CƠ BẢN VỀ DART

Huỳnh Tuấn Anh
Khoa CNTT - ĐHNT

Nội dung

- Chương trình Hello World
- Các khái niệm cơ bản trong ngôn ngữ Dart
- Luồng điều khiển
- Hàm
- Lập trình hướng đối tượng trong Dart
  - Class
  - Constructor
  - Inheritance
  - Factories và named constructor
  - Enum
- Async

Chương trình "Hello World"

```
void main() {
  helloDart("Flutter");
}

void helloDart(String name) {
  print("Hello $name");
}
```


Các thư viện trong Dart

- Dart SDK bao gồm nhiều thư viện. Thư viện được nạp mặc định vào chương trình là 'dart : core'. Các thư viện khác khi sử dụng phải khai báo thông qua từ khóa *import*.
  - VD: import 'dart : io'
- Một số thư viện thông dụng: *dart:html, dart:async, dart:math...*

Các khái niệm cơ bản

- Dart là một ngôn hướng đối tượng và hỗ trợ đơn thừa kế.
- Trong Dart, mọi thứ đều là object, mọi object đều là một thể hiện của một lớp. Mọi đối tượng đều kế thừa từ lớp Object. Các con số cũng là đối tượng chứ không phải thuộc các kiểu dữ liệu nguyên thủy.
- Dart là một ngôn ngữ định kiểu. VD: Không thể trả về một con số từ một hàm được khai báo kiểu trả về là String.
- Dart hỗ trợ hàm và biến được khai báo bên ngoài lớp (top-level functions và top-level variables), chúng được xem như là thành viên của thư viện

Kiểu trong Dart

- Chỉ định kiểu khi khai báo biến:
  - VD: String name;
  - int age = 30;
- Kiểu dữ liệu phức hợp:
  - Khi sử dụng các cấu trúc dữ liệu như List hay Map, sử dụng <> để định nghĩa kiểu dữ liệu cho mỗi thành viên.
  - VD:

```
List<String> names;
List<int> ages;
Map<String, int> people;
```

A list of strings

A list of integers

A map whose keys are strings and values are integers

Kiểu trả về của hàm

- Khi định nghĩa hàm, cần phải chỉ ra kiểu trả về của hàm
- VD:

```
int addNum(int x, int y)
{
  return x + y;
}

int addNum(int x, int y) => x+y;
```


Các kiểu dữ dữ liệu cơ bản

- package dart:core
  - https://api.dart.dev/stable/2.10.3/dart-core/dart-core-library.html
- Kiểu số và kiểu Booleans

```
int meaningOfLife = 42;
double valueOfPi  = 3.141592;
bool visible      = true;
```

- Kiểu chuỗi
- Collections
- Date và Time
- URI

Kiểu dynamic

- Khi sử dụng từ khóa dynamic để khai báo biến, trình biên dịch sẽ chấp nhận mọi kiểu dữ liệu cho biến này.
- VD 1:
  - dynamic number = “11” ; // kiểu String
  - number = 11; // kiểu int
  - var number2 = “11”; // Kiểu String
  - number2 = 11; // Lỗi
- Kiểu dữ liệu dynamic tỏ ra hữu dụng khi sử dụng với dữ liệu Map khi mỗi thành viên của Map là một cặp key-value trong dữ liệu JSON.
  - VD: Map<String, dynamic> json;
  - Khai báo Map<String, var> json; // *báo lỗi???*

Comment

```
// Inline comments
/*
Blocks of comments. It's not convention to use
block comments in Dart.
*/
/// Documentation
///
/// This is what you should use to document
your classes.
```


Biến và phép gán

- Khi khai báo biến nhưng không gán giá trị, biến sẽ nhận giá trị null (null cũng là một object).
  - VD: String name; // name nhận giá trị null.
  - Tất cả các biến đều có thể được gán giá trị null
- Từ khóa final, const: Dùng để khai báo một biến có giá trị không thể thay đổi khi chạy chương trình. Các biến khi khai báo với final hay const đều phải khởi tạo giá trị lúc khai báo.
  - VD1: final String name = “Smith”;
  - VD2: const String name = “Smith”;

Từ khóa final, const

- Khác nhau giữa final và const:
  - biến được khai báo const phải được xác định giá trị khi compile.
  - biến được khai báo với từ khóa final chỉ được gán giá trị một lần
  - VD1: const String name = ‘Nora $lastname’; // Lỗi???
  - VD2: final String name = ‘Nora $lastname’; // Sử dụng được???

