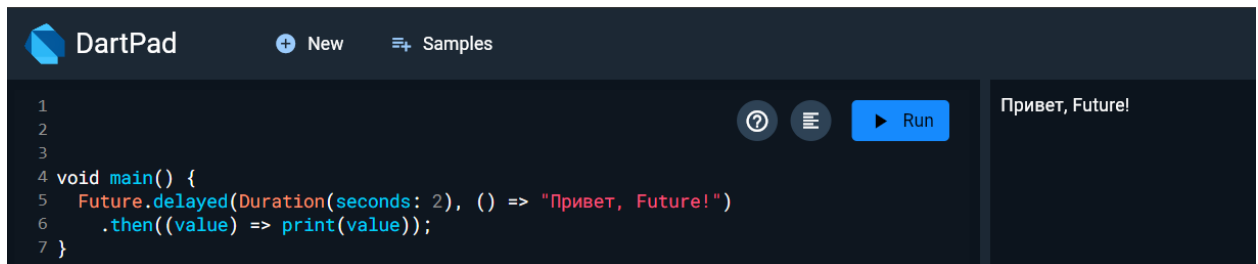


Задание № 1: Простой Future

Создайте Future, который через 2 секунды возвращает строку «Привет, Future!». Выведите эту строку в консоль.

Описание алгоритма решения:

1. Создайте Future с помощью конструктора Future.delayed.
2. Передайте ему длительность задержки (2 секунды) и анонимную функцию, которая возвращает строку «Привет, Future!».
3. Используйте then для вывода строки в консоль после завершения Future.



```
1
2
3
4 void main() {
5   Future.delayed(Duration(seconds: 2), () => "Привет, Future!")
6     .then((value) => print(value));
7 }
```

Привет, Future!

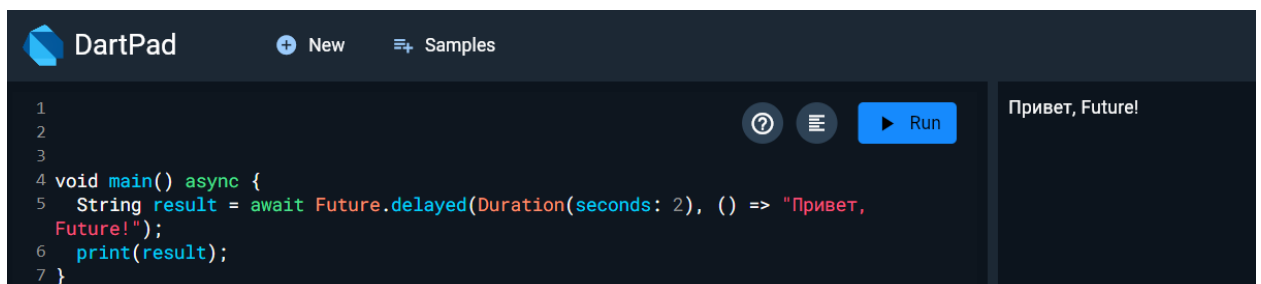
Рисунок №1 – Результат решения задания №1.

Задание № 2: Использование async/await

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await для ожидания завершения Future.
3. Выведите результат Future в консоль.



```
1
2
3
4 void main() async {
5   String result = await Future.delayed(Duration(seconds: 2), () => "Привет,
6     Future!");
7   print(result);
8 }
```

Привет, Future!

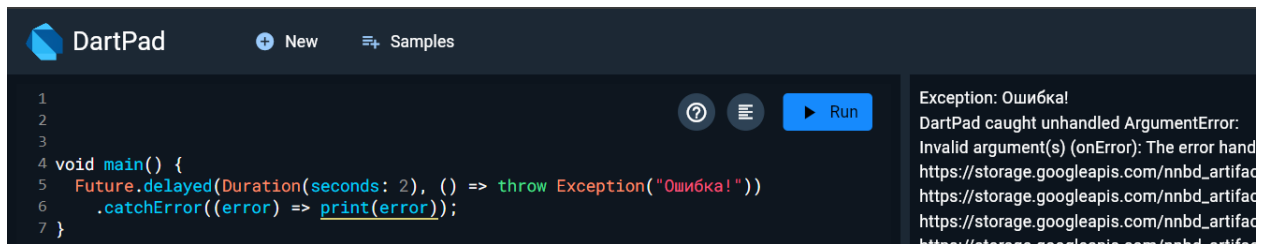
Рисунок №2 – Результат решения задания №2.

Задание № 3: Обработка ошибок в Future

Создайте Future, который через 2 секунды генерирует исключение. Обработайте это исключение и выведите сообщение об ошибке в консоль.

