LẬP TRÌNH BẤT ĐỒNG BỘ

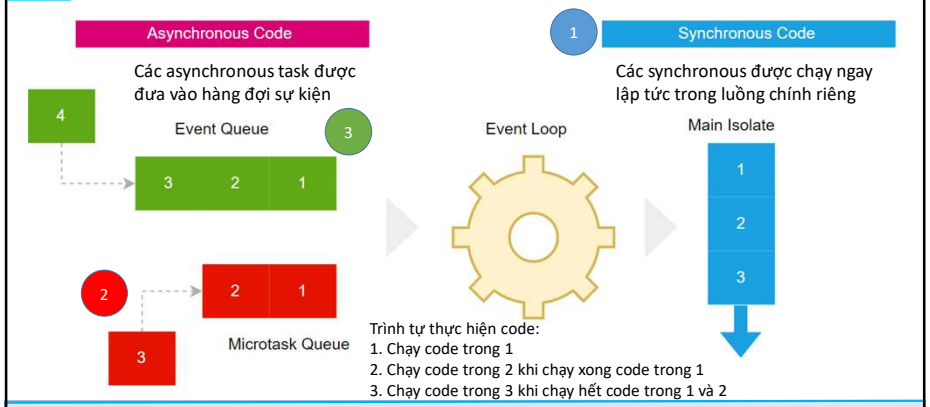
Lập trình bất đồng bộ (Asynchronous)

- Cung cấp khả năng tạo các tác vụ thực thi độc lập nhau, không nhất thiết phải chạy tuần tự.
- Tương tự như C#, Dart cũng cung cấp 2 từ khóa async/async*, await giúp lập trình viên lập trình bất đồng bộ một cách dễ dàng.
- Trong xây dựng ứng dụng Flutter, để nâng cao trải nghiệm người dùng, một số thư viện bắt buộc lập trình viên phải lập trình bất đồng bộ.

Dart Event Loop

- Dart là một ngôn ngữ đơn luồng có nghĩa là nó sử dụng một luồng cô lập duy nhất để thực hiện code.
- Trong thực tế, có thể có nhiều task có thể chạy bất đồng bộ. Ví dụ:
  - Tải ảnh về từ máy chủ
  - Kết nối với CSDL
  - Ghi dữ liệu vào tệp tin
  - ....

Dart Event Loop



Future

Future

- Future<T>: Lớp để tạo các đối tượng future. Trong thực hành, chúng ta thường sử dụng các đối tượng future hơn là tạo ra chúng.
  - Các đối tượng future thường là kết quả trả về khi thực hiện lời gọi hàm bất đồng bộ.
  - Hàm/phương thức bất đồng bộ là hàm/phương thức định nghĩa với từ khóa async/async*.
- future trong dart là một đối tượng biểu diễn kết quả của một hoạt động bất đồng bộ. Một future object sẽ trả về một giá trị sau một khoảng thời gian thực hiện tác vụ bất đồng bộ. Một future có hai trạng thái:
  - uncomplete/chưa hoàn thành
  - complete/hoàn thành: Khi một future hoàn thành, nó sẽ có hai khả năng
    - Hoàn thành với một giá trị
    - Bị lỗi với một lỗi.

Ví dụ

Trả về giá trị future chưa hoàn thành

Trả về giá trị future hoàn thành

```
32  Future<int> getNumber() async{
33    int num = await Future.delayed(
34      Duration(seconds: 5),
35      () => 3,
36    );  // Future.delayed
37    return num;
38  }
```


Ví dụ

```
var future = Future<int>.delayed(
  Duration(seconds: 1),
  () => 2
);
```

- Cho biết kiểu dữ liệu của biến future?
- Phương thức delayed còn được gọi là phương thức gì của lớp Future<T>

Sử dụng giá trị của future khi future hoàn thành

- Khi một future hoàn thành, có hai cách để lấy giá trị của nó:
  - Sử dụng các callback được cung cấp bởi lớp Future.
  - Sử dụng từ khóa async, await

Sử dụng các callback để lấy các giá trị

- then(): Được gọi khi future hoàn thành với giá trị. Trong then có 2 callback:
  - (value){...}: Được gọi để xử lý giá trị future hoàn thành.
  - onError: (error){}: là một tùy chọn, được gọi khi future hoàn thành với giá trị lỗi.
  - Nếu đăng ký onError trong then thì khi future hoàn thành với giá trị lỗi thì giá trị lỗi này sẽ do onError xử lý, catchError chỉ xử lý lỗi phát sinh khi thực hiện then.
- catchError(): Được gọi khi future hoàn thành với giá trị lỗi (nếu onError không được đăng ký trong then) hoặc khi thực hiện then bị lỗi. Trong catchError có callback để xử lý lỗi.
  - (error){...}: error chính là lỗi trong trường hợp này.
- whenComplete(): Phương thức cung cấp callback không có tham số để xử lý trong trường hợp future hoàn tất (có thể hoàn tất với giá trị hay gặp lỗi).

