Asynchronous Programming and Collections in Dart

A Comprehensive Guide

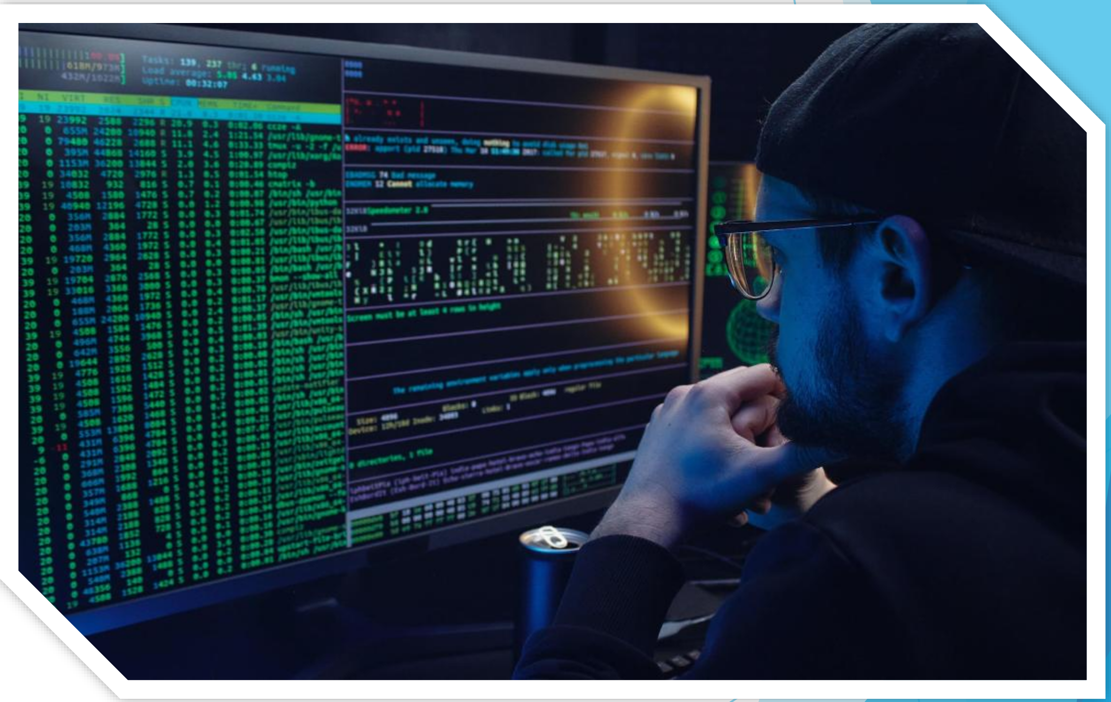
Introduction

Asynchronous Programming:

- Asynchronous programming is crucial for responsive apps.
- Dart offers powerful async features like Futures and Streams.

Collections in Dart:

- Collections are essential for managing data.
- Dart supports various collections like lists, sets, maps, and queues.

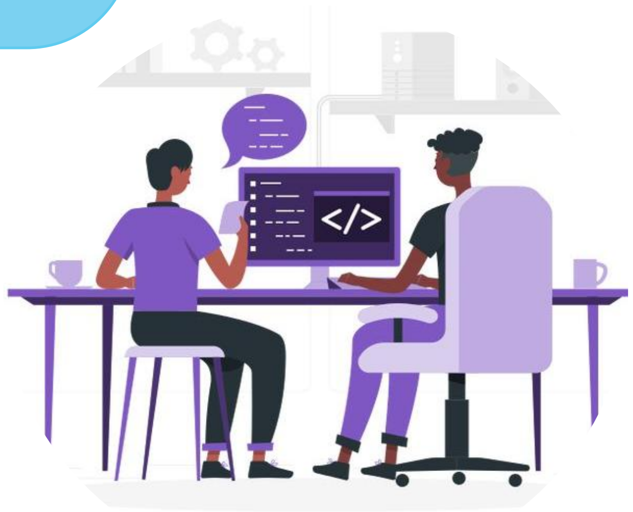


Aitrich Academy

Asynchronous Programming in Dart

Futures

- Represent an asynchronous operation that eventually completes.
- Handle the result of the operation with .then or await.
- Use await to wait for the result to be available.



Streams

- Represent sequence of data elements received asynchronously.
- Listen to the stream of data using methods like listen.
- Use async for loop to iterate over the stream elements.

Example of Future

1. Asynchronous Function:

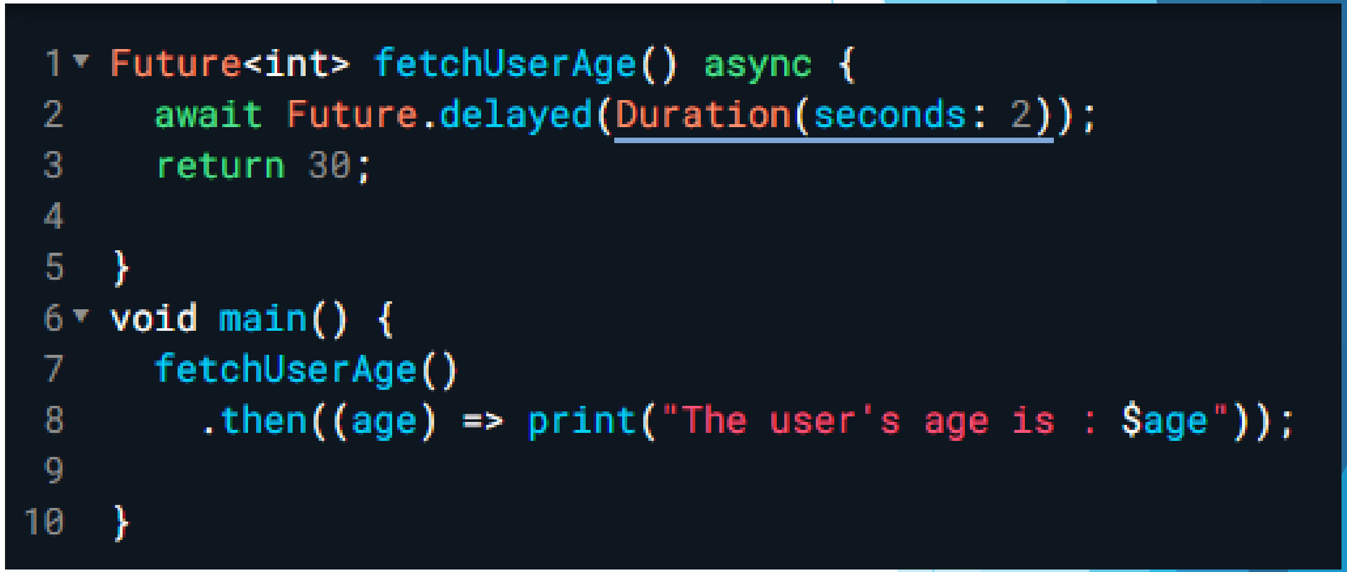
- Fetch User Age() is an asynchronous function that will eventually return an integer (user's age).

- It uses the async keyword to handle time-consuming operations without blocking the code's flow.