Описание алгоритма решения:

1. Создайте Future с помощью конструктора Future.delayed.
2. Передайте ему длительность задержки (2 секунды) и анонимную функцию, которая генерирует исключение.
3. Используйте catchError для обработки исключения и вывода сообщения об ошибке в консоль.



```
1
2
3
4 void main() {
5   Future.delayed(Duration(seconds: 2), () => throw Exception("Ошибка!"))
6     .catchError((error) => print(error));
7 }
```

Exception: Ошибка!
DartPad caught unhandled ArgumentError:
Invalid argument(s) (onError): The error hand
https://storage.googleapis.com/nnbd_artifac
https://storage.googleapis.com/nnbd_artifac
https://storage.googleapis.com/nnbd_artifac

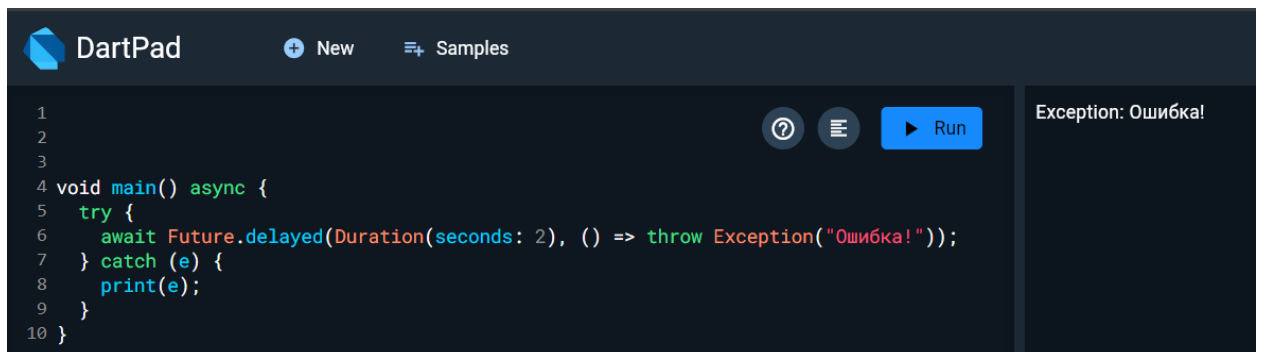
Рисунок №3 – Результат решения задания №3.

Задание № 4: Использование async/await с обработкой ошибок

Перепишите предыдущее задание с использованием async/await и блока try-catch.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await для ожидания завершения Future.
3. Используйте блок try-catch для обработки исключения и вывода сообщения об ошибке в консоль.



```
1
2
3
4 void main() async {
5   try {
6     await Future.delayed(Duration(seconds: 2), () => throw Exception("Ошибка!"));
7   } catch (e) {
8     print(e);
9   }
10 }
```

Exception: Ошибка!

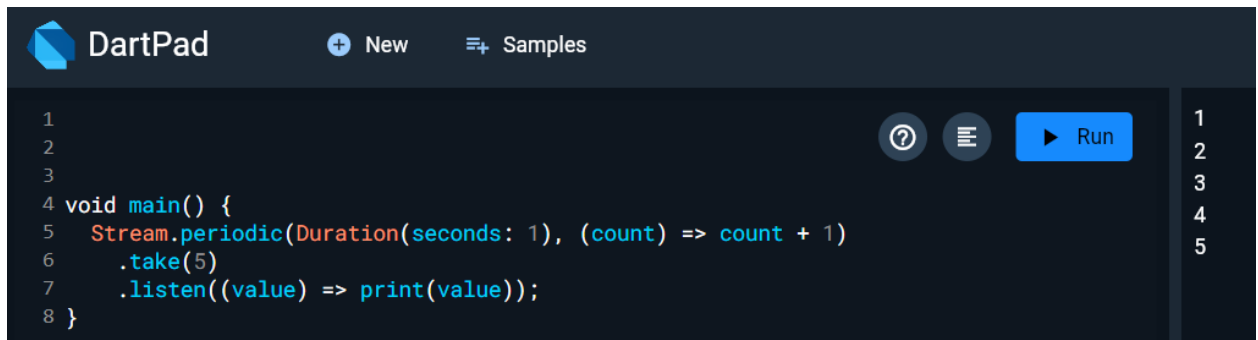
Рисунок №4 – Результат решения задания №4.

Задание № 5: Простой Stream

Создайте Stream, который каждую секунду генерирует число от 1 до 5. Выведите каждое число в консоль.

Описание алгоритма решения:

1. Создайте Stream с помощью конструктора Stream.periodic.
2. Передайте ему длительность интервала (1 секунда) и анонимную функцию, которая генерирует число.
3. Используйте take для ограничения количества элементов в потоке.
4. Используйте listen для вывода каждого элемента в консоль.



```
1
2
3
4 void main() {
5   Stream.periodic(Duration(seconds: 1), (count) => count + 1)
6     .take(5)
7     .listen((value) => print(value));
8 }
```

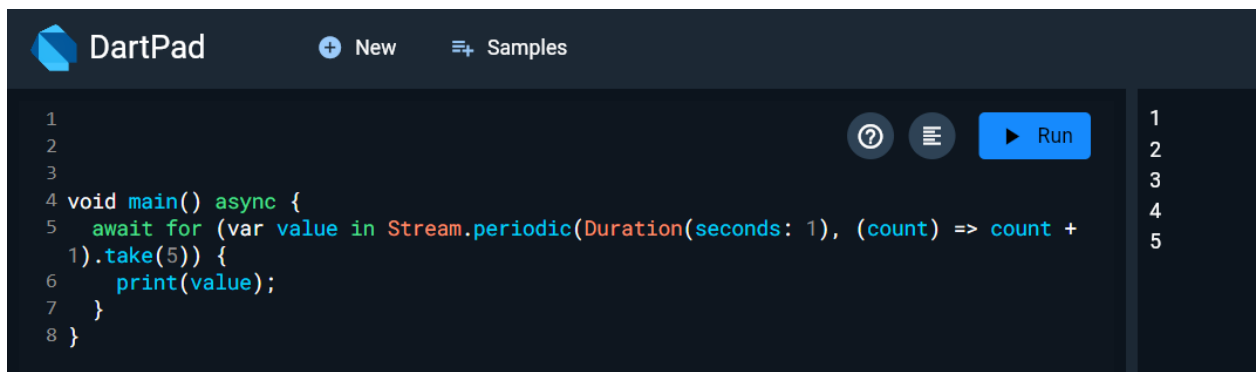
Рисунок №5 – Результат решения задания №5.