Các toán tử: Toán tử số học

OPERATOR	DESCRIPTION	SAMPLE CODE
+	Add	7 + 3 = 10
-	Subtract	7 - 3 = 4
*	Multiply	7 * 3 = 21
/	Divide	7 / 3 = 2.33

Toán tử so sánh

OPERATOR	DESCRIPTION	SAMPLE CODE
==	Equal	7 == 3 = false
!=	Not equal	7 != 3 = true
>	Greater than	7 > 3 = true
<	Less than	7 < 3 = false
>=	Greater than or equal to	7 >= 3 = true 4 >= 4 true
<=	Less than or equal to	7<= 3 = false 4 <= 4 = true

Toán tử kiểm tra kiểu

OPERATOR	DESCRIPTION	SAMPLE CODE
as	Typecast like import library prefixes.	import 'travelpoints .dart' as travel;
is	If the object contains the specified type, it evaluates to true.	if (points is Places) = true
is!	If the object contains the specified type, it evaluates to false (not usually used).	if (points is! Places) = false

as: còn được dùng để chuyển một object về đúng kiểu của nó.
VD: Person p = new Student("An", 20);
Student s = p as Student;

Toán tử logic

OPERATOR	DESCRIPTION	SAMPLE CODE
!	! is a logical 'not'. Returns the opposite value of the variable/expression.	if (!(7 > 3)) = false
&&	&& is a logical 'and'. Returns true if the values of the variable/expression are all true.	if ((7 > 3) && (3 < 7)) = true if ((7 > 3) && (3 > 7)) = false
||	|| is a logical 'or'. Returns true if at least one value of the variable/expression is true.	if ((7 > 3) || (3 > 7)) = true if ((7 < 3) || (3 > 7)) = false

Toán tử điều kiện

OPERATOR	DESCRIPTION	SAMPLE CODE
`condition ? value1 : value2`	If the condition evaluates to `true`, it returns `value1`. If the condition evaluates to `false`, it returns `value2`. The value can also be obtained by calling methods.	`(7 > 3) ? true : false = true` `(7 < 3) ? true : false = false`

- Cascade notation (..)

OPERATOR	DESCRIPTION	SAMPLE CODE

`..`	The cascade notation is represented by double dots (..) and allows you to make a sequence of operations on the same object.	`Matrix4.identity()`
		`..scale(1.0, 1.0)`
		`..translate(30, 30);`

Các toán tử

- ~/: toán tử chia số nguyên.
- as: từ khóa dùng để chuyển một object về đúng kiểu của nó.
  - VD: Person p = new Student("An", 20);
  - Student s = p as Student;
- is và is!: kiểm tra kiểu của một đối tượng.
  - VD: p is Person; p is! Person
- ?. : Kiểm tra null trước khi truy cập một thuộc tính của đối tượng:

```
this.userAge = user?.age;   ⟺   if (user != null) {
                                   this.userAge = user.age;
                                 }
```

Nếu user = null thì userAge không được gán giá trị

Null safety

- Dart version >=2.12
- Trong Flutter, khai báo trong pubspec.yaml như sau

```
environment:
  sdk: ">=2.12.0 <3.0.0"
```

- Thêm dấu ? phía sau tên kiểu dữ liệu khi khai báo biến để biểu thị rằng biến đó có thể nhận giá trị null
  - Ví dụ: String? ten;
- Khi sử dụng biến đã khai báo khả null, để bỏ qua kiểm tra null, thêm dấu ! phía sau biến đó:
  - Ví dụ: var x = ten!; // Nếu ten có giá trị null sẽ phát sinh lỗi run-time

