Dart Topics

1. String interpolation

new : String Interpolation print('My name is $name');

2. Data Types

num,int,double,String,bool are common primitive data types.
List,Map,Set(these types covered after in point 8.)

3. Null-Safety

Dart is a strong null safe language. Dart has a built-in Null-Safety, that no variable can contain null value at the time of initialization. To make it able to contain null you have to add a null-able **operator** '?' at the end of data type.

i.e. **String name;** cannot contain null it must have a string value otherwise it will show syntax error.
String? name; can contain null as it contains a null-able operator.

4. Cascading Operator '..'

using this operator after any constructor will allow us to access any instance member of that class.
kind of built-in builder :) factory design pattern.

(i) '?' **null-able operator** to make a type null-able (String (not-nullable) --> String? (nullable))

(ii) '!' **null-assertion operator** to make sure that null-able reference doesn't contain null. (int? a; [print(a) (error(could be null) ---> print(a!)]))

(iii) '?.' **conditional property access operator** if the object is not null then access the member otherwise skip. (student?.study();)

(iv) '??' **null-coalescing operator** to provide a default value if it contains null. ()

(v) '??=' **null-coalescing assignment operator** to assign a default value if it contains null. ()

(vi) '..' **cascading operator** to access any instance member.

(vii) '?..' **null aware cascading operator** to access any instance member if the reference is not null.

(viii) late

String? nullableString;

```
nullableString ??= “Default Value”;
print(nullableString ?? 'Default Value');
print(nullableString != null ? nullableString : 'Default Value');
```

5. Variable Types

(i) Strict Typing : cannot be change at anytime of program i.e. int a = 0;

(ii) Type Inference : Data initialized at first time will decide its type till the end of program i.e. var a; a = 2; now a is of 'int' type.

(iii) Dynamic Typing : can be changed at any time any type of data can be assigned to it.

Best Approach to use these declarations :

1. Strict typing for functional Structure(parameters, RDT) and class data members.
2. Type Inference can be used (optional) at functional level. or top-level variables.
3. Dynamic typing shouldn't be used until it's necessary. (Haram)

6. Constant Keywords

const : to make compile time constant i.e. static const int a = 4;
const a = 4; => Compile Time Constant Type Inference
final : to make runtime constant i.e. final Student s = Student();
final s = Student(); => Run Time Constant Type Inference
const s1 = Duration(); => Compile Time Constant Type Inference Example
we can make any object of the user-declared class compile time constant but using a specific design.

7. Compile time vs Run time

Compile Time:

1. When a variable is declared at compile time, its memory allocation occurs even before the execution of the program.
2. Compile time is the time when source code is translated into executable code.
3. The compile time errors are the errors which occur during compiling like syntax errors which can be detected by the compiler.

Run Time:

1. When a variable is declared at runtime, its memory allocation occurs after the execution of the program while it is running.
2. Runtime is the time when the program is running generally happens after compile time.
3. Run time is the time when executable code starts executing (program starts running).
4. The run time errors are errors which cause unpredictable results during runtime, like null pointer exceptions.

8. Collection Data types (List, Map, Set)

(i) List [] , Set {} , Map {key:value}

List : collection of objects(items) of any type.
Set : collection of unique objects(items) or non-repetitive entries of any type.
Map : collection of key:value pairs of any type. (i.e. Json is Map literal)

(ii) Spread Operator '...' :

to insert an existing Collection into the other Collection.

(iii) Null-Aware Spread Operator '...?' :

to insert an existing collection after checking that if it's not null to the other collection.

(iv) Collection If

i.e. [1,2,3,4,5,if(condition) literal else value,6,7,8,9,];

(v) can insert entry using iteration

i.e. [for(int i = 1;i<=2000;i++)i];

9. Programming Types

Imperative Programming : Imperative programming uses the paradigm of programming where we explain "HOW" the program will be executed. In this programming we explain how the flow will get controlled and how it will be resolved.
Declarative Programming : Declarative programming uses the paradigm of programming where we explain "WHAT" the program will do. It means it already knows how to resolve the program. We just need to tell it what to do.

10. Functions

Function is a subprogram in a program. There are different ways to declare a function according to parameters which are follow:

Function Type Name	Definition Sample	Call Sample
(i) Simple Parameters	num sum(num num1,num num2,num num3){return 0};	sum(1,2,3);
(ii) Optional Positional Parameters	num sum(num num1,[num? num2,num? num3]){return 0};	sum(1); / sum(1,2); / sum(1,2,3);

(iii) Optional Positional Default Parameters	num sum(num num1,[num num2 = 20,num num3 = 30]){return 0};	sum(1); / sum(1,2); / sum(1,2,3);
(iv) Optional Named Parameters	num sum(num num1,{num? num2,num? num3}){return 0};	sum(1,num2:5); / sum(1,num3:9) / sum(1,num3:9,num2:5);
(v) Optional Named Default Parameters	num sum(num num1,{num num2 = 20,num num3 = 30}){return 0};	sum(1,num2:5); / sum(1,num3:9) / sum(1,num3:9,num2:5);
(vi) Required Named Parameters	num sum({required num num1,required num num2}){return 0};	sum(num1:4,num2:5); / sum(num2:4,num1:5);
(vii) Optional/Required Named Parameters	num sum({required num num1,num? num2,num? num3}){return 0};	sum(num1:6); / sum(num2:4,num1:5,num3:7);
(viii) Optional/Required Named Default Parameters	num sum({required num num1,num num2 = 20 ,num num3 = 10}){return 0};	sum(num1:6); / sum(num2:4,num1:5,num3:7);

11. Functions as Objects and their callable Classes

Functions are **Objects** in **Dart**.

As everything in dart is an object. A function is also an object in Dart. The class of a Function which is a callable class always extends Function Class implicitly. There are three ways by which function type(class) is declared.

Anonymous Classes :

When you declare a function, the compiler implicitly creates its anonymous callable class (meaning it has a "call" function implicitly) which also implicitly extends Function Class .

