Chương 3

Tổng quan về Dart và Flutter

Mục lục

Mục lục

1.1 Giới thiệu về ngôn ngữ Dart

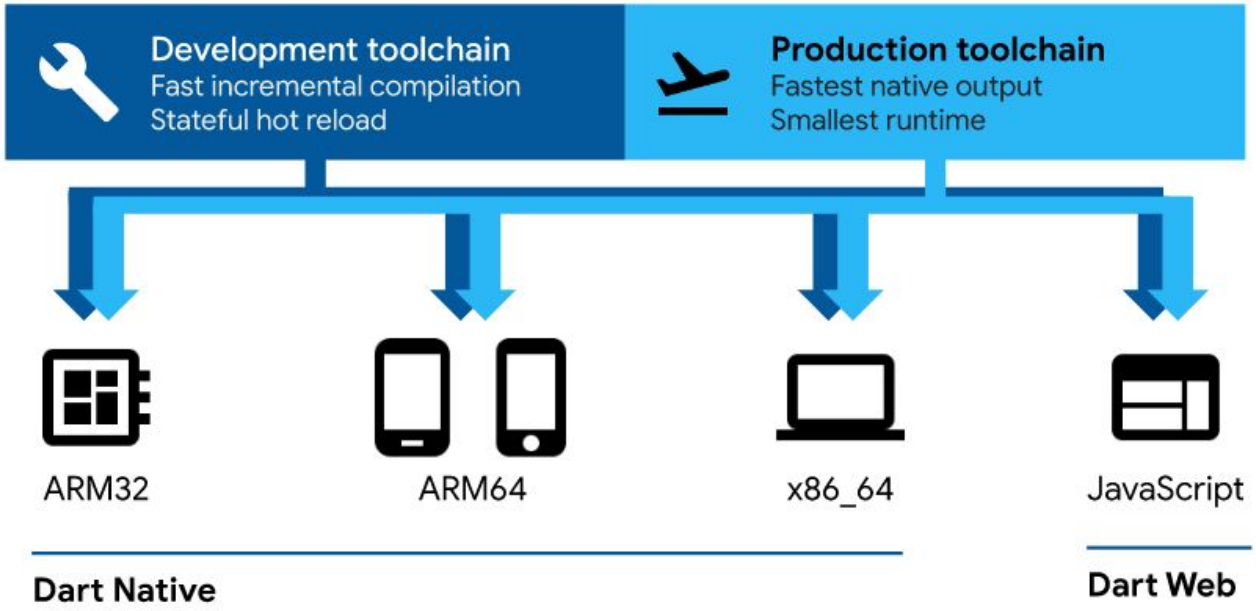
- Dart là một ngôn ngữ lập trình mã nguồn mở (open source) đa năng (general purpose) do Google phát triển.
- Dart được công bố vào năm 2011, bản phát hành ổn định ra đời năm 2013. Phiên bản mới nhất hiện tại là Dart 2.14.
- Dart là một ngôn ngữ đa nền tảng (cross-platform) có thể được sử dụng để phát triển các ứng dụng web, máy tính để bàn, máy chủ và thiết bị di động.
- Các ứng dụng Flutter được viết bằng Dart, cho phép cung cấp trải nghiệm tốt nhất cho nhà phát triển giúp tạo ra các ứng dụng di động chất lượng cao.

1.1 Giới thiệu về ngôn ngữ Dart (2)

- Đặc điểm của Dart
  - Công cụ hiệu quả (productive tooling)
  - Thu gom rác (garbage collection)
  - Chú thích kiểu (type annotations)
  - Kiểu tĩnh (statically typing)
  - Tính di động (portability)

1.2 Các nền tảng (platforms) thực thi

- Hai nền tảng: Dart Native và Dart Web



Các ứng dụng di động và máy tính để bàn, bao gồm máy ảo Dart (Dart VM) với trình biên dịch just-in-time (JIT) và trình ahead-of-time (AOT)