Một số toán tử Null safety

- Toán tử ??:
  - x ??= 5;
  - Nếu x có giá trị null, gán 5 cho x
  - Nếu x khác null, giữ nguyên giá trị của x
- Ví dụ

```
String? email;
// Nếu email bằng null thì thực hiện phép gán
email ??= "abc@gmail.com";
// Luôn thực hiện phép gán
email = "abc@gmail.com";
//Nếu email bằng null thì gán chuỗi "Không có email" cho x
String x = email?? "Không có email";
```

Null safety - Bang operator

- Khai báo biến có thể nhận giá trị null với toán tử ?
- Ví dụ:
  - String? phone; // biến phone có thể nhận giá trị null
  - String s; // biến s không nhận giá trị null
  - Câu lệnh: s = phone --> lỗi cú pháp
  - Sử dụng Bang operator: s = phone! ; --> không bị lỗi cú pháp, tuy nhiên khi sử dụng nếu phone có giá trị null --> lỗi lúc run-time
- Sử dụng bang operator để đảm bảo (từ lập trình viên) với dart rằng một biến Nullable không có giá trị null
- Có thể sử dụng: *s = phone?? "Không có số điện thoại"* --> không bị lỗi cú pháp lẫn lỗi lúc run-time

Từ khóa late

- Một thuộc tính nếu không được khởi tạo theo kiểu tham số vị trí thì phải được khai báo Nullable.
- Trong trường hợp không muốn khởi tạo thuộc tính trong hàm khởi tạo và chắc chắn rằng khi sử dụng thuộc tính sẽ không nhận giá trị null ta sử dụng từ khóa late khi khai báo thuộc tính trong một class.
- Ví dụ:
  - late int count;
  - late String name.
- Việc sử dụng từ khóa late kèm theo sự đảm bảo từ phía LTV là biến không có giá trị null khi sử dụng sẽ đỡ phải sử dụng các toán tử kiểm tra null sau này.

Toán tử spread

- Toán tử spread (...): Thêm tất cả các phần tử của một danh sách vào một danh sách khác:

```
var list = [1, 2, 3];
var list2 = [0, ...list];
assert(list2.length == 4);
```

- Null-aware spread operator: Tương tự như spread nhưng

```
List<int>? ls1 = [1,2,3];
var ls2 = [4,5,6, ...?ls1];
print(ls2);// [4, 5, 6, 1, 2, 3]
```


Luồng điều khiển

- if, else
- switch, case
- Vòng lặp
  - Standard for
  - for-in
  - forEach
  - while
  - do while
- break và continue

Vòng lặp

for:

```
for(var i=0; i<10; i++) {
  print(i);
}
```

for-in:

```
List<String> pets = ["Tom", "Jerry", "Poo"];
for(var pet in pets) {
  print(pet);
}
```

forEach:

```
List<String> pets = ["Tom", "Jerry", "Poo"];
pets.forEach((pet) => print(pet));
```


Hàm - Function

- Cấu trúc của function
- Parameters
  - Positional parameters: Tham số bắt buộc, phải nhập đúng vị trí
  - Named parameters: Tùy chọn, không bắt buộc phải nhập khi sử dụng.
  - Positional optional parameters: Tùy chọn không bắt buộc phải nhập khi sử dụng.
- Default parameter values: Định nghĩa giá trị mặc định cho tham số thông qua toán tử = khi định nghĩa hàm.
- Hàm ẩn danh (Anonymous functions)

functions

Cấu trúc của hàm trong Dart:

```
String makeGreeting(String name) {
  return 'Hello, $name';
}
```

Function signature

Return type

Cú pháp ngắn gọn khi thân hàm chỉ một dòng:

```
String makeGreeting(String name) => 'Hello, $name';
```


Parameter

- Positional parameters

```
void debugger(String message, int lineNum) {
  // ...
}

debugger('A bug!', 55);
```

- Named Parameters

Named parameters

Định nghĩa hàm:

```
void debugger({String message, int lineNum}) {
```

Gọi hàm:

```
debugger(message: 'A bug!', lineNum: 44);
```

Parameters

- Positional optional parameters:
-

```
int addSomeNums(int x, int y, [int z]) {
  int sum = x + y;
  if (z != null) {
    sum += z;
  }
  return sum;
}
```

optinal parameter

The third parameter is optional, so you don't have to pass in anything.

```
addSomeNums(5, 4)
addSomeNums(5, 4, 3)
```

You can pass in a third argument, since you've defined an optional parameter.