Concrete Class :

You can also create a concrete class and make it callable by adding a call function in it with the similar signature as its object(function) should have.

Abstract Class :

We actually define the type with the function signature so it can take reference of every function with the same signature as we defined.We can do this by using the reserved word “**typedef**” and then define our type.

12. Dynamic Functions

Using a Function (A) as a parameter in another function (B) will work as a dynamic function in function (B) .

Function (B) will work statically other than where function (A) is called will work dynamically.

13. Function as Return Data Type

Function is an object that also has a type so it can also be used as Return Data Type.
When a Function (A) returns a Function (B) it actually returns closure of that Function (B) and if the parameters or local variables of Function (A) are being used in the lexical scope of the Function (B), then they will not get destroyed or terminated on the execution of Function (A) until the Function (B) gets executed.

14. Helper Methods of Collections (Especially Using dynamic Programming)

In collections, we have helper methods to perform different operations on the list. Like add,remove,find,replace etc also we have few helper methods which use dynamic programming to perform functionalities dynamically like where,whereSingle,every,etc.

15. Constructors

Initializer List = " : "

It is mainly used to initialize the final data members or instance members using parameters or to redirect constructor.

(i) Simple Constructor :

Simple Constructor as we studied in Java It can include default parameterized or copy constructors. In Dart, there is no constructor overloading because of it''s parameter types using named optional or named required constructors to cover the need of overloading.

(ii) Named Constructor :

If we want to create object using JSON or any way other than using data members or an existing object we can use named constructor it is declared as we call static member in Java using class name so a named constructor in Student Class will look like :

```
Student.fromJson(Map<String,dynamic> json){
…(body)...
}
```

(iii) Redirecting Constructor :

These are constructors if we want to redirect a constructor to another constructor i.e. a named constructor to simple constructor.i.e. If we redirect the above constructor to the simple constructor.

Student.fromJson(<String,dynamic> json):this(name : json[name], age : jason[age]);

(iv) Factory Constructor :

There are in general two types of constructors. Generative Constructors which create the objects and Factory Constructors which will return the existing objects (which are already created) at run-time. Simple constructors are an example of Generative Constructor.
Factory constructors are the constructors which will return an existing object other than creating a new object. i.e. In a function there are objects already present so it will return that object on the basis of input. In simple words a special constructor which will not construct the new object but will return an object which is already present.

(v) Const Constructors :

These are the constructors which are used to make object compile time constants for creating these constructors it is important for all the data members to be final or static const so that they cannot be changed at run time. Simple Classes with simple constructors cannot be compile time constants as the objects create at run time but using const Constructors we can create objects which are compile time constants.
Class **Duration** has const Constructors so its object can be compile-time constants.

16. Properties of Data Member

We can also declare properties of Data Members. Properties are special member functions which we access and consume as data members. i.e. **set** and **get** can be properties of any private data member to handle that data member. If a data member has both set and get property we will say that the data member has both read and write properties if it has only get property we will say that it has read only property and if it has only set property we will say that it has write only property.There is no need to define the list of parameters when we define the get property because there is no parameter required when we get anything.

Book Model Class

```dart
void main(List<String> arguments) {

  Book demo = Book();

  // using set property to set the value
  // compile will figure out that it is 'set' property as accessed left side of assignment
operator
  demo.author = "Charles Babbage";

  // using get property to get the value
  // compiler will figure out that it is 'get' property as not accessed left side of
assignment operator
  print(demo.author);
}
```

Book Model Class

```dart
class Book {

  // private String type data member
  String? _author;

  // set property to set above private data member
  set author(String author){
    _author = author;
  }

  // get property to get above private data member
  String get author{
    return _author ??= "empty";
  }

}
```

17. Operator Overloading

1. Java doesn't Allow Operator Overloading but Dart does.
2. Dart allows us to overload the functionality of any operator (for compiler) for any class (Abstract Data Type).
3. Operator is a Function and Function Overloading is not allowed in Dart. so we actually override that operator function but for symmetry with other languages we call it as **Operator Overloading**.

4. For overriding, the signature must be the same but in Object class Object O come in as parameter but in custom class CustomClass C come as parameter so we use the word **covariant** before type so the compiler understands that it will also be a kind of an Object O.
5. If we define (actually overrides the abstract function in **Object** class) an operator for a custom class (Abstract Data Type), we can use that function for child classes too because of inheritance and also can override in any specific child if the functionality changed.
6. As we discussed, operator is getting overridden as all operators (as functions) in Object class are Abstract other than **==** operator as it has concrete implementation in Object class.

Operator Overloadding

```
class Operator {

  int _number = 21;
  int get number{
    return _number;
  }

  int operator + (Operator o){
    return _number + o._number;
  }

  @override
  bool operator == (covariant o){
    return true;
  }

}

    Operator o1 = Operator();
    Operator o2 = Operator();
    int o3 = o1+o2;

class ChildOperator extends Operator {
  int value = 0;

  @override
  int operator + (Object co){
    return 3;
  }

}
```

18. Inheritance

Inheritance is similar to Java. Every class can be inherited by only one class directly.
Inheritance has two major advantages.

(i) Generalization (so we can handle objects of different class with same data type or reference variable of parent)

i.e. Triangle, Circle, Square all can be handled using the Reference Variable of Shape.

(ii) Reusability (so the code in parent class can be reusable for every single child class, even if a child wants specialized implementation it can override the function.)

i.e. hashcode function declared in the Object class is getting reused by every single object of any class because of Inheritance as Every class Implicitly extends the Object class.

19. Abstract Class

1. Abstract Class is similar to Java.
2. We can declare abstract functions in Abstract Class.
3. Abstract Class can't create independent objects.
4. Abstract Class has a constructor which is called Implicitly in the Child Concrete Class.

20. Interfaces

Dart doesn't have (secondary) Interfaces as in Java but it has a primary interface. Class itself in dart is the primary interface so we can implement or extend a class as per requirement. Interface doesn't contain any constructor. and it must override all the data member and member functions regardless if they are abstract or not.