- Cho biết kết quả trên màn hình

```
void main(){
  Future.delayed(Duration(seconds: 5), () => 3,)
    .then(
      (value) {
        print(value);
      },
      onError:(error){
        print(error);
      }
    ).catchError((error){
      print(error);
    });
}
```


Sử dụng từ khóa async và await

- Các từ khóa async và await cung cấp cung cấp một cách khai báo để định nghĩa các hàm bất đồng bộ.
- Qui tắc:
  - Đặt từ khóa async trước thân hàm để làm cho hàm trở thành bất đồng bộ.
  - Từ khóa await chỉ được dùng bên trong hàm async

Ví dụ

- Cho biết kết quả trên màn hình của đoạn mã sau

```
23 void main()async{
24   print(1);
25   var result = await Future<int>.delayed(
26     Duration(seconds: 2),
27     () => 2,
28   );
29   print(3);
30   print(result);
31 }
```


Xử lý ngoại lệ

- Sử dụng các hàm bất đồng bộ

```
void main(){
  Future<int>.delayed(
    Duration(seconds: 2),
    () => 2,
  ).then( // try
    (value) => print(value),
  ).catchError( // catch
    (error)=>print(error)
  ).whenComplete( // finally
    () => print("Kết thúc"),);
}
```

- Sử dụng async, await

```
void main()async{
  try{
    var val = await Future<int>
      .delayed(
      Duration(seconds: 2),
      () => 2,
    );
    print(val);
  }catch(e){
    print(e);
  }finally{
    print("Kết thúc");
  }
}
```


Stream

Giới thiệu về Stream

- Một future biểu diễn một giá trị đơn được trả về từ một hoạt động async.
- Một stream có thể coi như một danh sách các future. Nó biểu diễn nhiều giá trị sẽ được trả về trong tương lai.
- Lớp Stream<T> được sử dụng để tạo các stream. Có thể tạo stream cho bất kỳ đối tượng nào.
- Stream thường được sử dụng trong các trường hợp:
  - Đọc dữ liệu từ một tệp lớn theo khối.
  - Tải các tài nguyên về từ máy chủ.
  - Lắng nghe các request đến server
- Trong thực hành, chúng ta thường sử dụng stream từ thư viện hơn là tạo chúng từ đầu.

Đọc dữ liệu từ stream sử dụng callback

```
void main(){
  File file = File("readme.txt");
  Stream<List<int>> stream = file.openRead();
  StreamSubscription<List<int>>? subscription;
  subscription= stream.listen((data){
      print(data.length);
    },);
}
```

- file.openRead(): đọc file theo chunk (khối) và trả về một đối tượng stream.
- Stream<List<int>>: Định kỳ trả về một danh sách các số nguyên biểu diễn cho các giá trị byte.
- Gọi phương thức listen trên stream để đăng ký việc nhận thông báo khi có dữ liệu mới đến trong stream

Đọc dữ liệu từ stream sử dụng vòng lặp for không đồng bộ

```
Future<void> main() async {
  var file = File('big_lorem_ipsum.txt');
  var stream = file.openRead();
  await for (var data in stream) {
    print(data.length);
  }
}
```

- await for: tạm dừng vòng lặp cho đến khi có dữ liệu mới đến stream.
- Do sử dụng await nên cần phải có từ khóa async đặt trước thân hàm

Xử lý lỗi với stream với callback

- Xử lý với callback

```
void main(){
  File file = File("readme.txt");
  Stream<List<int>> stream = file.openRead();
  StreamSubscription<List<int>>? subscription;
  subscription= stream.listen((data){
      print(data.length);
    },
    onError: (error) => print(error),
    onDone: ()=> print("Done!"),
    cancelOnError: true,
  );
}
```