Các ứng dụng trên nền Web.
Mã nguồn Dart được dịch thành mã JavaScript

1.3 Công cụ phát triển ứng dụng trên Dart

- Dart SDK cung cấp các công cụ chuyên biệt cho từng hệ sinh thái phát triển
  - **dart**: Giao diện dòng lệnh để tạo, định dạng, phân tích, kiểm tra, biên dịch và chạy mã Dart
  - **dartaotruntime**: môi trường thực thi cho các ứng dụng được biên dịch AOT
  - **dartdoc**: trình tạo tài liệu
  - **pub**: trình quản lý gói
  - Phát triển các ứng dụng trên nền Web có thêm các công cụ: **dart2js**, **webdev**, **dartdevc**,…

1.4 Chương trình minh hoạ

- Ví dụ một chương trình Dart cơ bản

```
//Chương trình đầu tiên
void main() {
   var a = 'World';
   print('Hello $a');
}
```

Hàm main() là điểm bắt đầu của mọi ứng dụng Dart.

- Thực thi từ dòng lệnh: **`dart hello_world.dart`**
- **DartPad**: công cụ trực tuyến cho phép viết mã Dart và thực thi từ trình duyệt. Sử dụng tại: https://www.dartpad.dev/

Mục lục

2.1 Các quy tắc chung

- Hàm main: một ứng dụng Dart bắt đầu chạy từ hàm main, nó có thể có tham số hoặc không có tham số
- Các lệnh phân biệt chữ hoa/thường và cuối lệnh kết thúc bằng dấu ;
- Một nhóm các lệnh nhóm lại với nhau được gọi là một khối lệnh, sử dụng cặp dấu {} để tạo khối. Khối lệnh có thể lồng nhau.
- Chú thích:
  - Chú thích 1 dòng: //
  - Chú thích nhiều dòng: /* và */

2.1 Các quy tắc chung (2)

- **Quy tắc định danh** (tên biến, tên hàm, tên lớp ...): bắt đầu bằng chữ (a-zA-Z) hoặc _, theo sau là chuỗi ký tự chữ cái có thể kết hợp với chữ số
- Mọi thứ lưu trong biến đều là đối tượng (kể cả số, kể cả null), mọi đối tượng đều sinh ra từ định nghĩa lớp, mọi lớp đều kế thừa từ lớp cơ sở có tên Object
- Dart không có từ khóa *public, private, protected* khi khai báo phương thức, thuộc tính lớp.
- Nếu tên thuộc tính, phương thức bắt đầu bằng _ thì hiểu đó là private

2.2 Biến, hằng, kiểu dữ liệu

- **Khai báo biến**: sử dụng từ khóa **var**, không cần chỉ rõ kiểu dữ liệu, kiểu dữ liệu của biến phụ thuộc vào đối tượng gán cho biến.

```
var a = "Learn Dart"; //a có kiểu String, nó chỉ lưu chuỗi
```

- Có thể khai báo biến và chỉ định kiểu dữ liệu cụ thể cho nó

```
String  s     = 'Chuỗi ký tự'; // Khai báo biến chuỗi
double  d     = 1.1234;        // khai báo biến số thực
int     i     = 1;             // biến số nguyên
bool    found = true;          // biến logic
```

- Khi biến có thể chấp nhận mọi kiểu thì sử dụng từ khóa dynamic

```
dynamic dyn = 123;             // Khởi tạo là số int
dyn = "Dynamic";               // Gán chuỗi
dyn = 1.12345;                 // Gán số double
```

2.2 Biến, hằng, kiểu dữ liệu (2)

- **Khai báo hằng**: sử dụng từ khóa const hoặc final

```
const ten_hang = biểu_thức_giá_trị;
final name_1           = biểu_thức_giá_trị;
final String name_2    = biểu_thức_giá_trị; //Chỉ rõ kiểu của hằng
```

- const: hằng số lúc biên dịch, giá trị của nó phải là cụ thể ngay lúc viết code
- final: chỉ được gán giá trị một lần duy nhất, gán lần thứ 2 sẽ lỗi (trước khi sử dụng phải có 1 lần gán). Nó gọi là hằng số lúc thực thi, giá trị hằng số này có thể khác nhau mỗi lần chạy.

```
const minutes   = 24 * 60;

var so_ngau_nhien = Random(1000).nextInt(500);
//Tạo hằng số
final a = so_ngau_nhien * 2;
```