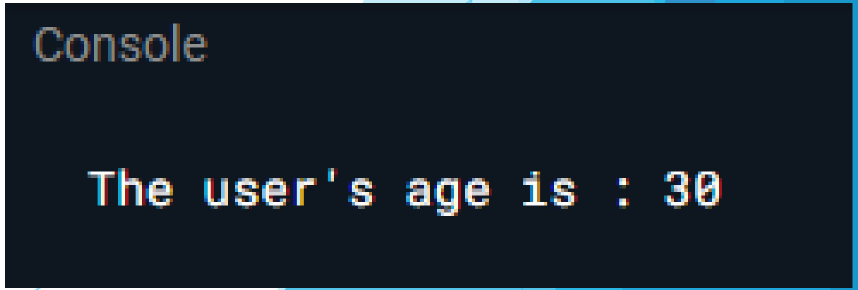
2. Simulating a Network Request:

- await Future .delayed(Duration(seconds: 2)) creates a 2-second delay, acting as a placeholder for a network request.

- The await keyword pauses the function's execution until the delay (or actual network request) completes.



```dart
Future<int> fetchUserAge() async {
  await Future.delayed(Duration(seconds: 2));
  return 30;

}
void main() {
  fetchUserAge()
    .then((age) => print("The user's age is : $age"));

}
```



```
The user's age is : 30
```

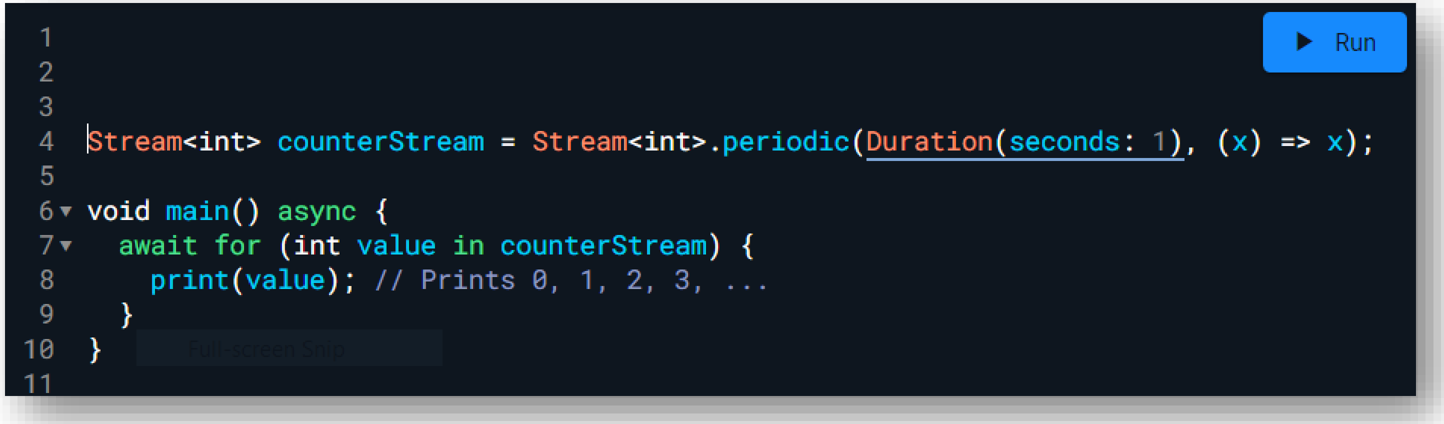
Example of Stream

1. Creating a Stream:

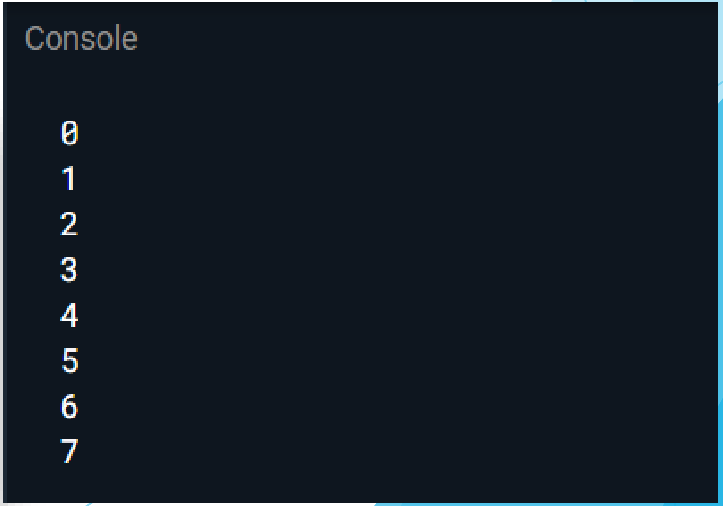
- Stream<int> counter Stream = Stream<int>.periodic(Duration(seconds: 1), (x) => x);
- Creates a Stream called counter Stream that emits an integer every second.
- Stream<int> specifies that the Stream produces integer values.

2. Asynchronously Iterating over the Stream:

- await for (int value in counter Stream) { ... }
- Uses an await for loop to iterate over the elements of the Stream asynchronously.
- await for pauses execution until the next value is available from the Stream.



```dart



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

void main() async {
  await for (int value in counterStream) {
    print(value); // Prints 0, 1, 2, 3, ...
  }
}

