서 정하고 있음.
어떤 커피를 제조 할 지를 **외부**에서 정한다면? 예를 들면, 손님이 원하는 커피 주문

👉 **의존성 주입**

▼ 왜 필요할까?

- **의존성이 줄어듬** 👍
  클래스간의 결합도가 줄어듬.
  다른걸로 교체하는데 편함.
- **테스트 하기 좋은 코드** 👍
  독단적으로 분리해서 테스트 하기 매우 좋음.
- **가독성이 높아짐**
  추상화 클래스로 기능 별도 분리

Injection 종류

Constructor Injection

```
Barista(AmericanoRecipe());
Barista(CafeLatteRecipe());
```

Setter Injection

```
final barista = Barista();
barista.setRecipe(AmericanoRecipe());
```

Method Injection

```
final barista = Barista();
barista.makeCoffee(AmericanoRecipe());
```

👉 *Constructor Injection*

실제로 어떻게 적용되고 있을까?

Dagger (Android)

- **@annotation 으로 편하게 사용 할 수 있음.** by google.
  - `@Module`
  - `@Component`
  - `@Singleton`
  - `@Inject`
- **컴파일 단계에서 결정 됨.**

Service Locator

```
class BaristaLocator {
  static CoffeeRecipe locator() => AmericanoRecipe();
}

BaristaLocator.locator();
```

- **pattern**
- **일종의 중개자 역할**
- **런타임에서 결정**
  - 도중 교체 가능

Service Locator 단점

왜 안티 패턴이라고 불릴까 ? 😂

- **전역 접근**
- **locator 별도로 구현**

Flutter 에서는 어떻게 쓸까?

get_it

This is a simple **Service Locator** for Dart and Flutter projects with some additional goodies highly inspired by Splat

```
final getIt = GetIt.instance;

void setup() {
  getIt.registerSingleton<AppModel>(AppModel());
}

MaterialButton(
```

```
  child: Text("Update"),
  onPressed: getIt<AppModel>().update   // given that your AppModel has a method update
),
```

기본 원리

```
// https://github.com/fluttercommunity/get_it/blob/master/lib/get_it_impl.dart#L289C1-L290

class _Scope {
  final factoriesByName = <String?, Map<Type, _ServiceFactory<Object, dynamic, dynamic>>>{};
}
```

get_it 주요 기능

- **Factory**

  ```
  void registerFactory<T>(FactoryFunc<T> func)
  ```

- **Singleton & LazySingleton**

  ```
  void registerSingleton<T>(T instance)
  ```

  - LazySingleton: 리셋 가능 (`resetLazySingleton`)
- **Overwriting registrations**
  - debug mode 에서는 assert 에 잡힘.
- **Testing if a Singleton is already registered**

  ```
  bool isRegistered<T>({Object instance, String instanceName});
  ```

- **Unregistering Singletons or Factories**

  ```
  void unregister<T>({Object instance,String instanceName, void Function(T) disposingFunction})
  ```

- **Resetting GetIt completely**
  - 테스트 코드 작성에 용이

  ```
  Future<void> reset({bool dispose = true});
  ```

Singleton

객체의 인스턴스가 오직 1개만 생성해서 **여러 곳**에서 사용.

장점 (Why?)

- 이미 생성되어 있는 인스턴스 사용 👉 속도 측면 유리
- 데이터 공유하기 편함 👉 개발자 편의

단점

- `static` 메모리상 할당 후 종료될 때까지 메모리에 유지됨
  👉 지나치게 쓸 경우, 메모리 낭비
- 결합도가 높아짐
  - 개방-폐쇄 원칙 위반
    객체지향 5원칙 (SOLID)
- 멀티쓰레딩 환경에서 동시성 문제 발생 가능성
  👉 대부분 언어 / 플랫폼에서 해결한 방법이 존재함

```
class AppModel {
  static final AppModel _instance = AppModel._internal();

  factory AppModel() => _instance;

  final String name = 'fast_campus';

  AppModel._internal() {
    // 인스턴스가 최초 생성될 때, 1회 발생하는 초기화 코드
  }

  void update() {

  }
}


AppModel().name;
AppModel().update();
```

injectable

Injectable is a **convenient code generator** for get_it.

- **@annotation 활용하여 구현**
  - `@singleton`
  - `@injectable`
- **가독성** 👍

as-is

```
class ServiceA {}

class ServiceB {
    ServiceB(ServiceA serviceA);
}

GetIt.instance.factory<ServiceA>(() => ServiceA());
GetIt.instance.factory<ServiceB>(ServiceB(getIt<ServiceA>()));
```

to-be

```
@injectable
class ServiceA {}

@injectable
class ServiceB {
    ServiceB(ServiceA serviceA);
}


/// Inside of the generated file
/// GENERATED CODE - DO NOT MODIFY BY HAND
import 'package:get_it/get_it.dart' as _i1;

extension GetItInjectableX on _i1.GetIt {
  /// initializes the registration of main-scope dependencies inside of [GetIt]
  Future<_i1.GetIt> init({
    String? environment,
    _i2.EnvironmentFilter? environmentFilter,
  }) async {
    final gh = _i2.GetItHelper(
      this,
```

```
      environment,
      environmentFilter,
    );
    gh.factory<ServiceA>(() => ServiceA());
    gh.factory<ServiceB>(() => ServiceB(getIt<ServiceA>()));
    return this;
  }
}
```

GetX

> GetX is an extra-light and powerful solution for Flutter.
> It combines high-performance state management, **intelligent dependency injection**, and route management quickly and practically.

- `GetInstance`

```
/// https://github.com/jonataslaw/getx/blob/master/lib/get_instance/src/extension_instance.dart#L53
Map<String, _InstanceBuilderFactory> _singl = {};

/// https://github.com/jonataslaw/getx/blob/master/lib/get_instance/src/extension_instance.dart#L497
class _InstanceBuilderFactory<S> { S? dependency; String? tag; }
```

참고 자료

https://developer.android.com/training/dependency-injection/dagger-android?hl=ko

get_it | Dart Package

❤️ Sponsor This is a simple Service Locator for Dart and Flutter projects with some additional goodies highly inspired by Splat. It can be used instead of InheritedWidget or Provider to access objects e.g. from your UI. Typical usage:

https://pub.dev/packages/get_it