Anonymous functions

- Hầu hết các hàm đều có tên để có thể gọi để sử dụng sau này.
- Hàm ẩn danh không có tên, đôi khi được gọi là *lambda* or *closure*
- Có thể gán một hàm ẩn danh cho một biến. Do đó, cũng có thể thêm vào hoặc loại bỏ một hàm ẩn danh ra khỏi một tập hợp.
- Các định nghĩa hàm ẩn danh cũng giống như hàm có tên thông thường, tuy nhiên ta bỏ qua khai báo phần tên của hàm

```
([[Type] param1[, …]]) {
  codeBlock;
};
```


Anonymous functions

- Ví dụ sau đây định nghĩa một hàm ẩn danh với một tham số không định kiểu, item. Hàm này chỉ đơn giản là nhận mỗi giá trị trong danh sách và in giá trị đó trên màn hình.
  - Kiểu dữ liệu của tham số sẽ được suy ra từ kiểu dữ liệu của mỗi phần tử trong danh sách

```
void main() {
  var list = ['apples', 'bananas', 'oranges'];
  list.forEach((item) {
    print('${list.indexOf(item)}: $item');
  });
}
```


Cú pháp rút gọn của hàm

- Được sử dụng khi hàm chỉ có một câu lệnh. Cú pháp rút gọn thường được dùng khi viết các hàm ẩn danh.
- Ví dụ:

```
int binhPhuong(int x) => x*x;
```

```
void main() {
  var list = ['apples', 'bananas', 'oranges'];
  list.forEach((item) =>
      print('${list.indexOf(item)}: $item'));
}
```

Lập trình hướng đối tượng trong Dart

- class
- constructor
- inheritance
- factories và named constructor
- enum

class

```
class Cat {
    String name;
    String color;
}
```

instance

```
Cat nora = new Cat();
nora.name = 'Nora';
nora.color = 'Orange';

Cat ruby = Cat();
nora.name = 'Ruby';
nora.color = 'Grey';
```

- ✓ Nếu không viết constructor cho class thì constructor mặc định không có tham số sẽ được tạo ra.
- ✓ Từ khóa new để tạo một thể hiện của class là tùy chọn

constructor

```
class Animal {
  String name;
  String type;

  Animal(String name, String type) {
    this.name = name;
    this.type = type;
  }
}
```

Declares properties of this class (they are null to start)

Default constructor

Passes in arguments to the constructor

```
class Animal {
  String name, type;

  Animal(this.name, this.type);
}
```

Automatically assigns arguments to properties with the same name

factory và named constructor

- Named constructor: Cho phép tạo ra các constructor với các tên có ý nghĩa.

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

factory và named constructor

- factory: Được sử dụng khi muốn tạo constructor không chỉ để tạo ra một instance mới của class mà có thể một instance từ cache, hoặc một subclass của nó.

```
class Customer{
  String name, location;
  int age;
  Customer(this.name, this.age, this.location);
  static final Customer origin = Customer("", 30, "");
  factory Customer.create() => origin;
  @override
  String toString() {
    return 'Customer{name: $name, location: $location, age: '
        '$age}';
  }
}
```

```
void main()
{
  Customer customer = Customer.create();
  print(customer);
}
```


Inheritance: Single inheritance

```
class Spacecraft {
  String name;
  DateTime launchDate;
  // Constructor, with syntactic sugar for assignment to members.
  Spacecraft(this.name, this.launchDate) {
    // Initialization code goes here.
  }
  // Named constructor that forwards to the default one.
  Spacecraft.unlaunched(String name) : this(name, null);

  int get launchYear =>
      launchDate?.year; // read-only non-final property
  void describe() {//...}
}
```

```
class Orbiter extends Spacecraft {
  double altitude;
  Orbiter(String name, DateTime launchDate, this.altitude)
      : super(name, launchDate);
}
```