Data Member of Interfaces will behave as instance member implicitly.

21. Mixins

1. Mixins are also a type like class or enum in dart.
2. A mixin behaves as an abstract class without a constructor.
3. A mixin cannot be extended but it can be used with the keyword "with".
4. A mixin can also be implemented as an interface with the keyword "implements".
5. A class can use multiple mixins at a time .
6. We use mixins for those entities which cannot have an object independently.
   (i) Roof top of a car will create its object with a car object.
   (ii) The back Cart of a car will create its object with a car object.
   (Independently these objects are useless)
7. A mixin cannot extend another mixin but it can implement another mixin.
8. However, if a mixin implements another mixin it is not necessary to override all the fields of the implemented mixin.

9. If a class used the mixin (A) which implemented another mixin (B) it will be necessary for that class to implement the methods of the implemented mixin (B) .
10. If a class can use multiple mixins then the diamond problem can occur but the language itself handels the diamond problem.A mixin can also be used to generalize all the classes that used the mixin.

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

void main(List<String> arguments) {
  print('Hello world: ${dart_mixin_type.calculate()}!');
}

mixin Shoot{
  void woo();
}

mixin Cart implements Shoot{
  void soo(){}
  void foo();
}

class CarOne with Cart {
  @override
  void foo() {}
  @override
  void woo() {}
}

class CarTwo implements Cart {
  @override
  void foo() {}
  @override
  void soo() {}
  @override
  void woo() {}
}
```

22. Class Modifiers

(i) Abstract

If we use an “abstract” modifier with a class then it can be implemented as well as extended outside as well as within its own library. However, its object cannot be created. Also we can declare abstract functions within the class which will have their implementation in concrete classes.

(ii) Base

If we use a “base” modifier with a class then the class can be extended as well as implemented within its own library but outside the library it can be extended only and cannot be implemented. However, its object can be created both within and outside the library.All the child classes that extend the base class should be final, base or sealed so that the child classes can also be not implemented.

(iii) Interface

If we use an “interface” modifier with a class the class can be extended and implemented within its own library but outside the library it can only be implemented and cannot be extended. However, its object can be created both within and outside its own library. We can declare both concrete and abstract functions in the class.

(iv) Abstract Interface

If we use an “abstract interface” modifier with a class then the class will behave as a genuine Interface as in Java. because abstract will restrict it for creating its own object and interface keyword will restrict it to get implemented outside the library. So it will behave as a pure interface type of java outside the library.

(v) Final

If we use the “final” modifier with a class the class can be extended as well as implemented within its own library but cannot be extended or implemented outside the library. We use this modifier when we don't want others to define their own implementation of the class and we are sure that the functions cannot have other functionality than the functionality already defined.However, its object can be created within as well as outside the library.

(vi) Sealed

If we use the “sealed” modifier with a class the class will be restricted to get implemented or extended outside the library. The compiler is aware of any possible direct child classes of sealed classes as they can only exist in the same library. Sealed classes are implicitly Abstract. However, the child classes of the sealed classes are not implicitly abstract.

Additional Notes :

- If you don’t want to use exhaustive switching, use **final** instead of **sealed**.
- It’s much better to use **abstract interfaces** rather than using **interface** to define a pure interface.
- Use **base** for purely Inheritance advantages, users can’t implement your class.

23. Package, Library and File

- In Dart, a **File** is a collection of classes or global functions or both.
- Implicitly a **File** is a **Library** but a library can contain multiple Files too.

(we can do this by adding “part Extended_File.dart” to main file and “part of Main_File.dart” to extended file)

- Non-executable Library is actually a **Package** as the package doesn’t contain a “bin” folder.

24. Extension methods

1. Dart allows us to add any functionality to built in classes.
2. We can do this using the Extension method where we add the keyword “extension” and define the name of our class which becomes the extension of the class whose extension method we want to create using the keyword “on”.
   i.e. **extension** Class **on** Defined_Class
3. We can then define extra methods in the class body for our use which can be only accessed by us

. **Example:** if we want to create an extension method in a class of String we can create its extension like :

```
extension Conversion on String{
  int toInt()=> int.parse(this);
}
```

4. We cannot create the object of the extension class that we made.However, we can access the methods that we created using the Parent class.
5. We can also create unnamed extensions but these extensions cannot be accessed outside the library.

25. Switch as an Expression

In Dart we can use a switch as a statement as well as an expression. If we use switch as a statement it does not return a value and it cannot be used on the right side of the assignment operator (A statement is a void expression). However , if we use switch as an expression it can return a value and it can be assigned to a variable i.e it can be used on the right side of the assignment operator. **Switch as an expression :**

```
var c = switch (a) {
   1=> 'One'   ,
   2 => 'Two',
   3 => 'Three',
   _ => 'Four'
};
```

The syntax of a switch differs from that of the switch statement syntax as follows :

- Cases *do not* start with the `case` keyword.
- A case body is a single expression instead of a series of statements.
- Each case must have a body; there is no implicit fallthrough for empty cases.
- Case patterns are separated from their bodies using `=>` instead of `:`.
- Cases are separated by `,` (and an optional trailing `,` is allowed).
- Default cases can *only* use `_`, instead of allowing both `default` and `_`.

26. Template Type (< >)

Dart allows us to make template type classes as well as functions. We can also use a Template type as a Return data type of a function. We can also define a specific type that can be passed in the template type by extending it with the desired class. Example if we want to make a function that takes a type which is a String or a class which is a child of String we will define it like this :

```
void foo<T extends String>(){

  print(T);

}
```

Now this function will only take a type which is a String or a type of string i.e Child of String.

27. Records

1. Records are primitive, anonymous, immutable, aggregate, algebraic and composite data types (it helps you store different types of data in a single object).
2. Records are fixed-sized, heterogeneous and typed.

3. Records are real values; you can store them in variables, nest them, pass them to and from functions, and store them in data structures such as lists, maps, and sets.
4. The types that we define inside the record make up the shape of record.The names of named fields in a record type are part of the record's type definition or its shape.
5. Two records with named fields with different names have different types and you cannot assign them to each other.
6. However, if you want to assign two named parameter records to each other then both the records should have the same data type in the shape and the name of the variables should also be the same. But if you want to assign two positional records to each other you can do that but the data types in the shape have to be the same . Records only have getters becoz they are immutable.

