Методическое указание к Практической работе №8 по дисциплине «Разработка кроссплатформенных мобильных приложений»

План практической работы:

- Изучить проблему зависимостей в проектах;
- Произвести работу с Inherited Widget;
- Произвести подключение и работу с DI контейнером GetIt;
- Выполнение практической работы №8;

Последовательность выполнения практической работы:

1. Проблема зависимостей в проекте

Ранее в практических работах мы уже сталкивались с необходимостью передавать один или более параметр в конструктор при создании объекта того или иного класса. Бывают ситуации, когда аргументов не много или же все их значения заранее известны, либо вычисляются в данном месте (Рисунок 1). В таких ситуациях создать объект такого класса не составляет никакого труда.

```
class MyClass {
  const MyClass({required String arg1});
}

Run | Debug | Profile
void main(List<String> args) {
  const obj = MyClass(arg1: 'arg');
}
```

Рисунок 1 – Класс с «простым» конструктором

Однако бывают ситуации, когда перечень аргументов становится многочисленным, а также значения аргументов конструктора уже невозможно рассчитать на месте и их приходится брать из других мест (Рисунок 2). В таких ситуациях приходится делать место, в котором будет создаваться объект такого сложного класса зависимым от тех данных, которые необходимы в конструкторе. При достижении определенного уровня такой вложенности становится все тяжелее взаимодействовать с программным кодом и требуется решать такого рода проблемы.

Рисунок 2 – Конструктор с большим перечнем полей

Самым простым способом решения проблемы поочередной зависимости классов от какого-либо набора данных является вынос этих данных во внешнее пространство, к которому у этих классов будет доступ. Это позволяет не передавать зависимость из класса в класс, а получать ее напрямую из хранилища. Такой принцип называется – внедрение зависимостей или, как в дальнейшем будет называться, DI. В фреймворке Flutter есть разные способы реализации DI, однако рассмотрено будет 2 из них: Inherited Widget и DI контейнер GetIt.

2. Inherited Widget

Inherited Widget является одним из основных типов Widget-ов в фреймворке Flutter. Он в основном выступает в качестве внешнего хранилища данных, располагаемых в дереве виджетов, как любой другой Widget. В отличии от Stateless и Stateful Widget, Inherited Widget не имеет и не

предполагает какого-либо отображения на экране, поэтому имеет отличные как внутреннее устройство, так и принцип взаимодействия.

Все Widget-ы, зависящие от данных в Inherited Widget подписываются на его состояние и получают данные по структуре Widget-ов. Если Widget имеет собственный State, то его жизненный цикл подразумевает автоматическое обновление при обновлении данных в Inherited Widget при помощи метода didChangedDepencies. Однако получить данные без автоматической актуализации данных может любой виджет, у которого есть собственный context. По нему Widget ищет объект класса наследника Inherited Widget выше по дереву и возвращает объект этого класса, если такой был найден.

2.1 Статический метод of

Основным требованием к хранилищу зависимостей является доступ к нему из разных мест приложения. Данное требование в Inherited Widget и его наследниках реализуется посредством реализации статического метода of. Данный метод позволяет найти объект класса Inherited Widget или его наследника в виджете дерева выше Widget-а, в котором необходима зависимость. В связи с тем, что объект класса Inherited Widget или его наследника будет искаться в структуре Widget-ов, то для его корректной логики требуется context Widget-а, в котором необходима зависимость. Пример синтаксиса статического метода of проиллюстрирован на рисунке 3.

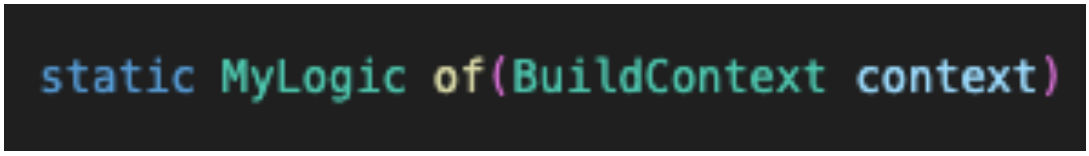

Рисунок 3 – сигнатура статического метода of

2.2 Метод dependOnInheritedWidgetOfExactType

Для поиска объекта класса Inherited Widget или его наследников фреймвокр предоставляет специальный метод dependOnInheritedWidgetOfExactType, который возвращает объект класса, указанный в качестве дженерик типа в случае, если объект был найден выше по дереву, или пустоту, если объекта данного класса обнаружено не было. Важно заметить, что этот метод используется для поиска объекта по дереву, а значит его нужно вызывать у context, объекта класса BuildContext. Именно для этого метод of обязательно должен иметь в качестве аргумента context Widget-а, в котором нужна зависимость. Полную реализацию метода of можно увидеть на рисунке 4.