Mixins – Thêm các feature vào một lớp

- Mixin là một cách sử dụng lại mã lệnh trong nhiều phân cấp lớp.
- Hai cách thực hiện mixin
  - Định nghĩa class mà không có constructor; *hoặc*
  - Thay thế từ khóa `class` bằng từ khóa `mixin`
- Thành phần mixin được bổ sung vào một lớp bằng từ khóa with
- Ví dụ:

Mixin, Ví dụ:

```
mixin Piloted { // hoặc class
  int astronauts = 1;
  void describeCrew() {
    print('Number of astronauts: $astronauts');
  }
}
```

```
class PilotedCraft extends Spacecraft with Piloted {
  PilotedCraft(String name, DateTime launchDate) : super(name, launchDate);
  // ...
}
```

Lớp PilotedCraft bây giờ đã được bổ sung thêm trường astronauts và phương thức describeCrew()

Mixin – từ khóa on

- Hạn chế các kiểu (type) có thể sử dụng mixin,
- Muốn thành phần mixin có thể sử dụng một số các phương thức, thuộc tính mà nó không định nghĩa.
- Ví dụ:

```
class Person{
  String firstName, lastName;
  int age;
  Person(this.firstName, this.lastName, this.age);

  String getInfo(){
    return "$firstName $lastName";
  }
}
```


mixin - on

- Chỉ có các lớp con của Person mới có thể sử dụng mixin `PersonWithAge`

```
mixin PersonWithAge on Person{
  String getFullInfo(){
    return "Tên: ${getInfo()} \nTuổi: $age";
  }
}
```

- Lớp SinhVien thừa kế lớp Person và có các phương thức của lớp PersonWithAge

```
class Student extends Person with PersonWithAge{
  String course;
  SinhVien(fn, ln, age, this.course):super(fn, ln, age);
  printInfoSV(){
    print(getFullInfo());
    print("Khóa học: $course");
  }
}
```


interface

- Mỗi class trong Dart ngầm định định nghĩa một interface chứa tất cả các thành viên của class đó và mọi interface mà nó implement.
  - Có thể implement một lớp bất kỳ
- Nếu muốn tạo class A hỗ trợ các API của class B mà không thừa kế các cài đặt phương thức đã có của class B, class A nên implement interface B.
- Có thể sử dụng abstract class để định nghĩa một interface.

```
class MockSpaceship implements Spacecraft {
  @override
  DateTime launchDate;
  @override
  String name;
  @override
  void describe() {
    // TODO: implement describe
  }
  @override
  // TODO: implement launchYear
  int get launchYear => throw UnimplementedError();
}
```


Asynchronus support

- Dart cung cấp các thư viện hay các hàm trả về hai kiểu dữ liệu Stream và Future. Các hàm này sẽ return sau khi *thiết lập* một hoạt động có thể tiêu tốn nhiều thời gian (thường là các hoạt động I/O) mà không phải đợi cho đến khi hoạt động đó hoàn thành
- Khi bạn cần một kết quả Future đã hoàn thành, bạn có hai lựa chọn:
  - Sử dụng từ khóa async và await
  - Sử dụng Future API
- Một số trường hợp lập trình bất đồng bộ:
  - Lấy dữ liệu từ server.
  - Ghi vào database.
  - Đọc nội dung từ file.
  - Sử dụng các thư viện hỗ trợ lập trình bất đồng bộ.