```
void main(List<String> arguments) {

// custom record
(String,int,bool) RecordType = ("String",4,true);
// general two objects record reference
(Object, Object) generalObject;
// function returning a record
(String,int,bool) foo(){
  return ("Ok",4,true);
}
// assignable because of same record nature
RecordType = foo();
// assignable because of positional parameters
(int,int) a = (1,2);
(int,int) b = a;
// assignable because of positional parameters
(int a,int b) c = (3,2);
(int c,int d) d = c;
// named parameters
({int a,int b}) e = (b: 3,a: 2);
// error can't assignable because of different named parameters
({int c,int d}) f = e;
// assignable because of same named parameters
({int a,int b}) g = e;
// assignable because of specified the named parameters while assigning
({int c,int d}) h = (d: e.b,c: e.a);
// assignable because of generalized nature
generalObject = h as (Object,Object);

}
```

28. Pattern Matching

Patterns are a syntactic category in the Dart language, like statements and expressions. A pattern represents the shape of a set of values that it may match against actual values.In general, a pattern may **match** a value, **destructure** a value, or both, depending on the context and shape of the pattern.
First, *pattern matching* allows you to check whether a given value:

- Has a certain shape.
- Is a certain constant.
- Is equal to something else.
- Has a certain type.

Then, *pattern destructuring* provides you with a convenient declarative syntax to break that value into its constituent parts. The same pattern can also let you bind variables to some or all of those parts in the process.The most simple use case of pattern matching is using a switch case that if the coming expression matches a case then what should we do. Many patterns also make use of subpatterns Example if we want to check whether the coming list contains two strings ”a” and “b” we can write is as :

```
//pattern matching
  const a = 'a';
  const b = 'b';
  var list = <String>['a', 'b'];
  switch (list) {
    case [a, b]:
      print('$a , $b');
      break;
    default:
  }
```

When an object and pattern match, the pattern can then access the object’s data and extract it in parts. In other words, the pattern *destructures* the object. If we want to destructure a list and assign it to variables we can do it in the following way:

```
//destructuring list
  var numberList = <int>[1, 2, 3];
  var [d, e, c] = numberList;
  print(d + e + c);
```

We can use patterns at several places in Dart like :

- Local variable declarations and assignments
- for and for-in loops
- if-case and switch-case
- Control flow in collection literals

If we want to declare variables we can do it like this :

var (a,[b,c]) =(“ hello “,[1,2]);

This declares one **String** variable and two **int** variables. If we want to declare many variables of the same type we can do it by writing them in a list literal like above example variables “b” and “c” are of type int. A variable declaration may start with **var** or with **final**.

We can also use pattern matching in destructuring objects. The pattern matching checks whether the object matches a certain pattern and if it matches it assigns the values of the object to the variables as follows :

```
//Variable assignment
var (k, i) = (5, 6);
(i, k) = (k, i);
print('$k , $i');
```

In the above example the patterns matches the values destructure the object and assign it to the variables and then we can easily swap the values without creating a new variable and using the existing variables only.

If a function has multiples returns(Record) we can assign it to the variables with the help of pattern matching:

```
//Destructuring multiple returns
var (rollNo,name, fees)=student(2, 'Ali', 3849.938);
print(rollNo.runtimeType);
print(name.runtimeType);
print(fees.runtimeType);
```

In the above example the student function returns a record which then matches the pattern and if the pattern matches it destructures the object and assigns the values to the variables.

We can also destructure class instances and assign them to variables. However, if we use named parameters for pattern matching we can make a variable of any name but if we use only a semi colon on the left side of the variable then the variable has to have the same name as the data member of the class otherwise the pattern will not match. Consider the following example :

```
//destructuring instances
  Student s = Student();
  var Student(:rollNo, :name, :fees,) = s;
  print('${rollNo + 2}');
  var Student(rollNo: r, name: n, fees: f) = s;
  print(name.runtimeType);
```

We can also use map and list patterns for destructuring key-value pairs for incoming JSON-data.

```
//destructuring Json
var json = {
  'user': ['Lily', 13]
};
var {'user': [name3 as String, age3 as int]} = json;
print(age3 + 6);
// Same method use "case" keyword
if (json case {'user': [String name4, int age4]}) {
  print('$name4,$age4');
}
```

29. Asynchronous Programming / Functions

1. Dart allows us to make functions that return **Future** or **Stream** objects.
2. Dart also contains built-in functions that return **Future** or **Stream** objects.
3. **Stream** is an asynchronous collection of Futures.
4. **Future** or **Stream** are asynchronous functions.
5. In **synchronous functions** the function gets executed in the main thread means it will block the execution until the function gets executed while **asynchronous function** gets executed in a separate thread means the function will be executed in separate thread and in the meantime the other code will keep getting executed.
6. Functions which are potentially time consuming should be **asynchronous** like downloading a file or downloading resources.
7. To create an asynchronous function simply make it return **Future<T>** it simply means that in future whenever the function gets executed completely you'll have the return value in **then** function.
8. Future is a built-in class in the compiler but not considered a primitive type.

```
2
3  Future<String> createString(){
4    return Future.delayed(const Duration(seconds: 5),() => "Hello");
5  }
6
7  class Foo {
8
9  qoo(){
10   createString().then((value) => print(value));
11 }
12
13 }
14
```

In the above example the asynchronous function **createString** returns the Future of String after 5 seconds delay and if we assign this function to a variable the variable will receive the future of string and not the value of String.

9. However we can make our asynchronous function behave like a synchronous function i.e if a synchronous function comes after an asynchronous function the compiler will wait till the asynchronous function is completed and then execute the next function.
10. To make this happen we have to make the function async in which we are calling the asynchronous function and use the word **await** with the async function. Now the compiler will w8 till the async function is completed and then it will move to the next function in other words the async function will behave synchronously.
11. In this case the function will not return the **Future<T>** but it will return the value and assign it to the variable after the future is achieved.