2.2 Biến, hằng, kiểu dữ liệu (3)

- Các kiểu dữ liệu cơ bản

Kiểu dữ liệu	Mô tả
int	Biểu diễn các giá trị số nguyên
double	Biểu diễn giá trị số thực 64bit
String	Biểu diễn chuỗi ký tự Unicode(UTF-16). Nó được nhập vào trong cặp nháy đơn " hoặc nháy kép "". Dùng ký hiệu \", \' để biểu diễn ký tự ', " trong chuỗi
bool	Biểu diễn logic đúng / sai với hai giá trị true và false.
List / Mảng	Còn gọi là danh sách list, nó lưu tập hợp các dữ liệu, giống khái niệm của JavaScript. Khởi tạo một mảng dùng ký hiệu [], giá trị các phần tử liệt kê cách nhau bởi ,. Các phần tử mảng có chỉ số từ 0, để truy cập đến phần tử nào dùng ký hiệu [chỉ_số]
Map	Cũng lưu tập hợp các giá trị (còn gọi là mảng kết hợp), thay vì sử dụng chỉ số từ 0 để tham chiếu đến phần tử, mỗi phần tử trong Map lưu theo cặp key:value, dùng ký hiệu {} để khởi tạo Map hoặc khởi tạo bằng Map(); Truy cập đến phần tử Map dùng ký hiệu chấm .key

2.3 Các toán tử trong Dart

- Toán tử số học:
  - Các toán tử phổ biến: +, -, *, /
  - ~/ : phép chia lấy phần nguyên,
  - % : phép chia lấy phần dư
- Toán tử tăng giảm đơn vị: ++var, var++, --var, var --
- Toán tử logic

Toán tử	Ý nghĩa
`||`	Phép logic hoặc, `a || b` kết quả `true` nếu `a` hoặc `b` là `true`
`&&`	Phép logic và, `a && b` kết quả `true` nếu `a` và `b` đều `true`
`!biểu_thức`	Phép phủ định `!a` nếu `a` là `true` thì kết quả phép toán là `false`

2.3 Các toán tử trong Dart (2)

- Toán tử so sánh:

Toán tử	Ý nghĩa
`==`	So sánh bằng `5 == 5` kết quả `true`, `5 == 6` kết quả `false`
`!=`	So sánh khác `5 != 5` kết quả `false`, `5 != 6` kết quả `true`
`>`	So sánh lớn hơn `5 > 5` kết quả `false`, `6 > 5` kết quả `true`
`<`	So sánh nhỏ hơn `5 < 5` kết quả `false`, `5 > 6` kết quả `true`
`<=`	So sánh nhỏ hơn hoặc bằng
`>=`	So sánh lớn hơn hoặc bằng

- Toán tử gán: = và các toán tử gán rút gọn

??=

Assign the value only if the variable is null

2.3 Các toán tử trong Dart (3)

- Toán tử điều kiện:
  - **biểu_thức_điều_kiện ? biểu_thức_1 : biểu_thức_2**
  - **biểu_thức_1 ?? biểu_thức_2**
- Phép toán với kiểu String
  - Nối hai chuỗi ký tự: +
  - So sánh hai chuỗi ký tự: ==
  - Chuỗi ký tự gồm nhiều dòng, dùng cú pháp sau (các dòng nằm giữa cặp ' ' ' hoặc " " "):
  - Chèn một biến hoặc một biểu thức vào chuỗi bằng cách ký hiệu $tên_biến, ${biểu thức giá trị}

```
String s1 = '''
Các dòng
chữ trong chuỗi s1
```

```
var a = 10;
var b = 20;
String kq = "Hai số $a, $b có tổng ${a + b}";
print(kq); //Hai số 10, 20 có tổng 30
```

2.3 Các toán tử trong Dart (4)

- Một số toán tử trên lớp, đối tượng:

Toán tử	Ý nghĩa
`[]`	Truy cập phần tử mảng
`.`	Truy cập phương thức, thuộc tính đối tượng
`.`	Truy cập phương thức, thuộc tính đối tượng khi đối tượng đó khác null `myobject?.method();`
`as`	Chuyển kiểu: `(var as MyClass)`
`is`	Kiểm tra kiểu: `(var is MyClass)`
`is!`	Kiểm tra kiểu: `(var is! MyClass)`