Задание № 6: Использование async/await с Stream

Перепишите предыдущее задание с использованием async/await и цикла for-in.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await for для итерации по элементам Stream.
3. Выведите каждый элемент в консоль.



```
1
2
3
4 void main() async {
5   await for (var value in Stream.periodic(Duration(seconds: 1), (count) => count +
6     1).take(5)) {
7     print(value);
8   }
9 }
```

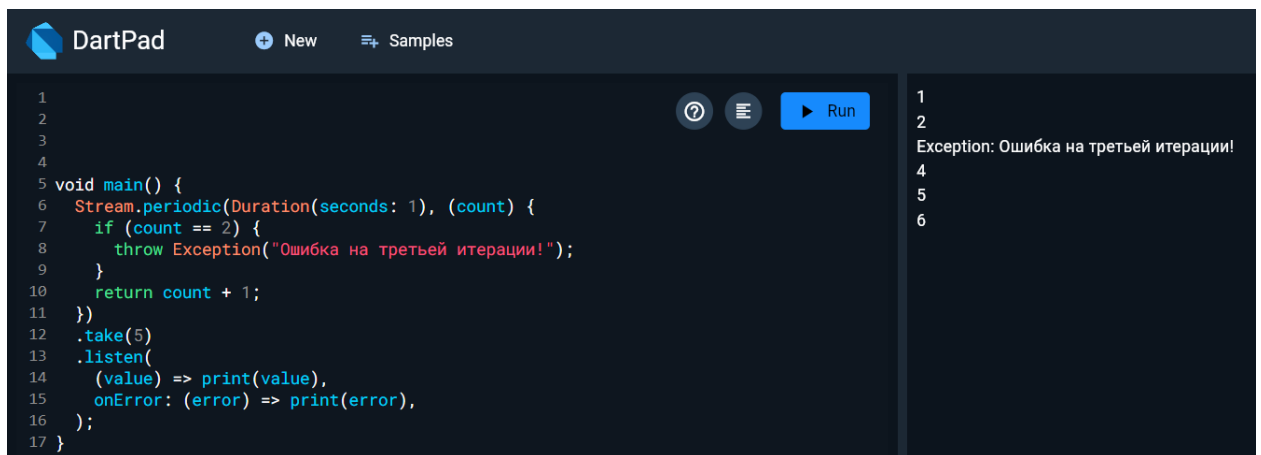
Рисунок №6 – Результат решения задания №6.

Задание № 7: Обработка ошибок в Stream

Создайте Stream, который каждую секунду генерирует число от 1 до 5, но на третьей итерации генерирует исключение. Обработайте это исключение и выведите сообщение об ошибке в консоль.

Описание алгоритма решения:

1. Создайте Stream с помощью конструктора Stream.periodic.
2. Передайте ему длительность интервала (1 секунда) и анонимную функцию, которая генерирует число или исключение.
3. Используйте take для ограничения количества элементов в потоке.
4. Используйте listen для вывода каждого элемента в консоль и обработки исключения.



The image shows a screenshot of the DartPad web IDE. The left pane contains the following Dart code:

```
1
2
3
4
5 void main() {
6   Stream.periodic(Duration(seconds: 1), (count) {
7     if (count == 2) {
8       throw Exception("Ошибка на третьей итерации!");
9     }
10    return count + 1;
11  })
12  .take(5)
13  .listen(
14    (value) => print(value),
15    onError: (error) => print(error),
16  );
17 }
```

The right pane shows the output of the code execution:

```
1
2
3 Exception: Ошибка на третьей итерации!
4
5
6
```

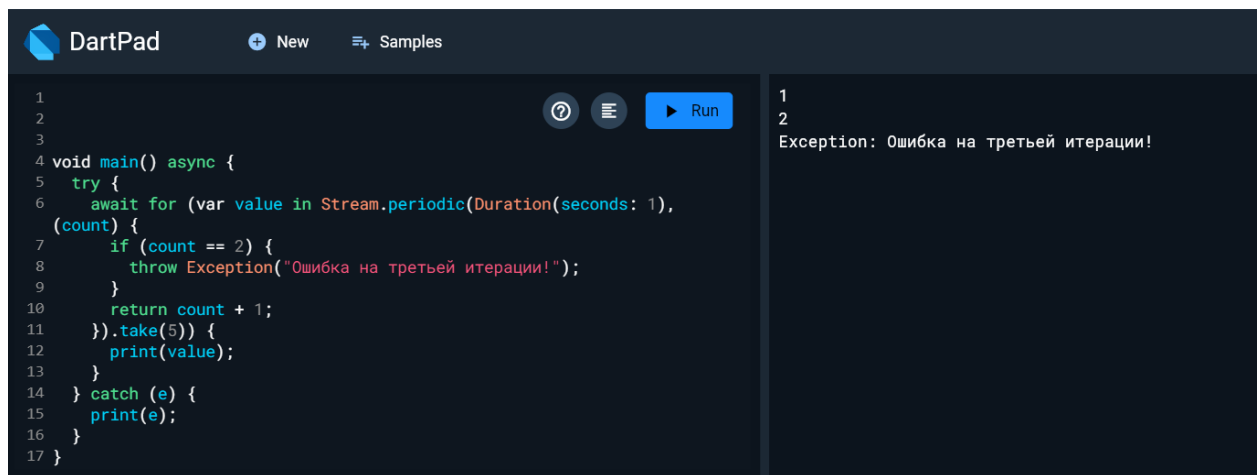
Рисунок №7 – Результат решения задания №7.