```



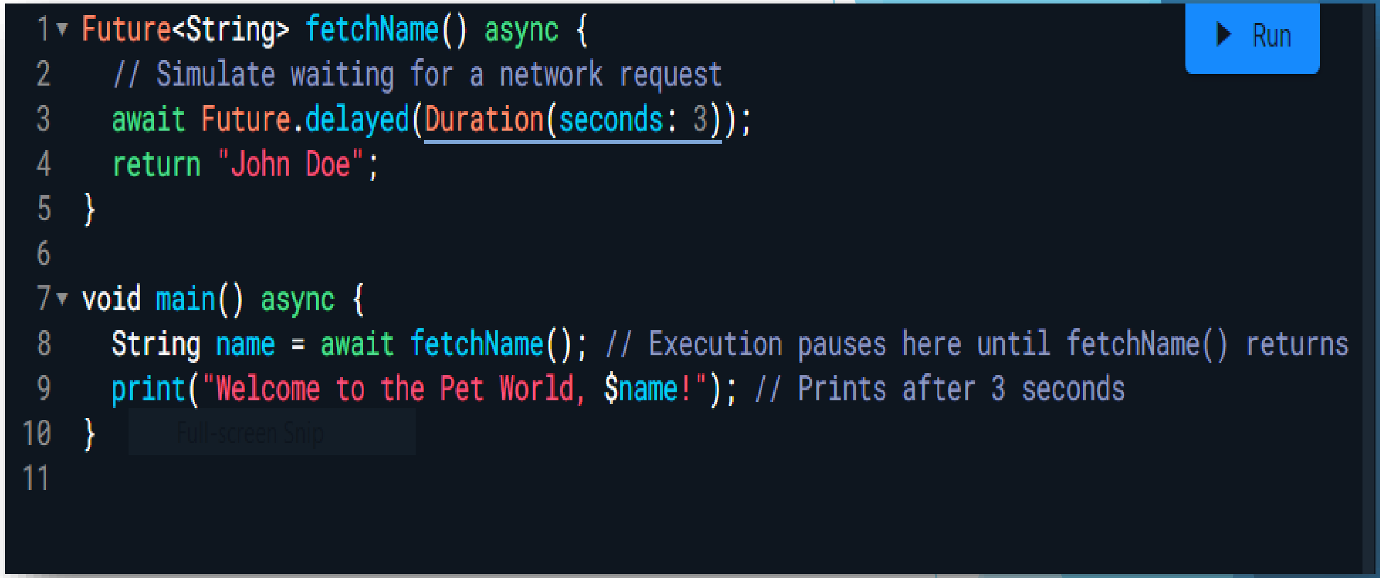
```
0
1
2
3
4
5
6
7
```

Async and Await

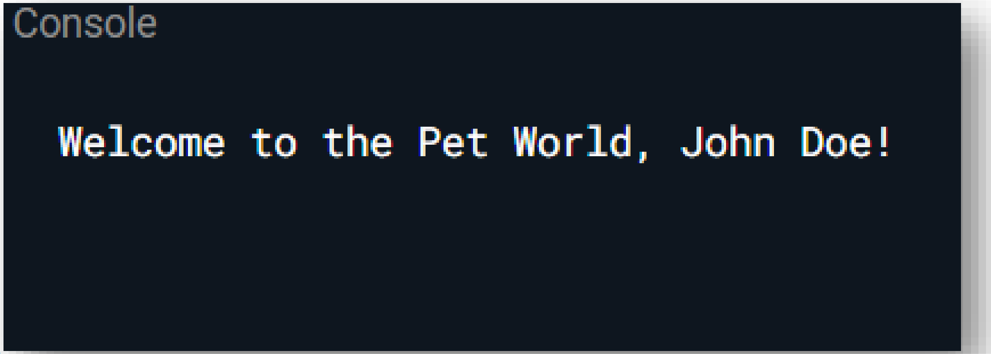
1.async:

- Used to **mark a function** as asynchronous. This means the function might take some time to complete because it may be waiting for an external event, like a network request or user input.

- When an async function is called, it immediately returns a Future object that represents the eventual result of the function.



```dart
Future<String> fetchName() async {
  // Simulate waiting for a network request
  await Future.delayed(Duration(seconds: 3));
  return "John Doe";
}

void main() async {
  String name = await fetchName(); // Execution pauses here until fetchName() returns
  print("Welcome to the Pet World, $name!"); // Prints after 3 seconds
}

```



```
Welcome to the Pet World, John Doe!
```

2. await:

- Used **within an async function** to **pause the execution** of the function until the awaited value is ready.

- The awaited value can be anything that represents an asynchronous operation, such as another Future or a stream.

- While the awaited value is being retrieved, the rest of the code in the async function **doesn't wait**.

Aitrich Academy

Collections in Dart

- Like any other programming languages dart doesn't support arrays
- Dart collections can be used as data structures like an array.
- A collection is an object that represents a group of objects called elements.
- Iterables are a kind of collection.
- A collection can be empty, or it can contain many elements.
- Depending on the purpose, collections can have different structures and implementations.

Dart collections can be basically classified as:

- List
- Set
- Map
- Queue

Aitrich Academy

Lists

Lists

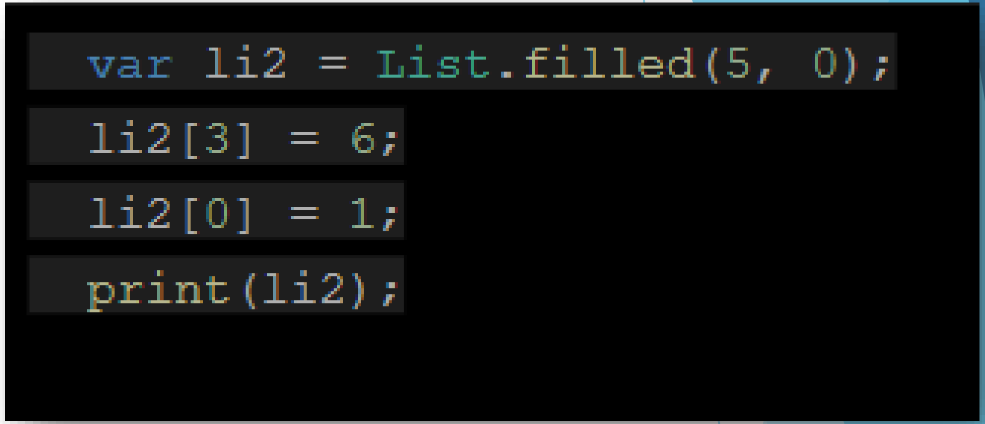
- Lists are ordered collections of elements that can be accessed by index.
- The List class provides methods for adding, removing, and searching for elements.

List in dart can be classified as:

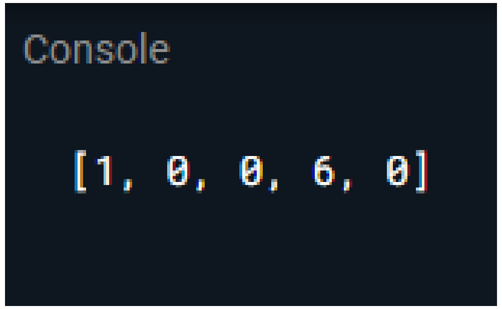
- **Fixed Length List**
- **Growable List**

List in dart can be classified as:

- **Fixed Length List**
- In Fixed Length List the list’s length cannot be changed at run-time.
- These list’s are defined with a specific length.



```
var li2 = List.filled(5, 0);
li2[3] = 6;
li2[0] = 1;
print(li2);
```