```
Run | Debug
void main(List<String> arguments)async{
    var result=await futuremessage();
    print('hello');
    print('hi');


}
```

30.Generative Functions

Generative functions are the functions that allow us to return multiple values from a function. A simple function can return a single value.A generative function does not use the word return but it uses the word **yield**. Generative functions are of two types :

- Asynchronous generative functions
- Synchronous generative functions

(i) **Asynchronous generative functions**

When an asynchronous function is called it generates an object of **Stream<T>** and as the function returns the values they are continuously synced in the object of the Stream. The return data type of an asynchronous generative function is a Stream.A Stream is an asynchronous collection or a Stream is a list of futures.To make an asynchronous generative function simply write the word **async*** between the function header and the function body.

```
Stream<num> naturalNumbers({required int value})async*{
         for (var i = 1; i <= value; i++) {
        await Future.delayed(const Duration(seconds: 20));
          yield i;
         }

}
```

(ii) **Synchronous generative functions**

When a synchronous function is called it generates an object of **Iterable<T>** and as the function returns the values they are continuously synced in the object of the Iterable. The return data type of a synchronous generative function is a Iterable.To make a synchronous generative function simply write the word **sync*** between the function header and the function body.

```
Iterable<num> number()sync*{
for (var i = 0; i < 7; i++) {
      yield i;
}
}
```

31. Stream

1. Stream is Asynchronous Collection of Future.
2. If we want to use the **yielded** value we can do it by listening to the stream and this can be done by using the **listen();** method which provides us the value each time the stream receives a new value.

Types of Streams :

- **Single Subscription Stream**
- **Broadcast Stream**

(i) Single Subscription Stream

In the Single Subscription stream we can listen to the stream only once and we cannot do multiple tasks on the stream. By default the stream is a single subscription stream.We assign our stream function to the variable if we want to "**pause,stop or resume "** our stream if a specific value comes that is not needed in the stream collection other- wise we can anonymously listen to our stream and perform the operation.

```
//If we want to handle different scenerios froma  stream we assign it to a Stream subscription
  StreamSubscription stream=naturalNumbers(value: 12).listen((event) {print(event)});
                stream.pause();
                stream.cancel();
                stream.resume();
//if we only want to listen to the stream
naturalNumbers(value: 15).listen((event) {print(event);});

}
Stream<num> naturalNumbers({required int value})async*{
          for (var i = 1; i <= value; i++) {
         await Future.delayed(const Duration(seconds: 20));
           yield i;
          }

}
```

(ii) Broadcast Stream

If we want to listen to our stream multiple times we make it a Broadcast Stream. After that we can perform multiple functions on a single stream and even map it to other streams.To make a stream a broadcast stream we have to use a function "**asBroadcastStream()"** when we are calling our generative function and assigning it to a variable.

```
var  noStream=naturalNumbers(value: 10).asBroadcastStream();

  noStream.listen((event) {print(event);});

  var strStream=noStream.map((event) => event*2.5);

  strStream.listen((event) {
   print('The value of this number after multiplying it with 2.5 is = $event');
  });
```

In the above Example we register our stream as a broadcast Stream so we can listen to it multiple times. And we are mapping our stream of **int** type to a stream of type **double** and then listening to the Stream that is of type **double.**

32. Stream Controller

There are two ways to make the object of stream.The first way is to create a Stream using a generative function that yields the value that is constantly synced in the object of stream. However, we can do it in another way by using the stream controller. Firstly, we have to make an object of the stream controller and it is a template type class"**StreamController<T>**". The template of the controller will be the same as the template of the Stream object. There are two ways to make an object of stream :

(i) First Way

```
Stream<num> numbers(){
  StreamController<num> streamController=StreamController();
  StreamSink sink=streamController.sink;
  Stream<num> stream=streamController.stream;
  for (var i = 0; i < 200; i++) {
    sink.add(i);
  }
  streamController.close();
  return streamController.stream;
}
```

In the above Example we make two objects using the streamController the first object is the **StreamSink** which is the input method of the controller to sink all the value in the Stream object and the second object is the Streams Object in the value will be synced. In other words the **sink** is the input for the controller and the **stream** is the output method of the controller.

(ii) Second Way

```
Stream<num> numbers(){
  StreamController<num> streamController=StreamController();
  for (var i = 0; i < 200; i++) {
    streamController.sink.add(i);
  }
  streamController.close();
  return streamController.stream;
}
```

In this example we don't create two objects, we directly sink the value in the stream and then return the stream.However, both the examples are the same.It depends on you which example suits you the best.

However, when we call the function the loop inside the function does not execute; however the object of the Stream controller is created and the stream is returned to the variable. When we listen to this stream then the for loop starts executing and the values are synced in the stream and the listener listens to the stream and after the loop is finished the stream closes.

```
var stream=numbers();
stream.listen((event) {print(event);});
```

33. Using Stream Controller and Generative Functions

One question arises that if both the stream controller and the generative functions do the same thing i.e generate the object of the stream and return the values in it. So when should we use generative functions ? and when should we use a Stream controller ?. The answer is simple: when we know that there is only one function that will generate the object of the stream then we will use generative functions and when there are multiple functions that will create the objects of the stream we will use the steam controller and use that same controller to sync the values of the stream. Usually we use this when we make our own class of stream.

```
class Generator {
 final StreamController<num> _streamController=StreamController.broadcast();


   Stream<num> intNumbers(){

  for (var i = 0; i < 200; i++) {
    _streamController.sink.add(i);
  }

  return _streamController.stream;
}

Stream<num> doubleNumbers(){

  for (var i = 0; i < 200; i++) {
    _streamController.sink.add(i*2.5);
  }

  return _streamController.stream;
}

}
```

34. Input and Output

If we want to show a message on the console we can do it by using the function "**stdout.write()**". And if we want to read anything from the console we can do it by using the function "**stdin.readLineSync()**". Both these functions are global functions of the package "**dart : io**". The function **stdin.readLineSync()** is an asynchronous function and returns the value in the form of **String?.** We can parse it in any data type that we want to and then use it.