Задание № 8: Использование `async/await` с обработкой ошибок в `Stream`

Перепишите предыдущее задание с использованием `async/await` и блока `try-catch`.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова `async`.
2. Внутри функции используйте `await for` для итерации по элементам `Stream`.
3. Используйте блок `try-catch` для обработки исключения и вывода сообщения об ошибке в консоль.



The screenshot shows the DartPad interface. On the left, the code editor contains the following Dart code:

```
1
2
3
4 void main() async {
5   try {
6     await for (var value in Stream.periodic(Duration(seconds: 1),
7       (count) {
8         if (count == 2) {
9           throw Exception("Ошибка на третьей итерации!");
10        }
11        return count + 1;
12      }).take(5)) {
13       print(value);
14     }
15   } catch (e) {
16     print(e);
17   }
18 }
```

On the right, the console output shows:

```
1
2
Exception: Ошибка на третьей итерации!
```

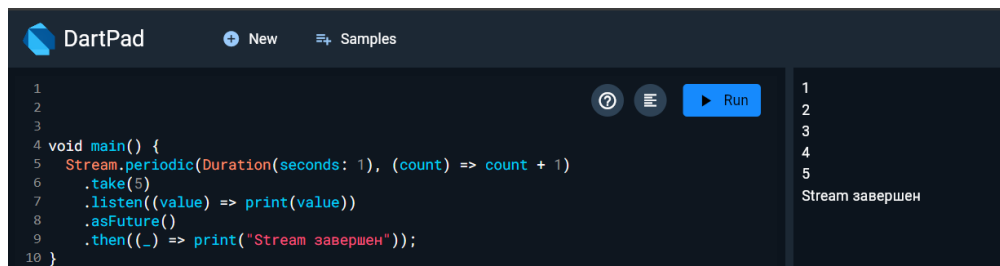
Рисунок №8 – Результат решения задания №8.

Задание № 9: Создание Future из Stream

Создайте Stream, который каждую секунду генерирует число от 1 до 5. Преобразуйте этот Stream в Future, который завершается, когда Stream завершается. Выведите сообщение «Stream завершен» в консоль.

Описание алгоритма решения:

1. Создайте Stream с помощью конструктора Stream.periodic.
2. Передайте ему длительность интервала (1 секунда) и анонимную функцию, которая генерирует число.
3. Используйте take для ограничения количества элементов в потоке.
4. Используйте listen для вывода каждого элемента в консоль.
5. Используйте asFuture для преобразования Stream в Future.
6. Используйте then для вывода сообщения «Stream завершен» в консоль после завершения Stream.



```
1
2
3
4 void main() {
5   Stream.periodic(Duration(seconds: 1), (count) => count + 1)
6     .take(5)
7     .listen((value) => print(value))
8     .asFuture()
9     .then((_) => print("Stream завершен"));
10 }
```

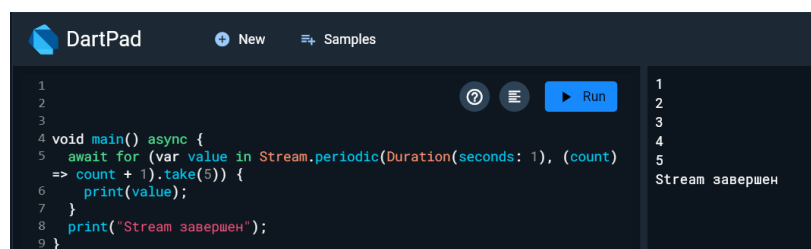
Рисунок №9 – Результат решения задания №9.

Задание № 10: Использование async/await с Future из Stream

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await for для итерации по элементам Stream.
3. Выведите каждый элемент в консоль.
4. Используйте await для ожидания завершения Stream.
5. Выведите сообщение «Stream завершен» в консоль.



```
1
2
3
4 void main() async {
5   await for (var value in Stream.periodic(Duration(seconds: 1), (count)
6     => count + 1).take(5)) {
7     print(value);
8   }
9   print("Stream завершен");
10 }
```

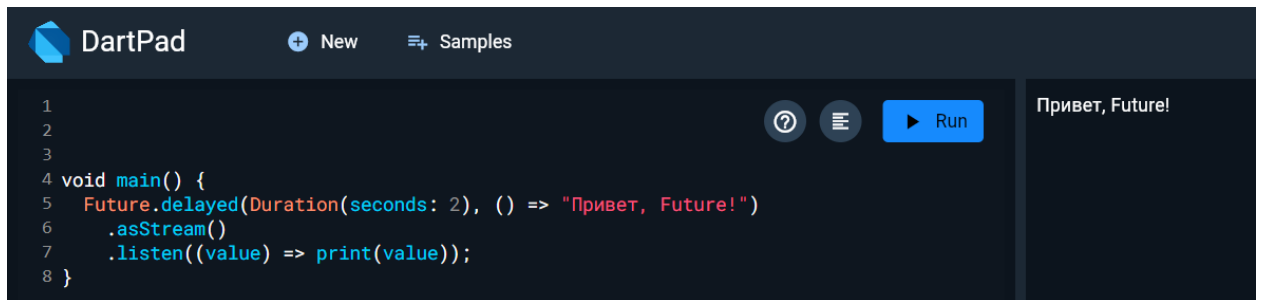
Рисунок №10 – Результат решения задания №10.