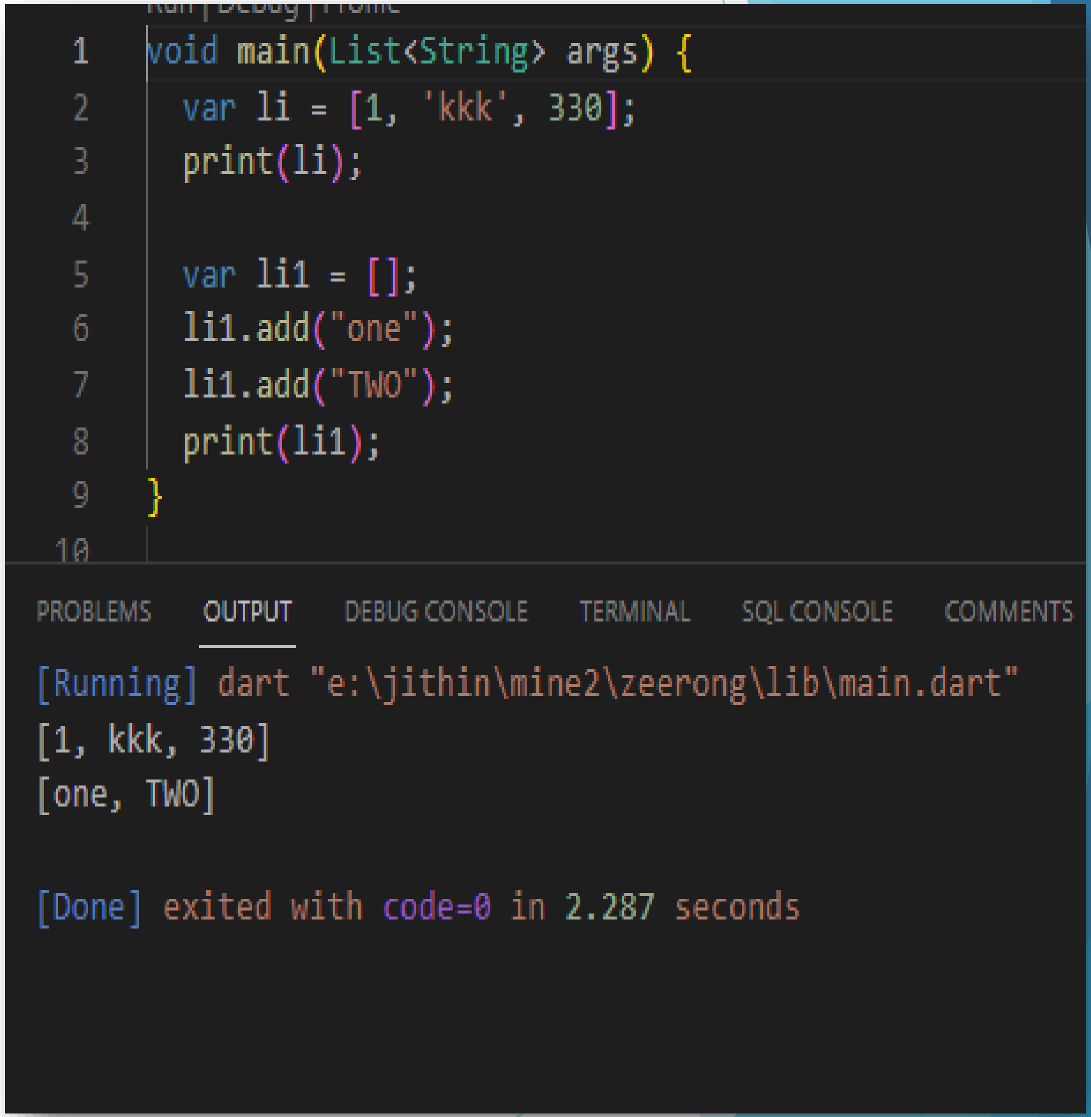


Console

```
[1, 0, 0, 6, 0]
```

- **Growable List**

- **Growable List**
- In Growable List the list’s length can be changed at run time.
- The following example shows how to create a list of 3 elements and another example which creates a zero-length list using the empty List() constructor.
- The add() function in the List class is used to dynamically add elements to the list.

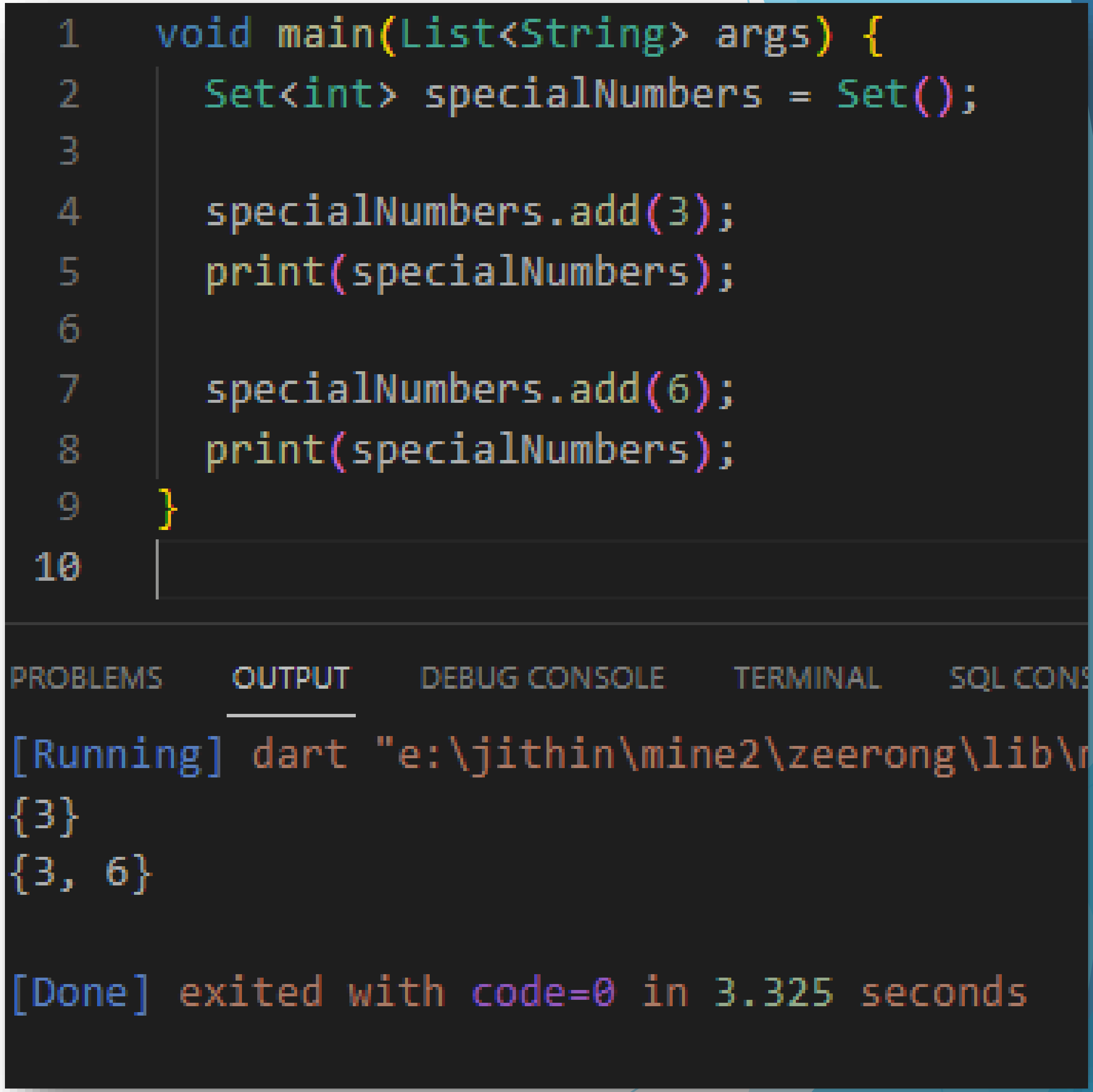


```dart
void main(List<String> args) {
  var li = [1, 'kkk', 330];
  print(li);

  var li1 = [];
  li1.add("one");
  li1.add("TWO");
  print(li1);
}
```

```
[Running] dart "e:\jithin\mine2\zeerong\lib\main.dart"
[1, kkk, 330]
[one, TWO]

[Done] exited with code=0 in 2.287 seconds
```

Set

- Set represents a collection of objects in which each object can occur only once.
- The dart :core library provides the Set class to implement the same.
- Must be useful when the program doesn't want a object to be added twice.