Xử lý lỗi với try, catch

```
Future<void> main() async {
  var file = File('readme.txt');
  try {
    var stream = file.openRead();
    await for (var data in stream) {
      print(data.length);
    }
  } catch (error) {
    print(error);
  }
}
```

Hủy bỏ một stream

```
void main(){
  File file = File("readme.txt");
  Stream<List<int>> stream = file.openRead();
  StreamSubscription<String>? subscription;
  subscription= stream.transform(utf8.decoder).listen((data){
      print(data);
      subscription?.cancel();// Hủy bỏ đăng ký stream
    },
    onError: (error) => print(error),
    onDone: ()=> print("Done!"),
    cancelOnError: true,
  ) ;
}
```


Transforming a stream

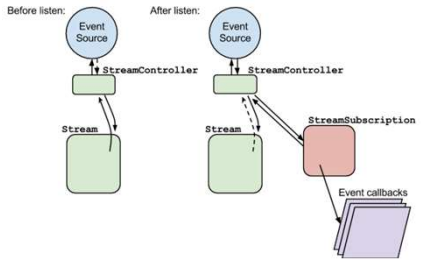
- Có thể chuyển đổi nội dung của stream bằng cách sử dụng phương thức transform hoặc map(đã được học):

```
void main(){
  File file = File("readme.txt");
  Stream<List<int>> stream = file.openRead();
  StreamSubscription<String>? subscription;
  subscription= stream.transform(utf8.decoder).listen((data){
      print(data);
    },
    onError: (error) => print(error),
    onDone: ()=> print("Done!"),
    cancelOnError: true,
  ) ;
}
```

Chuyển bytes thành String

Stream Controller

- Thành phần kết nối Event Source và Stream
- 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



Ví dụ về Stream Controller

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

```
void main(){
  MyStream<String> myStream = MyStream();
  myStream.stream.listen((event) {
    print(event);
  });
  myStream.addEvent("Hello World");
}
```

Sử dụng GetX với Stream và Future

- Để hiển thị một Stream ta sử dụng StreamBuilder (đã được thực hành)
- Để hiển thị một future ta sử dụng FutureBuilder (đã được thực hành)
- GetX với dữ liệu Stream: Nguyên tắc
  1. Viết Controller quản lý trạng thái ứng dụng có kiểu dữ liệu Rx{Type}.
  2. Gắn kết (Bind) trạng thái ứng dụng với Stream thông qua phương thức bindStream trong phương thức onInit hay onReady tùy theo yêu cầu khởi tạo Stream.
  3. Sử dụng các widget Obx hay GetX để hiển thị dữ liệu.

Ví dụ:

```
Stream<List<SinhVienSnapshot>> getALL(){
Stream<QuerySnapshot> sqns = FirebaseFirestore.
instance.collection("SinhVien").snapshots();
return sqns.map((qsn) => qsn.docs
    .map((doc) => SinhVienSnapshot.fromSnapshot(doc)).toList());
}
class ControllerFirebase extends GetxController{
  final _list = <SinhVienSnapshot>[].obs;
  List<SinhVienSnapshot> get list => _list.value;
  @override
  void onInit() {
    super.onInit();
    _list.bindStream(getALL());
  }
}
```


GetX với dữ liệu Future

- Nguyên tắc:
  - Viết Controller quản lý trạng thái ứng dụng.
  - Viết các phương thức truy xuất dữ liệu bất đồng bộ
    - Sử dụng phương thức then để gán dữ liệu bất đồng bộ vào các trạng thái ứng dụng
    - Sử dụng refresh hay update nếu cần để cập nhập trạng thái lên màn hình trong trường hợp trạng thái là các Custome Type
  - Sử dụng Obx/GetX/GetBuilder để hiển thị trạng thái của ứng dụng


```
Future<List<SinhVienSnapshot>> dsSVTuFirebase() async{
  QuerySnapshot qs = await FirebaseFirestore.instance.collection("SinhVien").get();
  return qs.docs.map((doc) => SinhVienSnapshot.fromSnapshot(doc)).toList();
}
class FirebaseController extends GetxController{
  final _list = <SinhVienSnapshot>[].obs;
  List<SinhVienSnapshot> get list => _list.value;
  void getData(){
    dsSVTuFirebase().then((value) {
      _list.value = value;
      _list.refresh();
    }).catchError((error){
      print(error);
      _list.value =[];
      _list.refresh();
    });
  }
}
```

```
@override
void onInit() {
  super.onInit();
  getData();
}
```