Định nghĩa hàm bất đồng bộ: async/async* và await

- Hai từ khóa *async/async** và *await* cung cấp một cách khai báo để định nghĩa hàm bất đồng bộ.
  - Để định nghĩa một hàm bất đồng bộ ta thêm từ khóa async trước thân hàm.
  - Từ khóa await chỉ làm việc trong hàm async. Từ khóa await được sử dụng để lấy chờ lấy một kết quả xử lý bất đồng bộ trong thân hàm async
  - Hàm được khai báo cùng với từ khóa async luôn trả về kiểu dữ liệu được *gói* bởi kiểu dữ liệu Future và sử dụng từ khóa return để trả về giá trị của hàm.
  - Hàm được khai báo cùng với từ khóa async* luôn trả về kiểu dữ liệu Stream và sử dụng từ khóa yield để trả về giá trị thay cho return.
  - Sử dụng từ khóa await để đợi một giá trị được phát ra bởi một Stream.

return Future

- Sau khi thiết lập một hoạt động bất đồng bộ (được xác định bởi từ khóa await), hàm BĐB trả về một đối tượng Future.
- Future là kết quả của hoạt động bất đồng bộ và có 2 trạng thái là **chưa hoàn thành** và **hoàn thành**:
  - Chưa hoàn thành: Khi chúng ta gọi một hoạt động bất đồng bộ, nó sẽ trả về một Future chưa hoàn thành, đây là trạng thái của Future trước khi trả về kết quả.
  - Hoàn thành: Khi hoạt động bất đồng bộ thực hiện xong thì Future sẽ ở trạng thái hoàn thành, Future có thể hoàn thành với một giá trị hoặc là một lỗi.
  - Future hoàn thành với một giá trị thì nó có thể là Future<T> với giá trị có kiểu T, Future<String> với giá trị kiểu String, hoặc là một giá trị void với Future<void>.
  - Hoàn thành với một error: nếu một hoạt động bất đồng bộ được thực hiện bởi một hàm thất bại vì bất kỳ lý do gì thì Future hoàn thành với một error

Asynchronous functions

```
Future<String> fetchUserOrder() async{
    String st = await Future.delayed(
        Duration(seconds: 2),
        () => 'Large Late');
    return st;
}
```

```
Future<String> createOrderMessage () async {
  var order =await fetchUserOrder();
  return 'Your order is $order';
}
```

```
void main()async{
  print('Fetching user order...');
  print(await createOrderMessage());
}
```


then, whenComplated, cachError, onError

- Có thể sử dụng các phương thức trên khi lập trình bất đồng bộ để thay thế async, await.
- Future<T> whenComplete(FutureOr<**void**> action()) : Đăng ký một action sẽ được gọi khi future hoàn tất bất chấp có lỗi xảy ra hay không.
  - Có thể xem whenComplete tương đương với khối "finally" trong cơ chế bắt lỗi try-catch

```
void main() async {
  var value =
      await waitTask().whenComplete(() => print('do something here'));
  // Prints "do something here" after waitTask() completed.
  print(value); // Prints "done"
}

Future<String> waitTask() {
  Future.delayed(const Duration(seconds: 5));
  return Future.value('done');
}
// Outputs: 'do some work here' after waitTask is completed.
```

- Future<R> then<R>(FutureOr<R> onValue(T value), {Function? onError}):
  - onValue: callback sẽ được gọi với dữ liệu của future khi hoàn tất
  - onError: Nếu được cung cấp callback này sẽ được gọi khi future hoàn tất cùng với lỗi

then, whenComplated, cachError, onError

- Future<T> catchError(Function onError, {bool test(Object error)?}): Xử lý lỗi phát sinh bởi Future.
  - Tương đương với khối "catch" trong cơ chế bắt lỗi try-catch

```
someFuture().then((value) {
  print('Future finished successfully i.e. without error');
}).catchError((error) {
  print('Future finished with error');
}).whenComplete(() {
  print('Either of then or catchError has run at this point');
});
```


Asynchronous programming: streams

- Stream cung cấp một chuỗi dữ liệu không đồng bộ.
- Chuỗi dữ liệu bao gồm các sự kiện do người dùng tạo ra và dữ liệu đọc từ các tệp.
- Bạn có thể xử lý luồng bằng cách sử dụng await hoặc listen() từ Stream API.
- Stream cung cấp một cách để xử lý lỗi
- Có hai loại luồng: single subscription or broadcast.
- Lập trình bất đồng bộ trong Dart được đặc trưng bởi hai lớp Future và Stream