```
//Readin a string form the console
  stdout.write("Enter the name : ");
  String? name=stdin.readLineSync();
  print(name);

  //Reading integer data type from console
  stdout.write('Enter the number : ');
  int number=int.parse(stdin.readLineSync()!);
  print(number);
```

35. Reading a File

We can also read a file from disk and print it. We can do this by using the built in library "**File**" and giving the path of the file into its constructor.

```
File file =File("C:\\Users\\PME16\\OneDrive\\Documents\\note.txt.txt");
file.readAsString().then((value) => print(value));
```

We can read the file using the function "**readAsString()"** which is an asynchronous function which will give us the file in the future when the file will be read which we can get using the "**then()"** function. We can also make this asynchronous function behave as synchronous by making the main function async and then using the keyword **await.** Then the function **readAsString()** will return the value in the string when the future will be achieved.

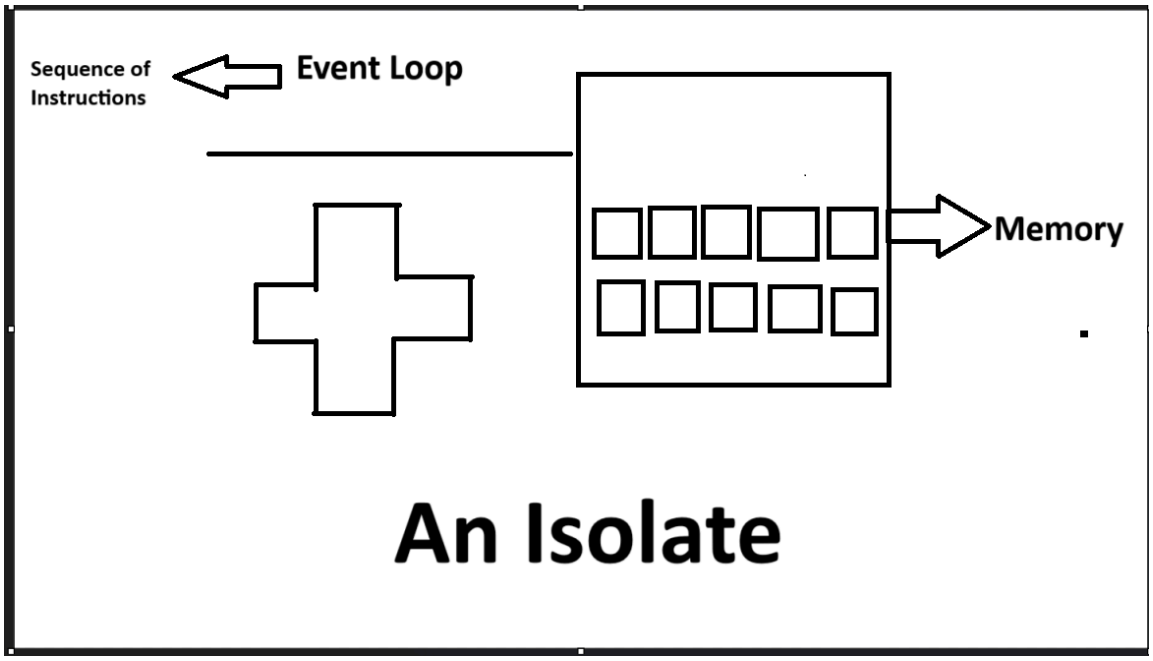
36. Isolates and Threads

(i) Threads

Threads are the execution flow of a program. Mainly our application runs in a single thread i.e the **main** thread. However, if there is a long running operation then that operation is conducted in a separate thread and this process is called **Multi Threading.** In threading some problems occur which may be difficult to handle. Firstly, All the threads share common memory and common resources. So if two threads want to share a common resource at the same time one of the two threads may wait for the other thread to execute and notify it when its execution is completed. Secondly, sometimes a **deadlock** can occur which may be impossible to solve. For example, suppose there are two functions **foo()** and **soo().** The function **foo()** is using resource **A** and the function **soo()** is using resource **B.** After some time the function **foo()** wants to use resource **B** and function **soo()** wants to use resource **A**. The function **foo()** cannot move to resource **B** because the function **soo()** is using it and it will remain on **wait()** and the function **soo()** cannot move to **A** because **foo()** is using it and it will remain on **wait** and a **deadlock** occurs. All the languages use threads for handling multiple long running operations.

