

Additional Subjects on Dart

File Operations

→ File handling is an important part of any programming language.

Reading Text File

```
import 'dart:io';

void main() {
  // creating file object
  File file = File('test.txt');
  // other way of creating file object
  File file = File('C:\\Users\\test.txt');
  // read file
  String contents = file.readAsStringSync();
  // print file
  print(contents);
}
```

Reading Cvs File

```
// dart program to read from csv file
import 'dart:io';

void main() {
  // open file
  File file = File('test.csv');
  // read file
  String contents = file.readAsStringSync();
  // split file using new line
  List<String> lines = contents.split('\n');
  // print file
  print('-----');
```

```
for (var line in lines) {  
    print(line);    } }
```

Writing Text File

```
// dart program to write to file  
import 'dart:io';  
  
void main() {  
    // open file  
    File file = File('test.txt');  
    // write to file  
    file.writeAsStringSync('Welcome to test.txt file.');
```

→ If you have already some content in **test.txt** file, then it will be removed and replaced with new content.

Add New Content To Previous Content

```
// dart program to write to existing file  
import 'dart:io';  
  
void main() {  
    // open file  
    File file = File('test.txt');  
    // write to file  
    file.writeAsStringSync('\nNew content.', mode: FileMode.append);  
    print('New content is added on top of previous content.');
```

Write Cvs File

```
// dart program to write to csv file  
import 'dart:io';  
  
void main() {
```

```
// open file
File file = File("students.csv");
// write to file
file.writeAsStringSync('Name,Phone\n');
for (int i = 0; i < 3; i++) {
    // user input name
    stdout.write("Enter name of student ${i + 1}: ");
    String? name = stdin.readLineSync();
    stdout.write("Enter phone of student ${i + 1}: ");
    // user input phone
    String? phone = stdin.readLineSync();
    file.writeAsStringSync('$name,$phone\n',
mode: FileMode.append);
}
print("Congratulations!! CSV file written successfully.");
}
```

→ You can create any type of file using **writeAsStringSync()** method. For example, **.html**, **.json**, **.xml**, etc.

Delete Text File

```
// dart program to delete file
import 'dart:io';

void main() {
    // open file
    File file = File('test.txt');
    // delete file
    file.deleteSync();
    print('File deleted.');
```

→ If you try to delete a file that does not exist, then it will throw an exception.

Delete File If Exists

```
// dart program to delete file if exists
import 'dart:io';
```

```

void main() {
  // open file
  File file = File('test.txt');
  // check if file exists
  if (file.existsSync()) {
    // delete file
    file.deleteSync();
    print('File deleted.');
```

```

  } else {
    print('File does not exist.');
```

```

  }
}

```

Getting File Information

```

import 'dart:io';

void main() {
  // open file
  File file = File('test.txt');
  // get file location
  print('File path: ${file.path}');
```

```

  // get absolute path
  print('File absolute path: ${file.absolute.path}');
```

```

  // get file size
  print('File size: ${file.lengthSync()} bytes');
```

```

  // get last modified time
  print('Last modified: ${file.lastModifiedSync()}');
```

```

}

```



If you try to get information of a file that does not exist, then it will throw an exception.

Null Safety

→ **Null safety** is a feature in the Dart programming language that helps developers to avoid null errors

→ NNBD : Non-Nullable By Default

→ In Dart, variables and fields are non-nullable by default, which means that they cannot have a value **null** unless you explicitly allow it.

```
int productid = 20; // non-nullable
int productid = null; // give error

// Declaring a nullable variable by using ?
String? name;
```

→ You can use **if** statement to check whether the variable is null or not.

→ You can use **!** operator, which returns null if the variable is null.

→ You can use **??** operator to assign a default value if the variable is null.

```
// Declaring a nullable variable by using ?
String? name;
// Assigning John to name
name = "John";
// Assigning null to name
name = null;
// Checking if name is null using if statement
if(name == null){
    print("Name is null");
}
// Using ?? operator to assign a default value
String name1 = name ?? "Stranger";
print(name1);
// Using ! operator to return null if name is null
```

```
String name2 = name!;  
print(name2);
```

→ You can also store null in list values, the **items** is a list of nullable items. It can contain null values as well as item you specify for your list.

```
List<int?> items = [1, 2, null, 4];
```

→ Defining function with nullable parameters

```
// address is a nullable string  
void printAddress(String? address) {  
    print(address);  
}  
void main() {  
    // Passing null to printAddress  
    printAddress(null); // Works  
}
```

→ Defining null property in class

```
class Profile {  
    String? name;  
    String? bio;  
  
    Profile(this.name, this.bio);  
  
    void printProfile() {  
        print("Name: ${name ?? "Unknown"}");  
        print("Bio: ${bio ?? "None provided"}");  
    }  
}  
  
void main() {
```

```

// Create a profile with a name and bio
Profile profile1 = Profile("John", "Software engineer and a
profile1.printProfile();

// Create a profile with only a name
Profile profile2 = Profile("Jane", null);
profile2.printProfile();

// Create a profile with only a bio
Profile profile3 = Profile(null, "Loves to travel and try n
profile3.printProfile();

// Create a profile with no name or bio
Profile profile4 = Profile(null, null);
profile4.printProfile();
}

```

Late Keyword

→ In dart, **late** keyword is used to declare a variable or field that will be initialized at a later time. It is used to declare a **non-nullable** variable that is not initialized at the time of declaration.

```

// late variable
late String name;

void main() {
  // assigning value to late variable
  name = "John";
  print(name);
}

class Person {
  final int age;
  final String name;
  late String description;

  // constructor

```

```
Person(this.age, this.name) {  
  print("Constructor is called");}
```



The **late** keyword is contract between you and Dart. You are telling Dart that you will assign a value to the variable before you use it. If you don't assign a value to the variable before you use it, Dart will throw an error.