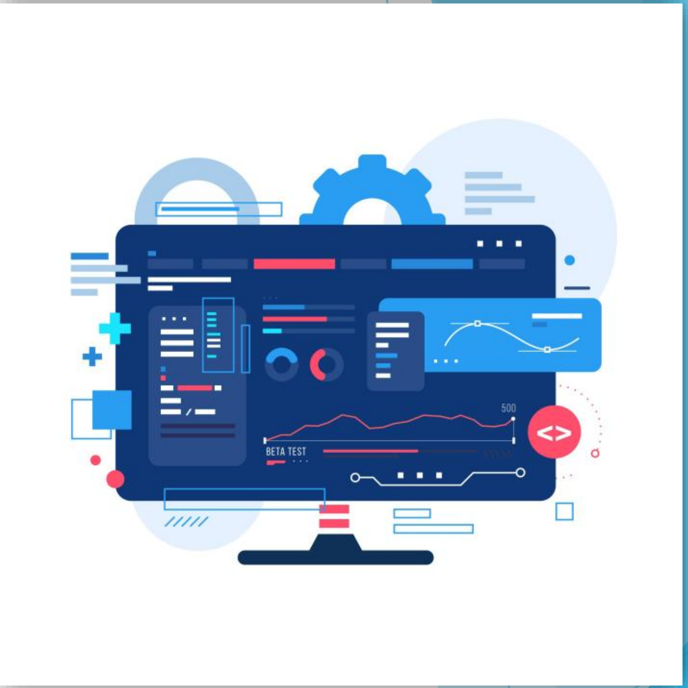
```dart
void main(List<String> args) {
  Set<int> specialNumbers = Set();

  specialNumbers.add(3);
  print(specialNumbers);

  specialNumbers.add(6);
  print(specialNumbers);
}

```

```
[Running] dart "e:\jithin\mine2\zeerong\lib\r
{3}
{3, 6}

[Done] exited with code=0 in 3.325 seconds
```

Maps

Maps

- Maps are collections of key-value pairs.
- The Map class provides methods for associating keys with values and retrieving values by key.



Maps can be declared in two ways :

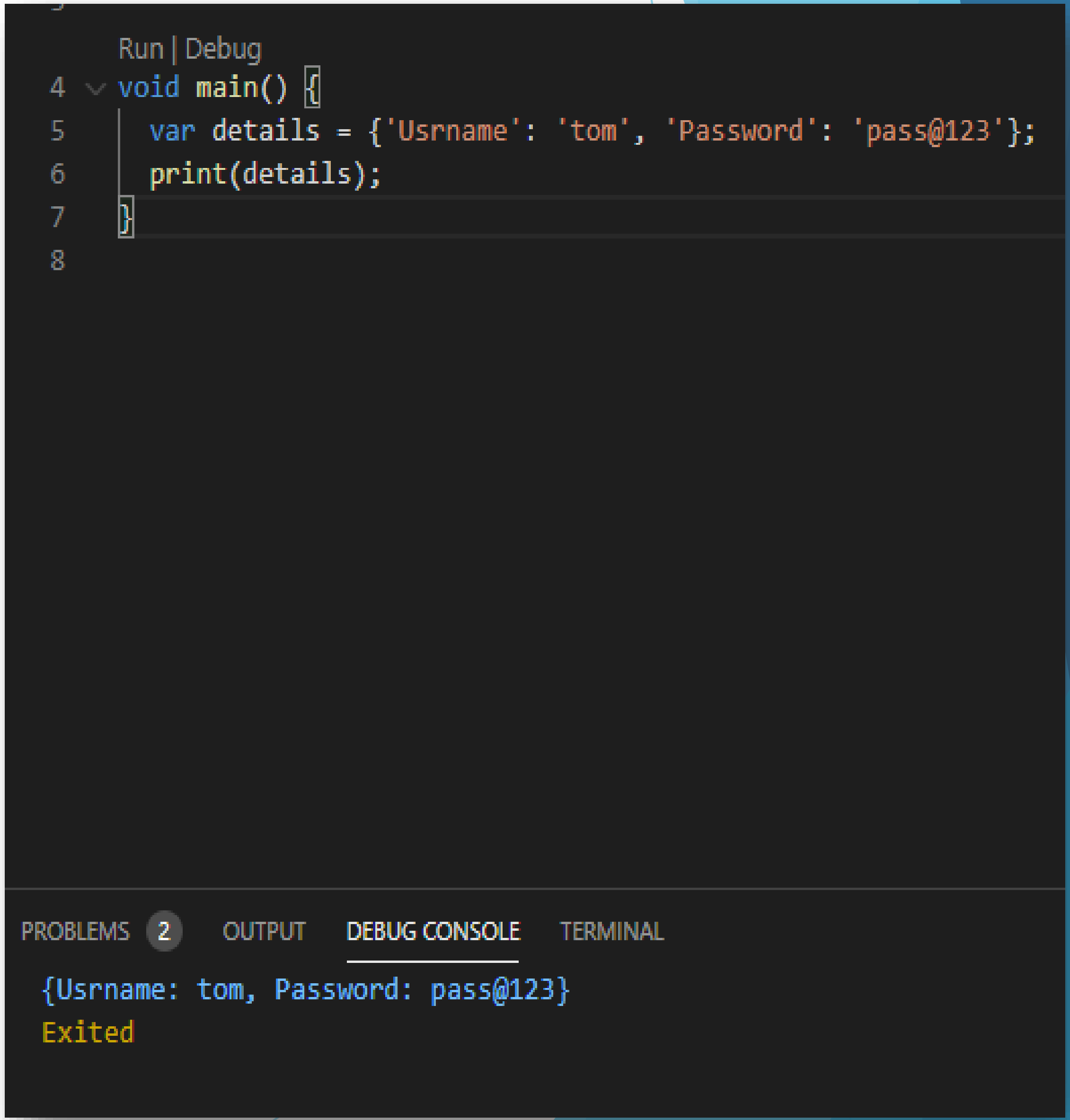
- **Using Map Literals**
- **Using a Map constructor**

Aitrich Academy

Maps can be declared in two ways :

Using Map Literal

- Just like we declare list using *var* keyword, we can also use var for declaring Maps.
- The main difference between declaration is, for declaring list we use [](square brackets), but to declare maps we have to use {}(curly braces).
- **for declaring list use [ ]**
- **for declaring map use {}**