(ii) Isolates

Dart uses **isolates** instead of threads. Every isolate has its own memory, resource and its own execution flow.So the problems that could occur in threads can no longer occur in isolates as they have their own memory and resources. Every isolate has its own event loop (sequence of instructions).



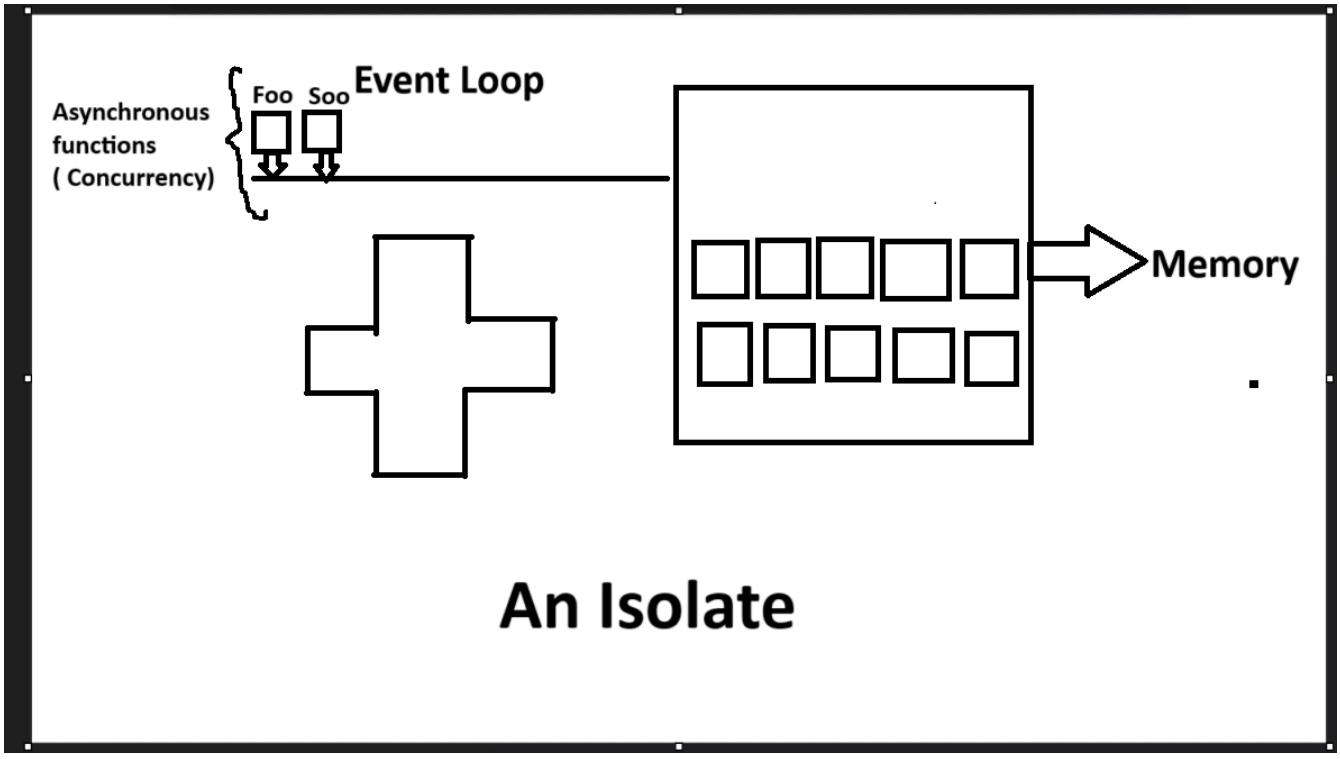
(iii) Inter process communication (IPC)

Inter process communication is the process of communication between two threads or isolates. In threads this process occurs by the function called **notify().** When we are downloading a big file it occurs in another thread and when the process is complete it notifies the other thread that the resource is modified and it can now use the resource.However in case of isolate the process is totally different. In isolates one isolates sends a **message** to another isolate so that the other isolate can use the resource.

(iv) Concurrency and Parallelism

(a) Concurrency

When two **asynchronous** functions share the same isolate the process is known as **concurrency.** Both the functions execute at the same time.The time is equally shared between the two functions and both of them are executed at the same time.Some part of function A is executed and some part of function B is executed and the process continues.

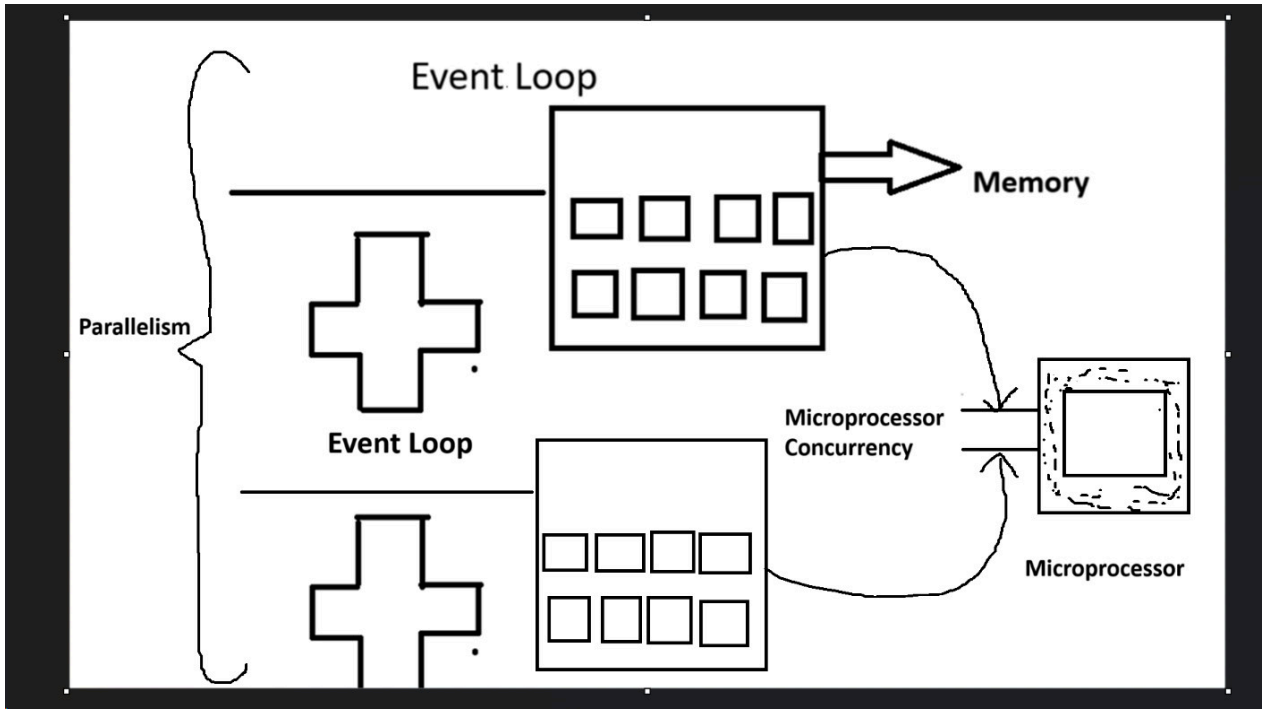


In the above example the two functions foo() and soo() share the same isolate and concurrency occurs. However, if the function foo() is a long running operation such that there are three nested loops inside it there will be no concurrency and the function foo() will be executed completely and after that the function soo() will be executed.If

both function should be executed by time sharing the function foo() will take a lot of time. So, here an asynchronous function will behave as synchronous.

(b) Parallelism

When two or more **isolates** are executed at the same time by the **processor** it is known as **parallelism**. Parallelism is also known as **concurrency of the processor**.



To solve the problem that occurred in concurrency, in isolates the function foo() will use the first isolate and the function soo() will use the main isolate even if the function foo() is a long running operation both will be executed at the same time.