2.4 Các cấu trúc điều khiển

- Dart cung cấp một số cú pháp luồng điều khiển rất giống với các ngôn ngữ lập trình khác
  - if-else
  - switch/case
  - Cấu trúc lặp: for, while, và do-while
  - Các lệnh break và continue
  - asserts
  - Xử lý ngoại lệ với: try/catch và throw, on

https://dart.dev/guides/language/language-tour#control-flow-statements

2.5 Xây dựng hàm

- Cú pháp:

```
kiểu-trả-về tên-hàm(danh-sách-tham-số) {
    //các lệnh
    return giá trị;
}
```

- Trong Dart function là một kiểu, nghĩa là hàm có thể được gán cho các biến hoặc truyền dưới dạng tham số cho các hàm khác

```
String sayHello() {
    return "Hello world!";
}
void main() {
    var sayHelloFunction = sayHello;
    print(sayHelloFunction());
}
```

2.5 Xây dựng hàm (2)

- Tham số tùy chọn: khi gọi hàm có sử dụng hoặc không. Các tham số tùy chọn gom lại trong [], nếu khi gọi hàm không có tham số này thì nó nhận giá trị null

```
double tinhtong(var a, [double b, double c]) {
   var tong = a;
   if (b != null)
     tong += a;
   tong += (c!=null) ?  c: 0;
}
print(tinhtong(1));        //1.0
print(tinhtong(1,2));      //3.0;
print(tinhtong(1,2,3));    //6.0;
```

- Tham số mặc định: khi gọi hàm mà thiếu giá trị cho tham số đó, thì nó sẽ nhận mặc định

tham số b mặc định là 1, c mặc định là 2

```
double tinhtong(var a, {double b:1, double c:2}) {
  return a + b + c;
}
```

2.5 Xây dựng hàm (3)

- Cú pháp viết rút gọn (còn gọi là hàm mũi tên hay hàm lambda):

[return_type] function_name(parameters) => expression;

```
double tinhtong(var a, var b) => a + b;
```

- Hầu hết khai báo hàm là có tên hàm, tuy nhiên trong nhiều ngữ cảnh khai báo hàm và không dùng đến tên, hàm đó gọi là **hàm ẩn danh**

```
(var a, var b) {
    return a + b;
};
//Có thể dùng ký hiệu mũi tên  () => {}
(var a, var b) => {
    return a + b;
}
//Nếu chỉ có 1 biểu thức trả về như trên có thể viết gọn
hơn
(var a, var b) => return a + b;
```

2.6 Quản lý các gói trong Dart

- Trình quản lý package: **pub** (https://pub.dev/)
- Mỗi gói trong pub cần một số metadata để nó có thể chỉ định các phần phụ thuộc.Thông tin này đặt trong tệp: **pubspec.yaml**
- Một pubsec có thể gồm các trường: `name, version, description, dependencies,…`

```
name: Ten_du_an
description: Mô tả cho dự án

dependencies:
    package1: '^version_pack1'
    package2: '^version_pack2'
    package3: '^version_pack3'
```

Ví dụ: dự án có sử dụng package dialog và google_maps:

```
name: my_project
description: Mô tả cho dự án

dependencies:
    dialog: '^0.7.0'
    google_maps: '^3.3.2'
```

2.6 Quản lý các gói trong Dart (2)

- Nạp các thư viện để sử dụng trong mã nguồn, cú pháp:

```
import 'uri';
```

- uri như là một định danh duy nhất trỏ đến thư viên cần nạp. Nó có thể có các dạng:
  - Các thư viện xây dựng sẵn của Dart có cấu trúc uri **dart:tên_thư_viện** ví dụ *dart:convert, dart:html, dart:math, dart:js, dart:web_sql*. Ví dụ cần nạp thư viện toán học *dart:math* thì dùng

```
import 'dart:math';
```

  - Nạp từ file dự án, url chỉ đến đường dẫn file dart cần nạp vào. Ví dụ:

```
import 'lib/mylib.dart';
```

  - Thư viện nạp từ các gói package tải về, thì uri có dạng **package:tên_gói/thư_viện_gói**.
    - Ví dụ gói googleapis_auth, có thành phần auth_browser cung cấp chức năng xác thực Auth với tài khoản google, thì nạp thư viện đó vào bằng

```
import "package:googleapis_auth/auth_browser.dart";
```

2.6 Quản lý các gói trong Dart (3)

- Các thư viện cung cấp sẵn trong Dart:

`dart:core`	Thư viện cung cấp các hàm, lớp cơ bản, nó tự động nạp nên bạn không cần import thư viện này
`dart:collection`	Cung cấp các cấu trúc dữ liệu như HashSet, HashMap, Queue ...
`dart:math`	Cung cấp hàm toán học
`dart:convert`	Tính năng mã hóa, giải mã dữ liệu, kể cả JSON, UTF-8
`dart:io`	Thư viện IO cung cấp các chức năng về File, Socket, HTTP
	Một số thư viện dánh cho WEB như `dart:js`, `dart:html` ...