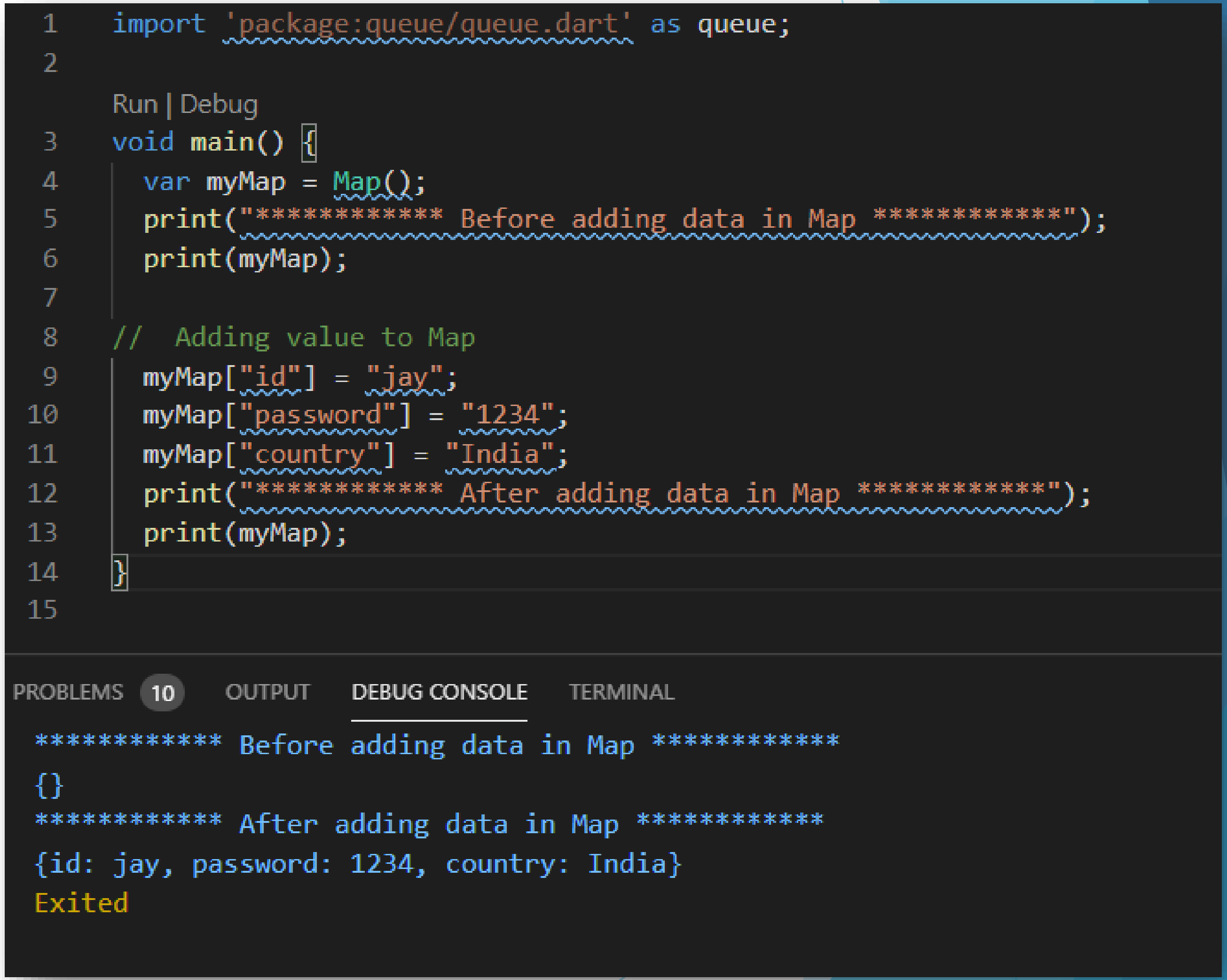
```
void main() {
  var details = {'Usrname': 'tom', 'Password': 'pass@123'};
  print(details);
}

```

```
{Usrname: tom, Password: pass@123}
Exited
```

Using Map Constructor

Using Map Constructor

- For declaring Map we can also use Map() constructor.
- It’s just which way you like.
- There nothing wrong if you declare map using standard method.

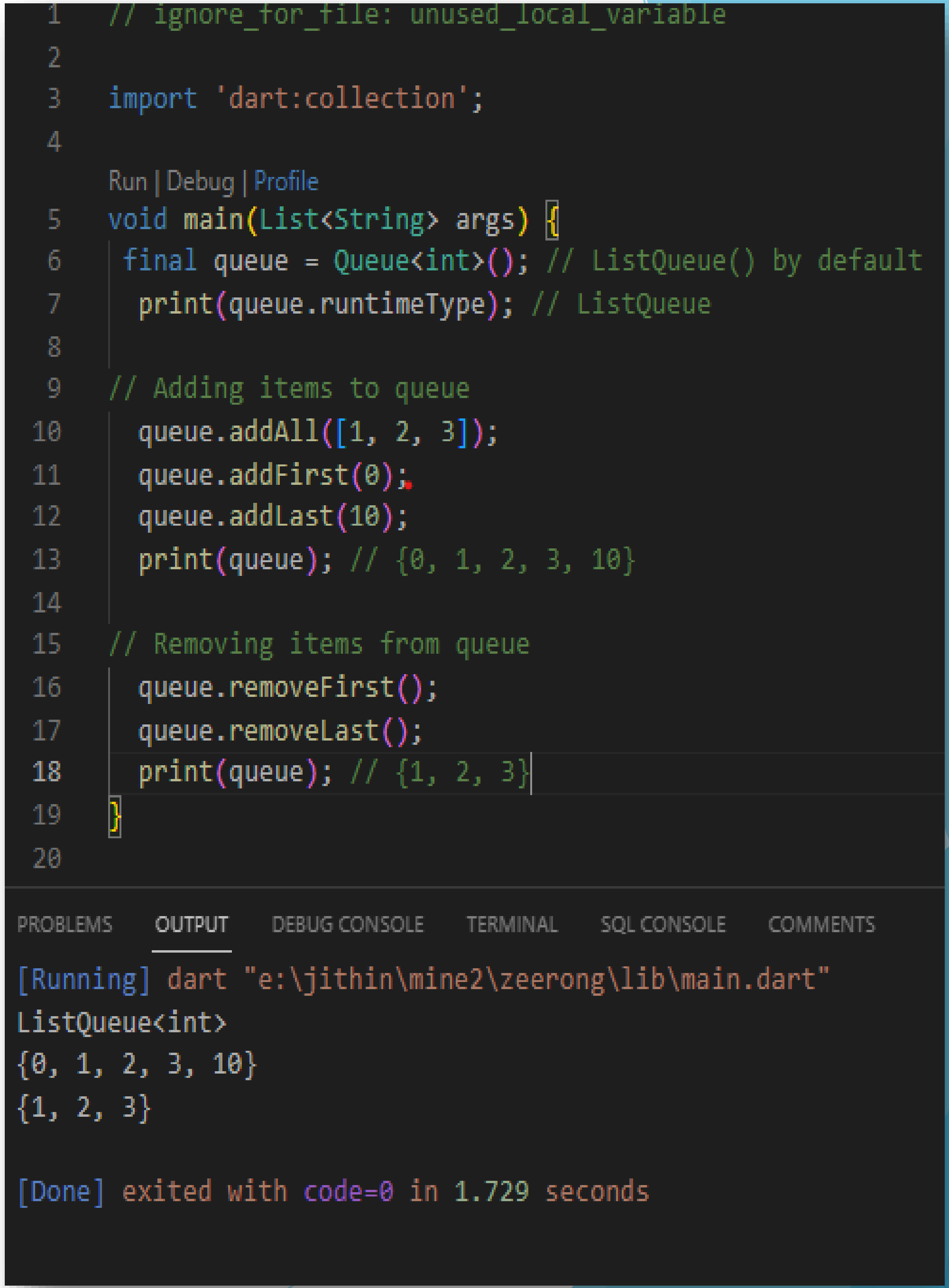


```dart
import 'package:queue/queue.dart' as queue;

void main() {
  var myMap = Map();
  print("************ Before adding data in Map ************");
  print(myMap);

//  Adding value to Map
  myMap["id"] = "jay";
  myMap["password"] = "1234";
  myMap["country"] = "India";
  print("************ After adding data in Map ************");
  print(myMap);
}
```

```
************ Before adding data in Map ************
{}
************ After adding data in Map ************
{id: jay, password: 1234, country: India}
Exited
```

Queue

- A Queue is a collection that can be manipulated at both ends.
- Queues are useful when you want to build a first-in, first-out (FIFO) collection.
- The values are removed / read in the order of their insertion.
- The add() function can be used to insert values to the queue.
- This function inserts the value specified to the end of the queue.



```dart
// ignore_for_file: unused_local_variable

import 'dart:collection';

void main(List<String> args) {
 final queue = Queue<int>(); // ListQueue() by default
  print(queue.runtimeType); // ListQueue

// Adding items to queue
  queue.addAll([1, 2, 3]);
  queue.addFirst(0);
  queue.addLast(10);
  print(queue); // {0, 1, 2, 3, 10}

// Removing items from queue
  queue.removeFirst();
  queue.removeLast();
  print(queue); // {1, 2, 3}
}

```

```
[Running] dart "e:\jithin\mine2\zeerong\lib\main.dart"
ListQueue<int>
{0, 1, 2, 3, 10}
{1, 2, 3}

[Done] exited with code=0 in 1.729 seconds
```

Generics in Dart

- Generics allow you to write code that is reusable and type-safe.