Stream, async, await

- Stream có thể được tạo theo nhiều cách, nhưng chúng có thể được sử dụng theo cùng một cách *asynchronous for loop* (vòng lặp for bất đồng bộ – thường được gọi là await for) lặp qua các sự kiện của một stream như vòng lặp for loop lặp qua một iterable.
- Từ khóa *await* chỉ làm việc trong hàm *async*
- **async**: Bạn có thể sử dụng từ khóa *async* trước thân hàm bất đồng bộ.
- **async function**: là một function được đánh dấu bởi từ khóa *async*.
- **await**: bạn có thể sử dụng từ khóa *await* để lấy kết quả từ một việc bất đồng bộ. Từ khóa *await* chỉ được sử dụng với hàm *async*.

Nhận các sự kiện stream

```
Future<int> sumStream(Stream<int> stream) async {
  var sum=0;
  await for(var value in stream)
    sum += value;
  return sum;
}
```

```
Stream<int> countStream(int to) async*{
  for(int i=0; i<=to; i++) {
    yield i;
  }
}
```

```
void main() async{
  var stream = countStream(10);
  var sum = await sumStream(stream);
  print(sum);
}
```


Sử dụng Future API: await, then

```
const oneSecond = Duration(seconds: 1);
Future<void> printWithDelay(String message) async {
  await Future.delayed(oneSecond);
  print(message);
}
```

```
const oneSecond = Duration(seconds: 1);
Future<void> printWithDelay(String message) {
  return Future.delayed(oneSecond).then((_) {
    print(message);
  });
}
```


Sử dụng Future API: await, then

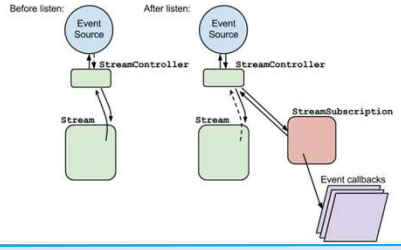
```
Future<int> sumStream(Stream<int> stream) async {
  var sum=0;
  await for(var value in stream)
    sum += value;
  return sum;
}
```

```
Stream<int> countStream(int to) async*{
  for(int i=0; i<=to; i++) {
    yield i;
  }
}
```

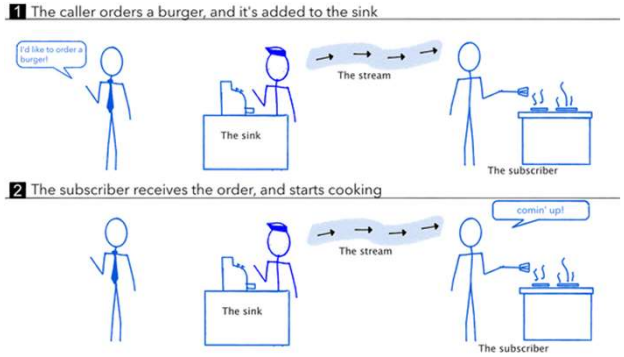
```
void main() async{
  var stream = countStream(10);
  sumStream(stream)
      .then((value) => print(value));
}
```


Stream Controller

- Stream Controller được sử dụng để:
  - Cung cấp một Stream mới
  - Thêm các sự kiện vào stream bất kỳ thời điểm nào và từ bất kỳ đâu
  - Cung cấp các logic để xử lý phía người nghe (listener) và tạm dừng stream



Stream



Ví dụ

```
class MyStream<T>{
  StreamController<T> _streamController = StreamController();
  Stream<T> get stream => _streamController.stream;
  void addEvent(T event){
    _streamController.sink.add(event);
  }
  void dispose(){
    _streamController.close();
  }
}
```

```
void main(){
  var myStream = MyStream<String>();
  StreamSubscription<String> subscriber = myStream.stream.listen((
    event){
      print("Please, your $event is done!");
  });
  myStream.addEvent("Hamburger");
}
```


Bài tập

- Sinh viên, làm các ví dụ trong các Slide để làm quen với các cú pháp trong ngôn ngữ Dart.
- Nâng cao kỹ năng về ngôn ngữ Dart thông qua các bài tập trên trang: Dart on Exercism: https://exercism.org/tracks/dart/exercises