Задание № 11: Создание Stream из Future

Создайте Future, который через 2 секунды возвращает строку «Привет, Future!». Преобразуйте этот Future в Stream и выведите результат в консоль.

Описание алгоритма решения:

1. Создайте Future с помощью конструктора *Future.delayed*.
2. Передайте ему длительность задержки (2 секунды) и анонимную функцию, которая возвращает строку «Привет, Future!».
3. Используйте *asStream* для преобразования Future в Stream.
4. Используйте *listen* для вывода результата в консоль.



```
1
2
3
4 void main() {
5   Future.delayed(Duration(seconds: 2), () => "Привет, Future!")
6     .asStream()
7     .listen((value) => print(value));
8 }
```

Привет, Future!

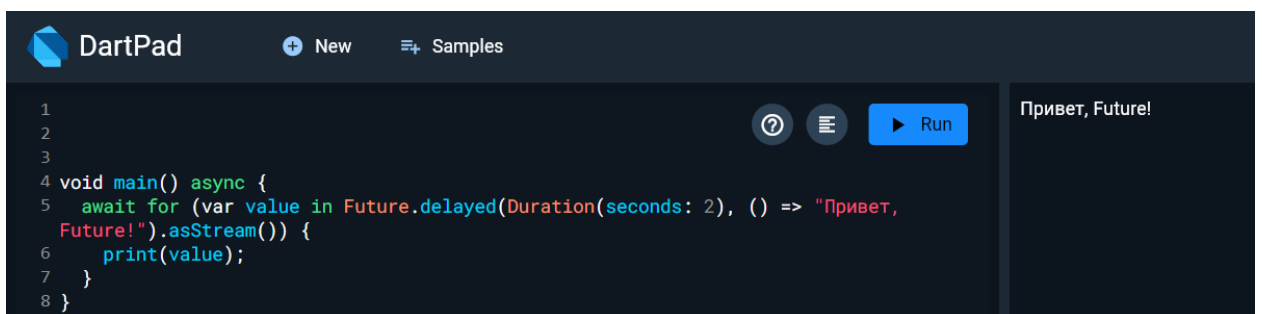
Рисунок №11 – Результат решения задания №11.

Задание № 12: Использование async/await с Stream из Future

Перепишите предыдущее задание с использованием *async/await*.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова *async*.
2. Внутри функции используйте *await for* для итерации по элементам *Stream*.
3. Выведите результат в консоль



```
1
2
3
4 void main() async {
5   await for (var value in Future.delayed(Duration(seconds: 2), () => "Привет,
6     Future!").asStream()) {
7     print(value);
8   }
9 }
```

Привет, Future!

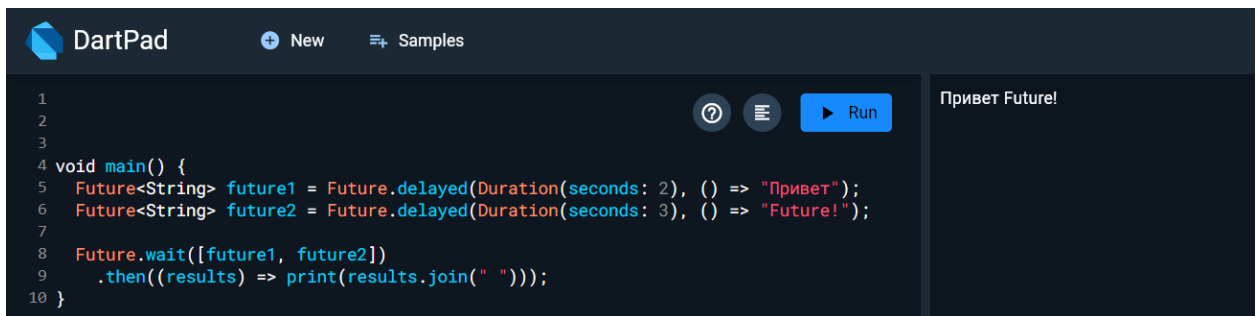
Рисунок №12 – Результат решения задания №12.

Задание № 13: Объединение двух Future

Создайте два Future: первый возвращает строку «Привет» через 2 секунды, а второй возвращает строку «Future!» через 3 секунды. Объедините результаты этих Future в одну строку и выведите её в консоль.

Описание алгоритма решения:

1. Создайте два Future с помощью конструктора Future.delayed.
2. Используйте Future.wait для ожидания завершения обоих Future.
3. Объедините результаты в одну строку и выведите её в консоль.



```
1
2
3
4 void main() {
5   Future<String> future1 = Future.delayed(Duration(seconds: 2), () => "Привет");
6   Future<String> future2 = Future.delayed(Duration(seconds: 3), () => "Future!");
7
8   Future.wait([future1, future2])
9     .then((results) => print(results.join(" ")));
10 }
```

Привет Future!