- Generic types are represented by type parameters that are declared with angle brackets (<>).

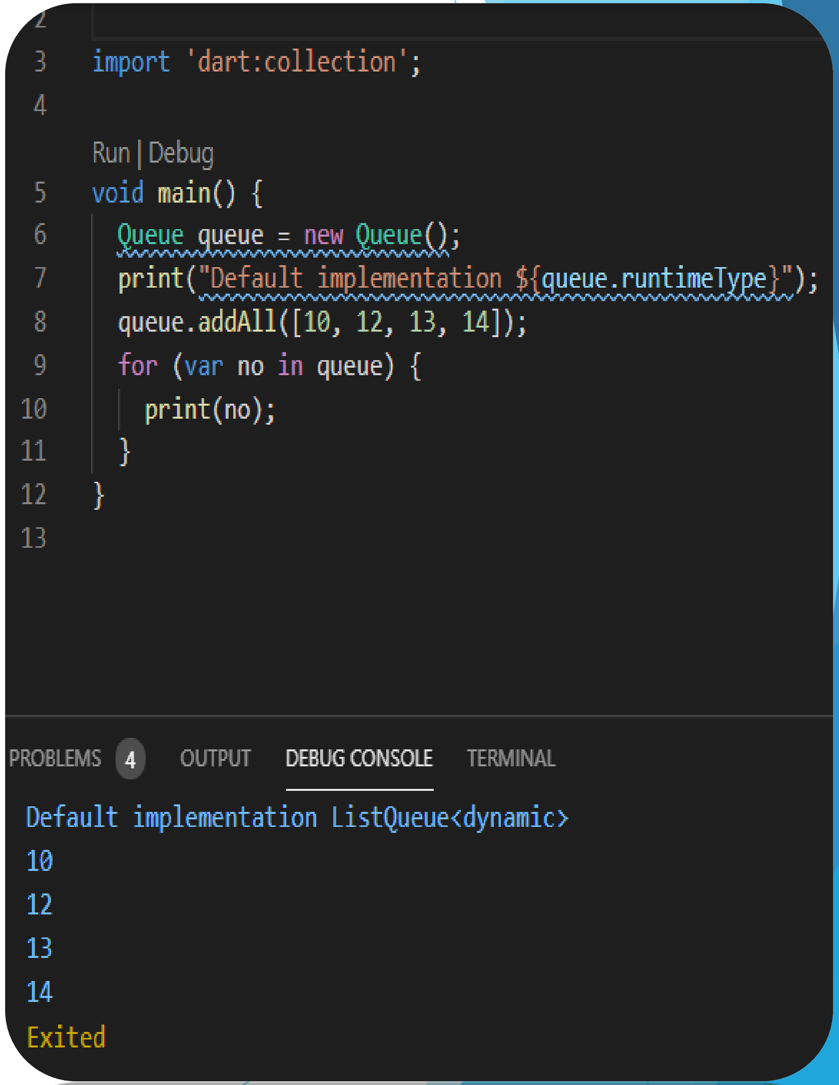
- Generic classes and methods can be used with different types by specifying the type arguments.

```
le of
::ToString( ),
e::ToString( String* ),
le::ToString( IFormatProvider* ), and
gle::ToString( String*, IFormatProvider* )
rates the following output when run in the [en-US] culture
ingle number is formatted with various combinations of form
rings and IFormatProvider.

IFormatProvider is not used; the default culture is [en-US]:
  No format string:                 11876.54
  'N5' format string:               11,876.54000
  'E' format string:                1.187654E+004
  'E5' format string:               1.18765E+004

A CultureInfo object for [nl-NL] is used for the IFormatProvider:
  No format string:                 11876,54
  'N5' format string:               11.876,54000
  'E' format string:                1,187654E+004

A NumberFormatInfo object with digit group size = 2 and
digit separator = '_' is used for the IFormatProvider:
  'N' format string:                1_18_76.54
  'E' format string:                1,187654E+004
Press any key to continue . . . _
```

Aitrich
Academy

Effective Dart: Best Practices and Style Guide

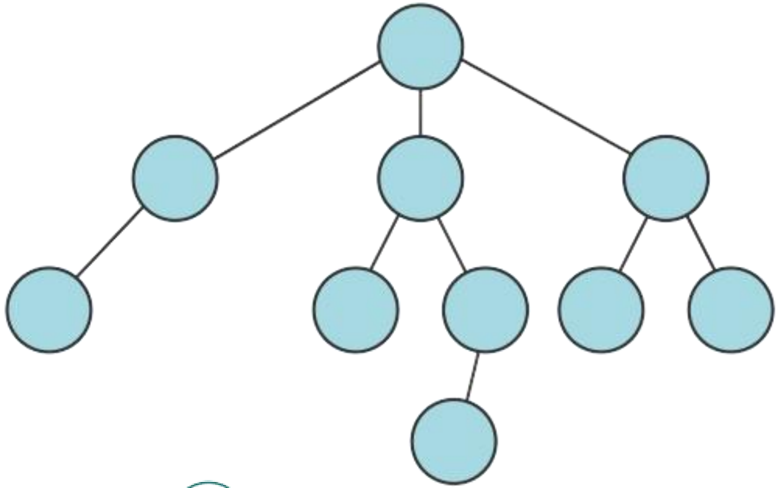
- Write clean, readable, and maintainable code.
- Follow consistent naming conventions.
- Use comments to explain complex code.
- Test your code thoroughly.