Важно заметить, что метод dependOnInheritedWidgetOfExactType вернет первый найденный объект класса, указанный в качестве дженерик класса. То есть если в дереве Widget-ов находятся два Widget-а одного Inherited Widget-а или его наследника, то метод dependOnInheritedWidgetOfExactType вернет именно ближайший Widget, расположенный верх по дереву от зависимого от данных Widget-а.

```
static MyLogic of(BuildContext context) =>
    context.dependOnInheritedWidgetOfExactType<MyLogic>()!;
```

Рисунок 4 – использование метода dependOnInheritedWidgetOfExactType в статическом методе of

2.3 Метод updateShouldNotify

Так как Inherited Widget и его наследники используются для передачи набора данных в Widget-ы, где эти данные требуются, то они также должны иметь реализацию автоматического оповещения этих виджетов при обновлении хранимых данных. Этот механизм реализован при помощи метода updateShouldNotify, возвращающего логическое значение, которое описывает необходимость слушателям этих данных обновить свои состояния, так как данные обновились. Разработчик вправе сам регулировать, когда и при каких условиях слушатели обновят свои состояния, именно регулируя логику возвращения данного логического значения, сравнивая экземпляр класса до изменения и после изменения. Фреймвокр самостоятельно сохраняет состояние объекта до изменения и предоставляет его в данный метод через аргумент oldWidget. Пример реализации логики метода updateShouldNotify можно увидеть на рисунке 5.

```
@override
bool updateShouldNotify(covariant MyLogic oldWidget) =>
  oldWidget.number != number;
```

Рисунок 5 – реализация метода updateShouldNotify на основе внутреннего поля number

3. DI контейнер GetIt

GetIt – это DI контейнер, который распространяется в одноименном пакете. Это простой DI контейнер для проектов Dart и Flutter с некоторыми дополнительными преимуществами. Его можно использовать вместо InheritedWidget для доступа к объектам, например, из вашего пользовательского интерфейса. Главное отличие GetIt от Inherited Widget заключается в том, что для доступа к данным не требуется context Widget-a, от которого происходит обращение. Очень часто в ходе разработки требуется получить данные в месте, где нет доступа к context и в таких моментах Inherited Widget становится очень неудобен. GetIt отлично закрывает эту проблему, так как он не является частью фреймворка Flutter и не зависит от его внутренней реализации. GetIt строится на объектном паттерне Singleton, который позволяет получить доступ к единому экземпляру классу в любой точке приложения без необходимости как-то добавлять его в дерево Widget-ов.

3.1 Подключение в проект

Для подключения пакета get_it в проект требуется в файл pubspec.yaml добавить пакет в dependency. Пример добавления изображен на рисунке 6.

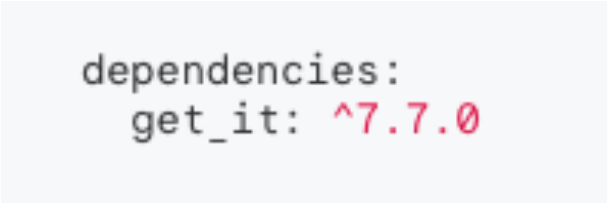

Рисунок 6 – добавления пакета get_it в приложение

Альтернативным вариантом является введение команды «flutter pub add get_it» в терминале, смотрящим в директорию проекта.

3.2 Доступ к DI контейнеру

Для того, чтобы получить доступ к DI контейнеру GetIt, необходимо обратиться к его статичному полю instance или полю I. Данное поле хранит в себе единый объект DI контейнера, используемого в приложении. Получив доступ к объекту GetIt, разработчику открывается полный доступ к контейнеру и лежащих в нем образах. Способ обращения к контейнеру можно рассмотреть на рисунке 7

```
GetIt getIt = GetIt.instance;

//There is also a shortcut (if you don't like it just ignore it):
GetIt getIt = GetIt.I;
```

Рисунок 7 – пример обращения к контейнеру GetIt

3.3 Регистрация объектов

Для того, чтобы некий образ класса появился в контейнере его необходимо сначала там зарегистрировать. В контейнере можно зарегистрировать образ как уже готовый объект или как алгоритм его создания. Для регистрации объектов в контейнере GetIt есть специализированный метод registerSingleton<T>. Указывая в качестве значения его аргумента объект дженерик класса, он регистрируется в контейнере и будет возвращаться каждый раз, когда из контейнера будут запрашивать образ данного класса. (Рисунок 8)

```
// if you want to work just with the singleton:
GetIt.instance.registerSingleton<AppModel>(AppModelImplementation());
GetIt.I.registerLazySingleton<RESTAPI>(() => RestAPIImplementation());
```

Рисунок 8 – регистрация объектов в контейнер GetIt

Иногда требуется хранить более одного объекта одного и того же класса в контейнере, тогда при регистрации объекта помимо самого объекта передают его идентификационное название, по которому в дальнейшем можно будет получить именно этот образ запрашиваемого класса. (Рисунок 9)

Рисунок 9 – указание идентификационное имени объекта в контейнере GetIt

При стандартной настройке контейнера если попытаться зарегистрировать объект уже имеющегося в контейнере класса, то будет получена runtime ошибка. Для решения этой проблемы можно либо задать каждому экземпляру собственное идентификационное название или включить настройку переписи объектов внутри контейнера. (Рисунок 10) Она позволяет заменять в контейнере объект одного класса без ручной очистки образа класса.