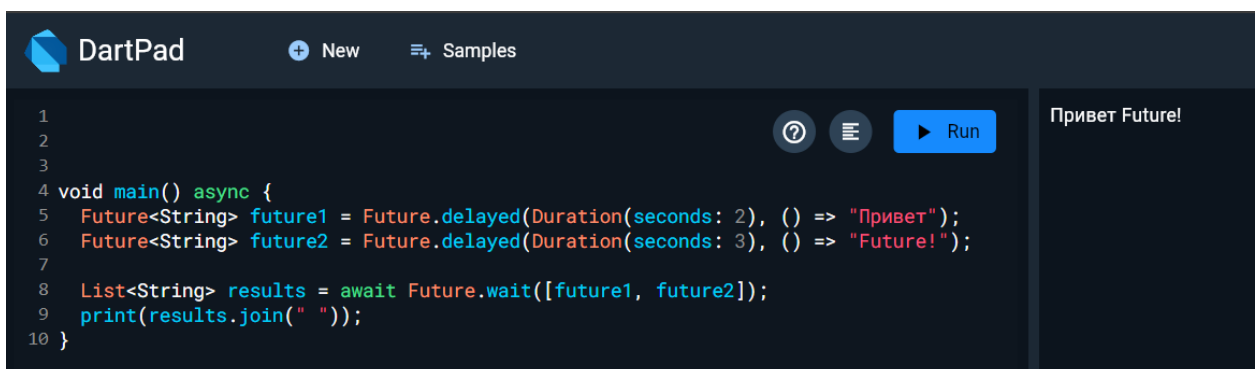
Рисунок №13 – Результат решения задания №13.

Задание № 14: Использование async/await с объединением Future

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await для ожидания завершения обоих Future.
3. Объедините результаты в одну строку и выведите её в консоль.



```
1
2
3
4 void main() async {
5   Future<String> future1 = Future.delayed(Duration(seconds: 2), () => "Привет");
6   Future<String> future2 = Future.delayed(Duration(seconds: 3), () => "Future!");
7
8   List<String> results = await Future.wait([future1, future2]);
9   print(results.join(" "));
10 }
```

Привет Future!

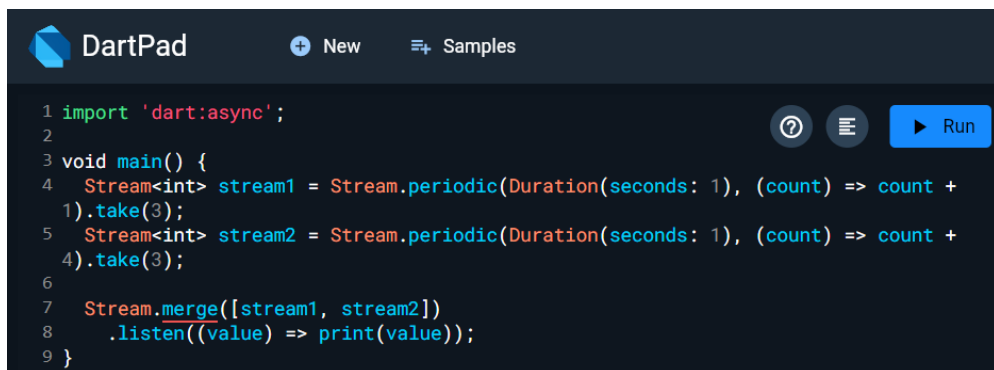
Рисунок №14 – Результат решения задания №14.

Задание № 15: Объединение двух Stream.

Создайте два Stream: первый генерирует числа от 1 до 3 с интервалом в 1 секунду, а второй генерирует числа от 4 до 6 с интервалом в 1 секунду. Объедините эти Stream в один и выведите все числа в консоль.

Описание алгоритма решения:

1. Создайте два Stream с помощью конструктора Stream.periodic.
2. Используйте take для ограничения количества элементов в каждом Stream.
3. Используйте Stream.merge для объединения двух Stream.
4. Используйте listen для вывода всех элементов.



```
1 import 'dart:async';
2
3 void main() {
4   Stream<int> stream1 = Stream.periodic(Duration(seconds: 1), (count) => count +
5   1).take(3);
6   Stream<int> stream2 = Stream.periodic(Duration(seconds: 1), (count) => count +
7   4).take(3);
8   Stream.merge([stream1, stream2])
9     .listen((value) => print(value));
10 }
11
12 }
```

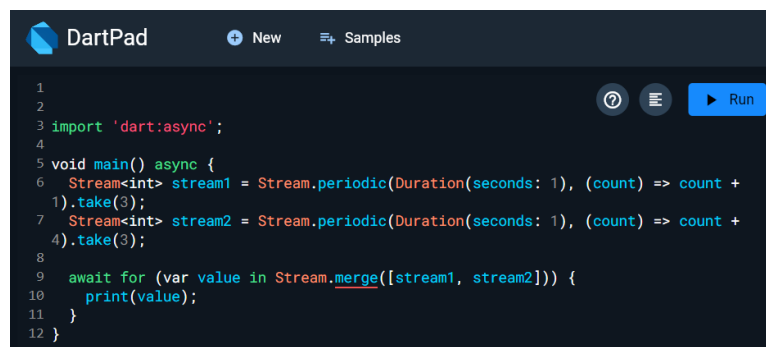
Рисунок №15 – Результат решения задания №15.