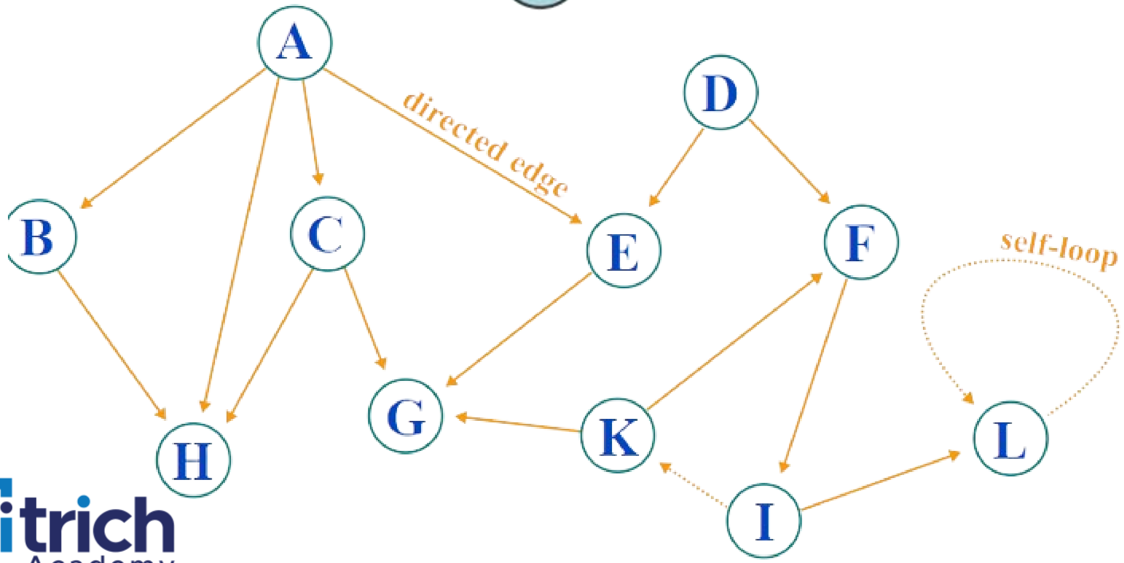


```dart
import 'dart:collection';

Run | Debug
void main() {
  Queue queue = new Queue();
  print("Default implementation ${queue.runtimeType}");
  queue.addAll([10, 12, 13, 14]);
  for (var no in queue) {
    print(no);
  }
}
```

```
Default implementation ListQueue<dynamic>
10
12
13
14
Exited
```

Advanced Data Structures in Dart





Trees

- Trees are hierarchical data structures that consist of nodes connected by edges.
- Dart provides the Tree class for representing and manipulating trees.

Graphs

- Graphs are collections of nodes connected by edges.
- Dart provides the Graph class for representing and manipulating graphs.

Aitrich Academy

Performance Optimization in Dart

- Identify and eliminate performance bottlenecks.
- Use efficient data structures and algorithms.
- Optimize code for the target platform.

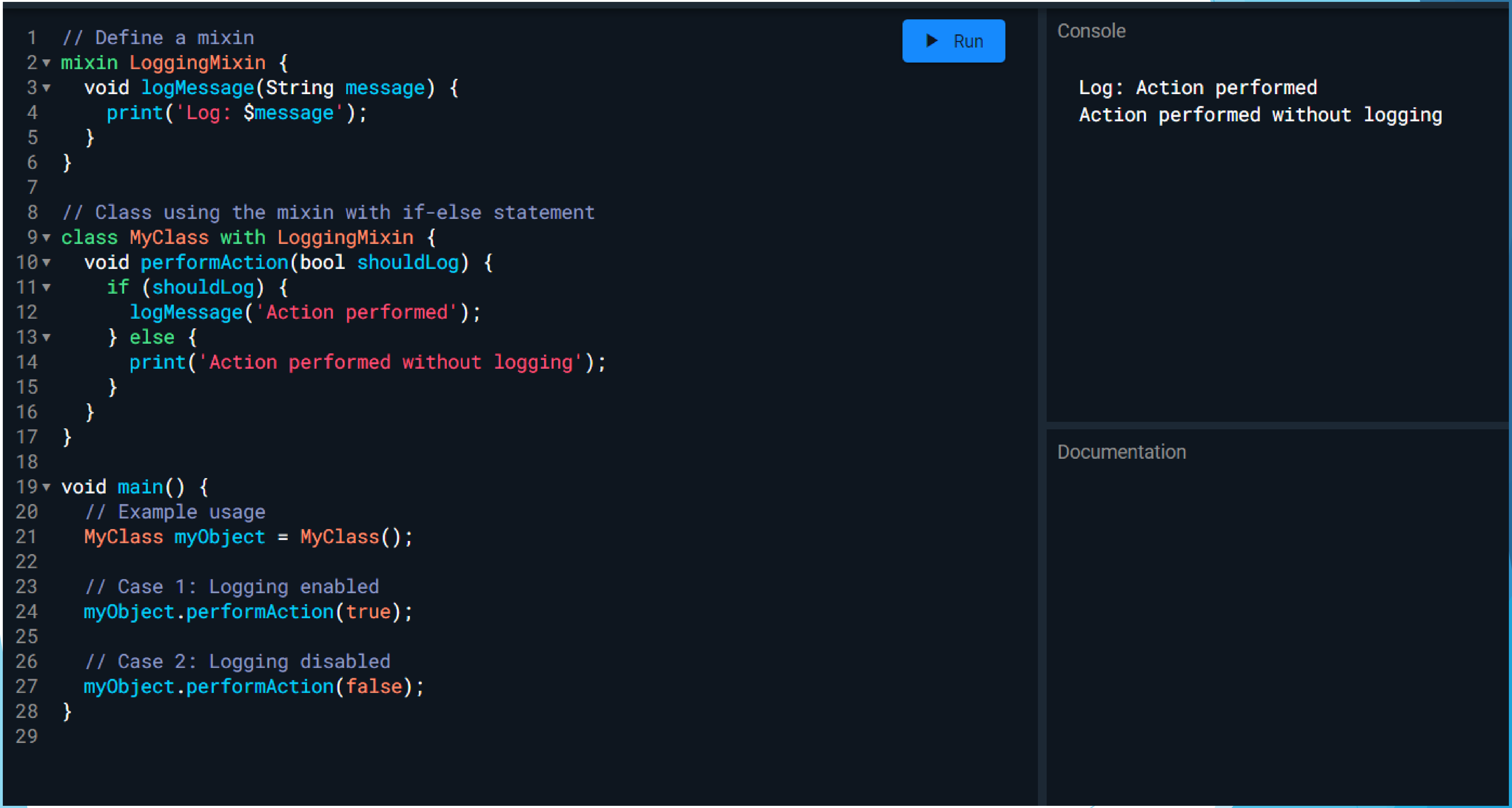
Aitrich Academy

Advanced Dart Features and Techniques

- Mixins: Reuse code across classes.
- Extensions: Add functionality to existing classes without modifying their source code.
- Metaprogramming: Manipulate code at runtime for powerful capabilities.

Aitrich Academy

Dart



```dart
// Define a mixin
mixin LoggingMixin {
  void logMessage(String message) {
    print('Log: $message');
  }
}

// Class using the mixin with if-else statement
class MyClass with LoggingMixin {
  void performAction(bool shouldLog) {
    if (shouldLog) {
      logMessage('Action performed');
    } else {
      print('Action performed without logging');
    }
  }
}

void main() {
  // Example usage
  MyClass myObject = MyClass();

  // Case 1: Logging enabled
  myObject.performAction(true);

  // Case 2: Logging disabled
  myObject.performAction(false);
}
```

```
Log: Action performed
Action performed without logging
```

In this example:

- We have a Logging Mixin, mixin that provides a log Message method.
- The My Class, class uses the mixin with the with keyword.
- The perform Action method in My Class takes a Boolean parameter should Log and uses an if-else statement to conditionally call the log Message method based on the value of should Log.
- In the main function, we create an instance of My Class and demonstrate two cases, one with logging enabled and one with logging disabled.

Concurrency and Multi-Threading in Dart

- Understand the concepts of threads and isolates.
- Use the isolate library for creating and managing isolates.
- Use the Future class for synchronizing concurrent operations.

Aitrich Academy

Conclusion

- Asynchronous programming and collections are essential tools for Dart developers.
- Dart provides a rich set of features for asynchronous programming and collections.
- By following best practices and using advanced techniques, you can write high-quality Dart code.

Aitrich Academy