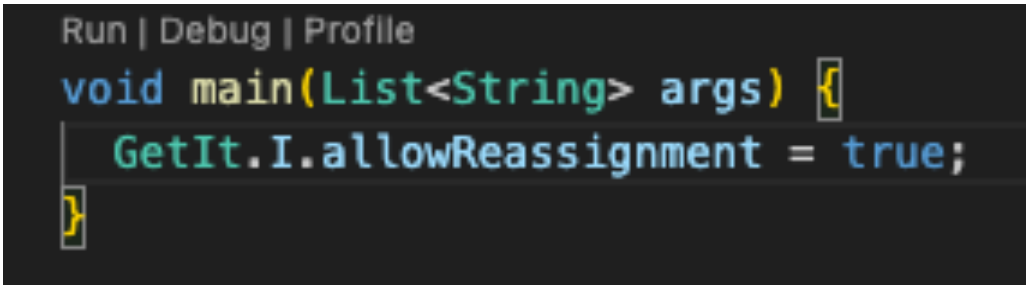

Рисунок 10 – включение настройки переписи объектов в контейнере GetIt

Так как регистрация объектов в контейнере – это очень затратный процесс по памяти, GetIt имеет возможность произвести «отложенную» регистрацию. Для этого необходимо вызвать метод registerLazySingleton<T>, в аргументы которого передается функция, возвращающая требуемых объект дженерик класса. (Рисунок 8) Это позволяет заложить в контейнер не сам объект, а функцию его получения, что уменьшает потребляемую память и позволяет заложить в контейнер объект на момент первого обращения к образу класса.

3.4 Регистрация фабрик создания образов

Не во всех случаях требуется хранить объект класса в контейнере. Существуют подходы, что классы описывают логику работы алгоритмов, однако не хранят в своих объектах никаких данных. Так как хранение объектов в контейнере очень затратный процесс по памяти, то хранить такого рода классы нет никакой необходимости. Вместо этого, контейнер позволяет сохранять в качестве образа не сам объект, а логику получение образа класса. Такие функции с логикой создания объектов требуемых классов называются фабриками. Для регистрации фабрики в качестве образа класса в контейнере используется метод registerFactory<T>. (Рисунок 11) При обращении в контейнер за образом класса будет каждый раз получено новый объект данного класса, построенный по логике, заложенной в фабрике.

Так же при регистрации фабрики ей можно задать идентификационное название, что бы отличать различные фабрики, возвращающие образ одного и того же класса.

Рисунок 11 – регистрация фабрики в контейнер GetIt

3.5 Получение образа из контейнера

В ситуации, когда требуется получить образ из контейнера, требуется использовать специализированный метод get<T>, который вернет образ запрашиваемого дженерик класса. (Рисунок 12) Если образ не был ранее зарегистрирован в контейнере, то данный метод вернет runtime ошибку, что такого образа в контейнере нет.

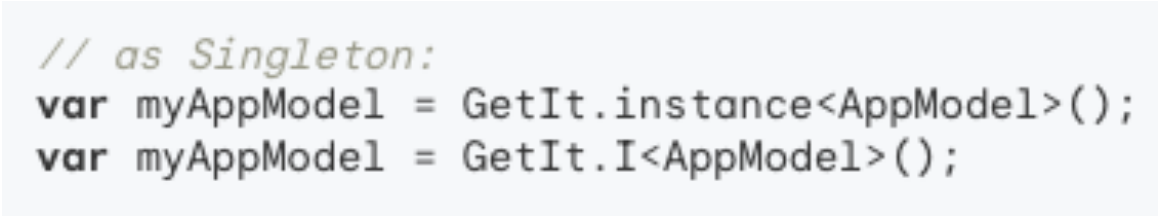

Рисунок 12 – получение образа из контейнера GetIt

3.6 Проверка на наличие образа в контейнере

Для того, чтобы быть уверенным в получении образа искомого класса из контейнера, GetIt позволяет проверять, зарегистрирован ли образ класса до того, как его получать. Для проверки регистрации образа используется метод isRegistered<T>. (Рисунок 13) Данный метод возвращает логическое значение, обозначающее регистрацию образа дженерик класса в контейнере. Так же, как и в методах регистрации, при помощи дополнительного аргумента можно указать идентификационное название для проверки регистрации конкретного образа дженерик класса.

Рисунок 13 – регистрация фабрики в контейнер GetIt

Задание на практическую работу

В качестве задания на практическую работу, студенту в индивидуальном порядке требуется подготовить следующее задание.

Студенту необходимо переработать имеющееся приложение, выполненное на предыдущих практических работах, таким образом, чтобы в нем производилась передача параметров в рамках всего приложения двумя изученными способами: через Inherited Widget и через DI контейнер GetIt.

Студенту требуется выполнить все пункты в плане практической работы с конспектированием своих действий, а также фотофиксацией контрольных точек в отчете о практической работе.