Задание № 16: Использование async/await с объединением Stream

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await for для итерации по элементам объединенного Stream.



```
1
2
3 import 'dart:async';
4
5 void main() async {
6   Stream<int> stream1 = Stream.periodic(Duration(seconds: 1), (count) => count +
7   1).take(3);
8   Stream<int> stream2 = Stream.periodic(Duration(seconds: 1), (count) => count +
9   4).take(3);
10   await for (var value in Stream.merge([stream1, stream2])) {
11     print(value);
12   }
13 }
```

Рисунок №16 – Результат решения задания №16.

Задание № 17: Преобразование Stream с использованием map

Создайте Stream, который каждую секунду генерирует число от 1 до 5. Преобразуйте каждое число в строку, добавив к нему слово «число». Выведите результат в консоль.

Описание алгоритма решения:

1. Создайте Stream с помощью конструктора Stream.periodic.
2. Передайте ему длительность интервала (1 секунда) и анонимную функцию, которая генерирует число.
3. Используйте take для ограничения количества элементов в потоке.
4. Используйте map для преобразования каждого числа в строку.
5. Используйте listen для вывода результата в консоль.



```
1
2
3
4 void main() {
5   Stream.periodic(Duration(seconds: 1), (count) => count + 1)
6     .take(5)
7     .map((value) => "число $value")
8     .listen((value) => print(value));
9 }
```

число 1
число 2
число 3
число 4
число 5

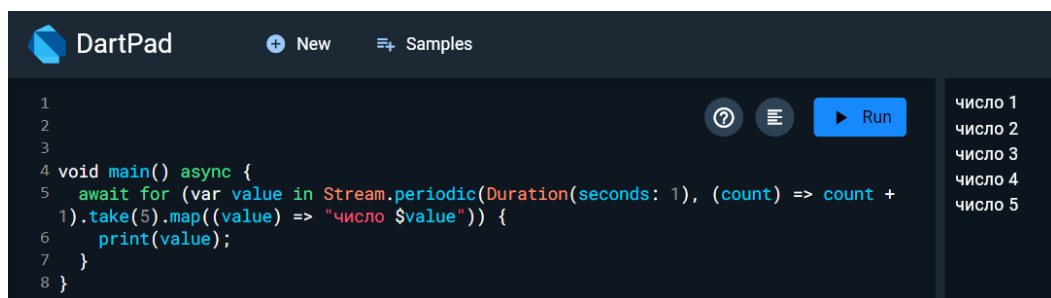
Рисунок №17 – Результат решения задания №17.

Задание № 18: Использование async/await с преобразованием Stream

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await for для итерации по элементам Stream.
3. Преобразуйте каждый элемент в строку.
4. Выведите результат в консоль



```
1
2
3
4 void main() async {
5   await for (var value in Stream.periodic(Duration(seconds: 1), (count) => count +
6     1).take(5).map((value) => "число $value")) {
7     print(value);
8   }
9 }
```

число 1
число 2
число 3
число 4
число 5

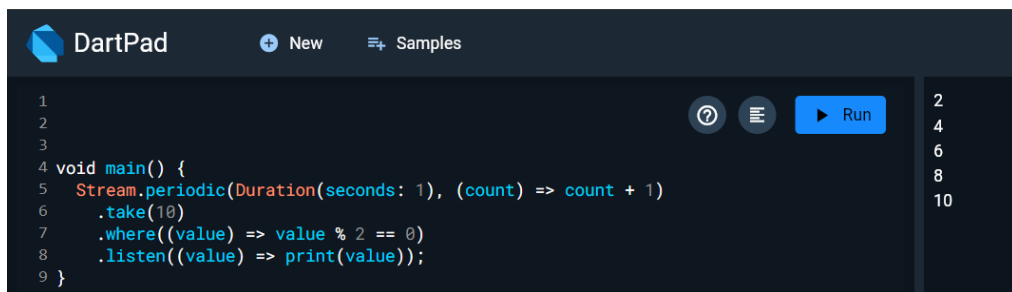
Рисунок №18 – Результат решения задания №18.

Задание № 19: Фильтрация Stream с использованием where

Создайте Stream, который каждую секунду генерирует число от 1 до 10. Отфильтруйте этот Stream, чтобы он содержал только четные числа. Выведите результат в консоль.

Описание алгоритма решения:

1. Создайте Stream с помощью конструктора Stream.periodic.
2. Передайте ему длительность интервала (1 секунда) и анонимную функцию, которая генерирует число.
3. Используйте take для ограничения количества элементов в потоке.
4. Используйте where для фильтрации только четных чисел.
5. Используйте listen для вывода результата в консоль



```
1
2
3
4 void main() {
5   Stream.periodic(Duration(seconds: 1), (count) => count + 1)
6     .take(10)
7     .where((value) => value % 2 == 0)
8     .listen((value) => print(value));
9 }
```

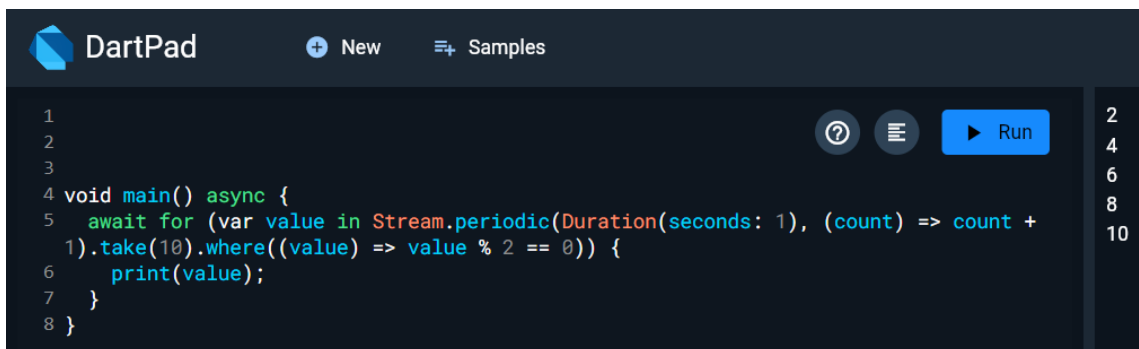
Рисунок №19 – Результат решения задания №19.