Mục lục

3.1 Các đặc trưng hướng đối tượng trong Dart

- Các nguyên lý OOP được thể hiện trong Dart:
  - Đối tượng và lớp
    - Dart class chứa methods và fields, các thành phần tĩnh.
  - Đóng gói
    - Không có các giới hạn truy cập tường minh
  - Kế thừa
    - Các lớp kết thừa trực tiếp hoặc gián tiếp từ lớp cha Object. Chỉ hỗ trợ đơn kế thừa.
    - mixins: được sử dụng để mô phỏng đa kế thừa lớp
  - Trừu tượng hoá: các lớp trừu tượng và giao diện
  - Đa hình
    - Cho phép ghi đè phương thức của lớp cha nhưng không hỗ trợ nạp chồng phương thức trong lớp

Ví dụ: Lớp và đối tượng

```
class Product {
  //Khai báo các thuộc tính
  String manufacture = '';
  String name = '';
  var    price;
  int    quantity;

  //Khai báo hàm khởi tạo
  Product(var price, {int quantity:0}) {
    this.price    = price;
    this.quantity = quantity;
  }


  //Khai báo các phương thức
  calulateTotal() {
    return this.price * this.quantity;
  }
  showTotal() {
    var tong = this.calulateTotal();
    print("Tổng số tiền là: $tong");
  }
}
```

để tham chiếu đến đối tượng của lớp dùng từ khóa **this**

```
var product = new Product(600,
quantity: 1);
product.showTotal();

product.quantity = 2;
product.showTotal();

//KẾT QUẢ CHẠY
Tổng số tiền là: 600
Tổng số tiền là: 1200
```

Ví dụ: Setter/Getter

Sử dụng từ khóa **get** trước một hàm **không tham số** thì hàm đó trở thành Getter, sử dụng từ khóa **set** trước **hàm 1 tham số** thì đó là hàm Setter

```
//getter
 get nameProduct {
   return this.name;
 }
```

```
//setter
  set nameProduct(name) {
    this.name = name;
    switch (this.name) {
      case 'Iphone 6':
        this.manufacture = "Apple";
      break;

      case 'Galaxy S6':
        this.manufacture = 'Samsung';
      break;

      default: this.manufacture = '';

    }
  }
```

```
product.nameProduct = "Galaxy S6";      //Gọi đến hàm Setter
var info = product.nameProduct;         //Gọi đến Getter
```

3.2 Kế thừa

- Kế thừa
  - Từ khóa **extends**
  - Lớp con truy cập đến dữ liệu của lớp cha với từ khoá **super**
  - Ghi đè phương thức của lớp cha: sử dụng chỉ thị **@override**

```
class Table extends Product {
  double length = 0;
  double width   = 0;
  //gọi hàm tạo lớp cha sau dấu :
  Table(var giatien) : super(giatien,
quantity:1) {
        this.name = "Bàn Ăn";
  }

  @override
  showTotal() {
      print('Sản phẩm:' + this.name);
      //gọi đến phương thức ở lớp cha
      super.showTotal();
    }
}
```

Toán tử cascade ..