```
Run | Debug
void main(List<String> arguments){

/*
number1 will work in another isolate whereas number2 will work in the main isolate and the processor will
execute both the isolates at the same time.
*/
Isolate.spawn(number1,1000);
number2(1000);
```

(v) Isolate groups

The concept of isolates created one problem that if every isolate had its own memory then isolates would not be efficient in terms of memory and they would consume a lot of space.So, Dart introduced isolate groups in which different isolates share a common memory. So if you use isolates as a group they will behave as a thread. So, problem solved :)

37. IPC Between Isolates

As we have discussed earlier, isolates use messages to communicate with each other. Let's see how this can be achieved. Suppose there is a long running operation such as downloading a file. So we will spawn another isolate for this purpose and this function will get executed in that isolate. But, how will the main isolate know that the file has been downloaded. The second isolate will send a message to the main isolate from time to time By using **ports**. Isolates can communicate either **one way** or **two ways**.To send and receive messages two classes "**Receive Port"** and "**Send Port"** are used.

(i) Receive Port

Receive port is the class which will receive the messages and we can obtain those messages using the function "**listen()**". Receive Port is a kind of Stream. If we go to the class of Receive port we will see that they have defined their own implementation of the class **Stream<T>.** The object of the receive port will constantly receive and sync the messages sent by the sending port.

```
//Receive port is a kind of Stream
ReceivePort receivePort=ReceivePort();

//receive port.listen will listen to messages sent by the sendPort.send from the download file function
receivePort.listen((message) {print(message);});
```

(ii) Send Port

Send port is the class which will send messages to the receive port which will be synced in the object of the Receive port. The method "**sendPort.send()**" is the method through which we will send messages to the receive port. The method can take anything in its parameter i.e "**sendPort.send(Object message)**" . **sendPort.send()** is the sink of the Receive port through which we will add the messages in the object of the Receiveport.

```
Future<void> downloadFile(SendPort sendPort)async{

  //receivePort.sendPort is the sink of receive port and receive port.listen will listen to the messages sent
sendPort.send('File is downloading');

await Future.delayed(const Duration(seconds: 3));

sendPort.send('File downloaded successfully');


return Future.value(null);
```

(a) One way Communication :

In one way communication only one isolate can send messages and the other can only receive the messages. Consider the example in which the isolate with the download file function will send messages and the main isolate will receive messages.

```
   //Receive port is a kind of Stream
  ReceivePort receivePort=ReceivePort();
  //receive port.listen will listen to messages sent by the sendPort.send from the download file function
  receivePort.listen((message) {
    print(message);
    });
  //we will send the function and receivePort.sendPort as the parameter of the function
  Isolate.spawn(downloadFile, receivePort.sendPort);
   await Future.delayed(const Duration(seconds: 30));
   Isolate.exit();
}
Future<void> downloadFile(SendPort sendPort)async{
  //receivePort.sendPort is the sink of receive port and receive port.listen will listen to the messages sent
sendPort.send('File is downloading');
await Future.delayed(const Duration(seconds: 3));
sendPort.send('File downloaded successfully');
return Future.value(null);

}
```

In the above example, First we will create the object of the receive port which will receive the messages and we will listen to the messages using the "**listen()"** method. After that we will spawn another isolate in which we will send the download function and "**receivePort.sendPort**" .Here receivePort.sendPort is the sink of the receive port through which we will send messages to the object of the receive port. The **receiveport.sendPort** will be received as the

parameter in the downloadFile function and will be used to send the messages.After the desired time is achieved we will exit the spawned isolate and the main isolate will be terminated by itself.

(b) Two way communication

In two way communication all the isolates can send and receive messages. Consider the example in which the isolate with the download file and the isolate main will both send and receive messages.

```
  //Receive port of main isolate
  ReceivePort receivePort=ReceivePort();
  //receive port.listen will listen to messages sent by the sendPort.send from the download file function
  receivePort.listen((message) {
    if(message is SendPort){
       message.send('Main is sendig a message');
    }else {
      print(message);
    }
    });
  //we will send the function and receivePort.sendPort as the parameter of the function
  Isolate.spawn(downloadFile, receivePort.sendPort);
   await Future.delayed(const Duration(seconds: 30));
   Isolate.exit();
}
//Download file function
Future<void> downloadFile(SendPort sendPort)async{
    //Receive port of the download file isolate
    ReceivePort receivePort=ReceivePort();
    //receive port will listen to the messages sent by the main isolate
    receivePort.listen((message) {
    print(message);
    });
    /*
    receivePort.sendPort is the sink of receive port and receive port.listen will listen to the messages sent
    first we will send the sink of the receive port of the isolate in which this function is executed
    */
    sendPort.send(receivePort.sendPort);
    sendPort.send('I am downloadind the file');
    await Future.delayed(const Duration(seconds: 3));
    sendPort.send('File downloaded successfully');
    return Future.value(null);
}
```

In the above example first we will create the receive port object in the main isolate which will receive the messages sent from the download file isolate and then we will create the receive port object in the download file function which will receive messages from the main isolate. In the download file function first we will send the sink of the receive

port that we created in the function to the main isolate so that it can send messages using this send port. In the **listen()** method that we called in the main isolate we will czech that if the send port is coming as a message we will send a message to the other isolate otherwise we will simply print the message. And in the **listen()** method that we called in the download file function we will simply print the message that is coming from the main isolate.