Задание № 20: Использование async/await с фильтрацией Stream

Перепишите предыдущее задание с использованием async/await.

Описание алгоритма решения:

1. Определите асинхронную функцию с помощью ключевого слова async.
2. Внутри функции используйте await for для итерации по элементам Stream.
3. Отфильтруйте только четные числа.
4. Выведите результат в консоль



```
1
2
3
4 void main() async {
5   await for (var value in Stream.periodic(Duration(seconds: 1), (count) => count +
6     1).take(10).where((value) => value % 2 == 0)) {
7     print(value);
8   }
9 }
```

Рисунок №20 – Результат решения задания №20.

Индивидуальные задания для закрепления материала

Базовые задания:

1. **Асинхронное чтение файла:** Напишите программу, которая асинхронно читает содержимое текстового файла и выводит его на экран. Используйте Future и async/await.
2. **Асинхронная загрузка изображения:** Создайте приложение, которое асинхронно загружает изображение по URL и отображает его на экране. Используйте Future и async/await.
3. **Асинхронная сортировка списка:** Напишите программу, которая асинхронно сортирует список чисел и выводит отсортированный список на экран. Используйте Future и async/await

Средний уровень:

1. **Асинхронная загрузка данных с API:** Напишите программу, которая асинхронно загружает данные с API (например, JSONPlaceholder) и выводит их на экран. Используйте Future и async/await.
2. **Асинхронное выполнение нескольких задач:** Напишите программу, которая асинхронно выполняет несколько задач (например, загрузка данных с разных API) и выводит результаты на экран. Используйте Future.wait.
3. **Асинхронная обработка потока данных:** Напишите программу, которая асинхронно обрабатывает поток данных (например, поток чисел) и выводит их на экран. Используйте Stream и async/await.

Продвинутый уровень:

1. **Асинхронная загрузка и обработка изображений:** Напишите программу, которая асинхронно загружает несколько изображений по URL, обрабатывает их (например, изменяет размер) и отображает на экране. Используйте Future, async/await и Canvas.
2. **Асинхронная загрузка и отображение данных в таблице:** Напишите программу, которая асинхронно загружает данные с API и отображает их в таблице на экране. Используйте Future, async/await и TableElement.
3. **Асинхронная загрузка и отображение данных в виде списка:** Напишите программу, которая асинхронно загружает данные с API и отображает их в виде списка на экране. Используйте Future, async/await и UListElement.
4. **Асинхронная загрузка и отображение данных в виде карточек:** Напишите программу, которая асинхронно загружает

данные с API и отображает их в виде карточек на экране. Используйте Future, async/await и DivElement.

5. **Асинхронная загрузка и отображение данных в виде диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде диаграммы на экране. Используйте Future, async/await и Canvas.
6. **Асинхронная загрузка и отображение данных в виде графика**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде графика на экране. Используйте Future, async/await и Canvas.
7. **Асинхронная загрузка и отображение данных в виде круговой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде круговой диаграммы на экране. Используйте Future, async/await и Canvas.
8. **Асинхронная загрузка и отображение данных в виде столбчатой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде столбчатой диаграммы на экране. Используйте Future, async/await и Canvas.
9. **Асинхронная загрузка и отображение данных в виде линейной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде линейной диаграммы на экране. Используйте Future, async/await и Canvas.
10. **Асинхронная загрузка и отображение данных в виде точечной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде точечной диаграммы на экране. Используйте Future, async/await и Canvas.
11. **Асинхронная загрузка и отображение данных в виде круговой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде круговой диаграммы на экране. Используйте Future, async/await и Canvas.
12. **Асинхронная загрузка и отображение данных в виде столбчатой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде столбчатой диаграммы на экране. Используйте Future, async/await и Canvas.
13. **Асинхронная загрузка и отображение данных в виде линейной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде линейной диаграммы на экране. Используйте Future, async/await и Canvas.

14. **Асинхронная загрузка и отображение данных в виде точечной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде точечной диаграммы на экране. Используйте Future, async/await и Canvas.
15. **Асинхронная загрузка и отображение данных в виде круговой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде круговой диаграммы на экране. Используйте Future, async/await и Canvas.
16. **Асинхронная загрузка и отображение данных в виде столбчатой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде столбчатой диаграммы на экране. Используйте Future, async/await и Canvas.
17. **Асинхронная загрузка и отображение данных в виде линейной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде линейной диаграммы на экране. Используйте Future, async/await и Canvas.
18. **Асинхронная загрузка и отображение данных в виде точечной диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде точечной диаграммы на экране. Используйте Future, async/await и Canvas.
19. **Асинхронная загрузка и отображение данных в виде круговой диаграммы**: Напишите программу, которая асинхронно загружает данные с API и отображает их в виде круговой диаграммы на экране. Используйте Future, async/await и Canvas.