1. \*\*Что такое mixins? Примеры случаев использования.\*\*

Mixins — это механизм повторного использования кода в объектно-ориентированных языках программирования, таких как Dart. Они позволяют добавлять функциональность к классам без необходимости наследования от другого класса. В отличие от классов, миксины нельзя инстанцировать напрямую — они просто "встраиваются" в классы, к которым подключаются.

\*\*Пример использования mixins:\*\*

- Добавление методов отладки или тестирования в класс без изменения его основной функциональности.

- Повторное использование общих методов (например, логгирование или валидация данных) для различных классов.

\*\*Пример кода:\*\*

```dart

mixin Debuggable {

void debug() {

print("Debugging...");

}

}

class Developer with Debuggable {

String name;

Developer(this.name);

}

void main() {

Developer dev = Developer('Alice');

dev.debug(); // Выведет: Debugging...

}

```

2. \*\*Какой функционал предоставляет интерфейс `Comparable`. Основные методы.\*\*

Интерфейс `Comparable` предоставляет механизм сравнения объектов, чтобы можно было их сортировать или определять порядок.

\*\*Основной метод:\*\*

- `compareTo(other)`: Сравнивает текущий объект с другим объектом и возвращает:

- Отрицательное значение, если текущий объект "меньше" другого.

- Ноль, если объекты равны.

- Положительное значение, если текущий объект "больше" другого.

\*\*Пример кода:\*\*

```dart

class Developer implements Comparable<Developer> {

String name;

int experienceYears;

Developer(this.name, this.experienceYears);

@override

int compareTo(Developer other) {

return experienceYears.compareTo(other.experienceYears);

}

}

```

3. \*\*Какой функционал предоставляет интерфейс `Iterator`. Основные методы.\*\*

Интерфейс `Iterator` позволяет последовательно обходить коллекцию данных.

\*\*Основные методы:\*\*

- `bool moveNext()`: Перемещает итератор к следующему элементу и возвращает `true`, если удалось перейти к следующему элементу, и `false`, если элементов больше нет.

- `T get current`: Возвращает текущий элемент коллекции.

\*\*Пример кода:\*\*

```dart

class DevExperienceIterator implements Iterator<String> {

final List<String> experienceList;

int \_index = -1;

DevExperienceIterator(this.experienceList);

@override

bool moveNext() {

if (\_index < experienceList.length - 1) {

\_index++;

return true;

}

return false;

}

@override

String get current => experienceList[\_index];

}

```

4. \*\*Какой функционал предоставляет интерфейс `Iterable`. Основные методы.\*\*

`Iterable` представляет коллекцию, по которой можно итерироваться (то есть проходить с использованием `Iterator`).

\*\*Основной метод:\*\*

- `Iterator<T> get iterator`: Возвращает итератор для коллекции.

\*\*Пример кода:\*\*

```dart

class DevExperienceIterable extends Iterable<String> {

final List<String> experienceList;

DevExperienceIterable(this.experienceList);

@override

Iterator<String> get iterator => DevExperienceIterator(experienceList);

}

```

5. \*\*Понятие асинхронности.\*\*

Асинхронность — это механизм, позволяющий программе выполнять задачи параллельно, не блокируя основной поток выполнения. Асинхронные операции, такие как запросы к серверу или чтение файлов, могут выполняться в фоновом режиме, и результат будет доступен по завершении операции.

\*\*Пример асинхронного кода:\*\*

```dart

Future<void> fetchData() async {

print("Fetching data...");

await Future.delayed(Duration(seconds: 2)); // Симуляция ожидания

print("Data fetched");

}

```

6. \*\*Что такое `Future`. Примеры использования.\*\*

`Future` представляет собой результат асинхронной операции, который будет доступен в будущем. Метод, возвращающий `Future`, указывает, что его результат будет доступен позже.

\*\*Пример использования `Future`:\*\*

```dart

Future<String> fetchDeveloperData() {

return Future.delayed(Duration(seconds: 2), () => 'Alice, Frontend Developer');

}

```

7. \*\*Что такое `Stream`. Примеры использования.\*\*

`Stream` — это поток данных, который передает данные по мере их появления, и позволяет обрабатывать их поэтапно. Используется для работы с событиями или непрерывным потоком данных, например, для получения данных с сервера или обработки пользовательских действий.

\*\*Пример использования `Stream`:\*\*

```dart

Stream<int> generateNumbers() async\* {

for (int i = 1; i <= 5; i++) {

await Future.delayed(Duration(seconds: 1));

yield i; // Отправляет значение в поток

}

}

```

8. \*\*Отличие `Single subscription stream` и `BroadcastStream`.\*\*

- \*\*Single subscription stream\*\*: Позволяет только одному слушателю подписаться на поток. После того как слушатель подписан, другие не смогут получить доступ к этому потоку. Этот поток закрывается после завершения.

\*\*Пример:\*\*

```dart

Stream<int> generateNumbers() async\* {

for (int i = 1; i <= 5; i++) {

await Future.delayed(Duration(seconds: 1));

yield i;

}

}

```

- \*\*Broadcast stream\*\*: Позволяет нескольким слушателям подписаться на один и тот же поток. Это полезно, когда требуется обрабатывать один поток событий для нескольких подписчиков одновременно.

\*\*Пример:\*\*

```dart

Stream<int> broadcastStreamExample() {

return Stream<int>.periodic(Duration(seconds: 1), (x) => x).asBroadcastStream();

}

```