→ Late keyword also may be used for Lazy Initialization:

- What is Lazy Initialization?
 - **Lazy initialization** is a design pattern that delays the creation of an object, the calculation of a value, or some other expensive process until the **first time you need it**.

→ If you want to assign a value to a variable only once, you can use the **late final** keyword. This is useful when you want to initialize a variable only once.

```
// Student class  
class Student {  
  // late final variable  
  late final String name;  
  
  // constructor  
  Student(this.name);  
}  
  
void main() {  
  // object of Student class  
  Student student = Student("John");  
  print(student.name);  
  student.name = "Doe"; // Error  
}
```


Asynchronous Programming

→ **Asynchronous Programming** is a way of writing code that allows a program to do multiple tasks at the same time. Time consuming operations like fetching data from the internet, writing to a database, reading from a file, and downloading a file can be performed without blocking the main thread of execution.

→ In Synchronous programming, the program is executed line by line, one at a time. Synchronous operation means a task that needs to be solved before proceeding to the next one.

→ Asynchronous Programming improves the responsiveness of the program.

Future

→ It is used to represent a potential value, or error, that will be available at some time in the future.

→ **Future** represents the result of an asynchronous operation and can have 2 states :

- **Uncompleted** : When you call an asynchronous function, it returns to an uncompleted future. It means the future is waiting for the function asynchronous operation to finish or to throw an error.
- **Completed** : It can be completed with value or completed with error. If the future doesn't produce any value, then the type of future is `Future<void>`. If the asynchronous operation performed by the function fails due to any reason, the future completes with an error.

```
// function that returns a future
Future<String> getUsername() async {
  return Future.delayed(Duration(seconds: 2), () => 'Mark');
}
```

→ Usage of Future with .then()

```
// function that returns a future
Future<String> getUsername() async {
    return Future.delayed(Duration(seconds: 2), () => 'Mark');
}

// main function
void main() {
    print("Start");
    getUsername().then((value) => print(value));
    print("End");
}
```

Async

→ When a function is marked **async**, it signifies that it will carry out some work that could take some time and will return a Future object that wraps the result of that work.

async	async*
It gives a Future.	It gives a Stream.
async keyword does some work that might take a long time.	async* returns a bunch of future values on at a time.
It gives the result wrapped in future.	It gives the result wrapped in the stream.

Await

→ The **await** keyword, allows you to delay the execution of an async function until the awaited Future has finished. This enables us to create code that appears to be synchronous but is actually asynchronous.

```
void main() {
    print("Start");
    getData();
    print("End");
}

void getData() async{
```

```
String data = await middleFunction();
print(data);
}

Future<String> middleFunction(){
  return Future.delayed(Duration(seconds:5), ()=> "Hello");
}
```



You can use the **async** keyword before a function body to make it asynchronous. You can use the **await** keyword to get the completed result of an asynchronous expression.

Streams

→ A stream is a sequence of asynchronous events.

```
// function that returns a stream
Stream<String> getUsername() async* {
  await Future.delayed(Duration(seconds: 1));
  yield 'Mark';
  await Future.delayed(Duration(seconds: 1));
  yield 'John';
  await Future.delayed(Duration(seconds: 1));
  yield 'Smith';
}
```

→ **yield** returns the value from the stream. To use **yield** you have to use `async*`.

→ You can also create a stream by using `Stream.fromIterable()` method.

```
// function that returns a stream
Stream<String> getUsername() {
```

```
    return Stream.fromIterable(['Mark', 'John', 'Smith']);  
  }
```

Usage of Stream

```
// function that returns a stream  
Stream<String> getUsername() async* {  
  await Future.delayed(Duration(seconds: 1));  
  yield 'Mark';  
  await Future.delayed(Duration(seconds: 1));  
  yield 'John';  
  await Future.delayed(Duration(seconds: 1));  
  yield 'Smith';  
}  
  
// main function  
void main() async {  
  // you can use await for loop to get the value from stream  
  await for (String name in getUsername()) {  
    print(name);  
  }  
}
```

→ There are 2 types of stream

- Single Subscription Stream (Default stream) : They hold onto the values until someone subscribes and can only be listened to once. You will get an exception if you try to listen more than once. Any event's value should not be missed and must be in the correct order. Inside the stream controller, there is only one stream, and only one subscriber can use that stream.
- Broadcast Stream : This is the stream that is set up for multiple subscriptions. They hold onto the values until subscribers can only listen many times. You can use the broadcast stream if you want more objects to listen to the stream. It can be used for mouse events in a browser.

Inside the stream controller, many streams can be used by many subscribers.

Future	Stream
Future represents the value or error that is supposed to be available in the Future.	Stream is a way by which we receive a sequence of events.
A Future can provide only a single result over time.	Stream can provide zero or more values.
You can use FutureBuilder to view and interact with data.	You can use StreamBuilder to view and interact with data.
It can't listen to a variable change.	But Stream can listen to a variable change.
Syntax: Future <data_type> class_name	Syntax: Stream <data_type> class_name


📌 This is not a detailed notes based on Stream, this is just the understand what is stream. For detailed information and methods of stream;

- <https://dart.dev/tutorials/language/streams>

References

OOP In Dart

Learn Dart Programming

 <https://dart-tutorial.com/object-oriented-programming/>



 **Author → Serhat Kumas**

<https://www.linkedin.com/in/serhatkumas/>

SerhatKumas - Overview

Computer engineering student who loves coding in different fields instead of focusing on a one spesific area. - SerhatKumas

 <https://github.com/SerhatKumas>