- Thay vì phải viết đầy đủ đối tượng thì chỉ cần viết nó một lần, các tương tác tiếp theo thay thế bằng ..

```
var table = new Table(1);
  table
  ..calulateTotal()      //Thay cho table.calulateTotal();
  ..length=100           //Thay cho table.length=100;
  ..name='Abc'
  ..quantity=100
  ..showTotal();
```

3.2 Kế thừa (2)

- Lớp trừu tượng: lớp không dùng trực tiếp để tạo ra đối tượng, sử dụng từ khoá abstract
  - Chứa phương thức trừu tượng, chỉ có khai báo, không có phần thân.

```
abstract class A {
  //Khai báo phương thức trừu tượng (chỉ có tên)
  void displayInfomation();
}
```

- Giao diện: khi một lớp được coi là giao diện thì lớp triển khai nó phải định nghĩa lại mọi phương thức, thuộc tính có trong giao diện, sử dụng từ khoá implements

```
class C implements B {
  @override
  String name;
  @override
  void displayInfomation() {
      // ...
  }
}
```

3.2 Kế thừa (3)

- Khái niệm mixin: mixin là một lớp, nó không được sử dụng trực tiếp để tạo ra đối tượng, một mixin chứa các phương thức, thuộc tính dùng để gộp vào một lớp khác.

```
mixin M {
  var var1 = null;
  showSomething() {
    print('Print message ...');
  }
}
```

```
class C extends B with M {
  @override
  String name;

  @override
  void displayInfomation() {
  }
}
```

3.3 Callable class

- Callable classes: các lớp Dart cũng có thể hoạt động giống như các hàm, nghĩa là, chúng có thể được gọi, nhận một số đối số và trả về kết quả

```
class Employee{
  String call(String name, int age){
    return "Employee: $name Age: $age";
  }
}
void main(){
  Employee emp = new Employee();
  var msg = emp("John", 30);
  print("Dart Callable Class Example");
  print(msg);

}
```

Phương thức **call()** là một phương thức đặc biệt trong Dart. Mọi lớp định nghĩa nó có thể hoạt động như một hàm Dart bình thường.

3.4 Giới thiệu về async programming

- Lập trình đồng bộ Synchronous:
  - Code chạy trong Dart là chạy trên một luồng (thread), dòng code được thi hành hết câu lệnh này sang câu lệnh khác.
  - Nếu một khối lệnh nào đó khóa thread (làm tắc thread) thì toàn bộ ứng dụng bị treo.

```
const info = '#4fs358w';
getInfomation() {
  return info;
}
showInfomation() {
  var data = getInfomation();
  print('This is your data -' +
DateTime.now().toString());
  print(data);
}
secondFunction() {
  print('Thời gian - ' +
DateTime.now().toString());
}
main() {
  showInfomation();
  secondFunction();
}
```

Giả sử ở code trên nếu showInfomation() (hoặc getInfomation()) mất nhiều thời gian để hoàn thành (10s, 20s...) thì các khối lệnh khác (hàm secondFunction()) phải chờ nó hoàn thành mới được thi hành.

3.4 Giới thiệu về async programming (2)

- Lập trình bất đồng bộ - Asynchronous: khi một phương thức đang thực hiện công việc của mình thì khối lệnh khác - hàm khác vẫn được thi hành.
  - Cơ chế bất đồng bộ là chương trình cho phép phân nhánh quá trình code hoạt động, làm cho có cảm giác như đa luồng (có thể vẫn là 1 thread) - có lúc thì chạy code ở nhánh này, có lúc thì chạy code ở nhánh khác - cảm giác thi hành nhiều việc đồng thời.
- Dart sử dụng lớp Future và các từ khoá async, await

3.4 Giới thiệu về async programming (3)

- Đối tượng **Future<T>** trong Dart đại diện cho một giá trị sẽ được cung cấp vào một lúc nào đó trong tương lai.
- Nó được sử dụng để đánh dấu một phương thức với một kết quả trả về trong tương lai; nghĩa là phương thức trả về đối tượng **Future<T>** sẽ không có giá trị kết quả thích hợp ngay lập tức mà thay vào đó, sau một số tính toán tại thời điểm sau đó.

3.4 Giới thiệu về async programming (4)

- Hàm bất đồng bộ async: khai báo có từ khóa async phía sau, và đối tượng trả về của hàm là Future<T>

```
Future<int> functionName() async {
  return 1;
}
```

- Nếu hàm đó đã khai báo là bất đồng bộ async thì trong hàm có thể sử dụng cú pháp **await biểu_thức;** cho biết chờ cho biểu thức thi hành xong mới thi hành các code tiếp theo của hàm

```
Future<int> functionName() async {
    await biểu_thức;     //ví dụ await getInfomation();
    return 1;
}
```

Mục lục

4.1 Giới thiệu về Flutter

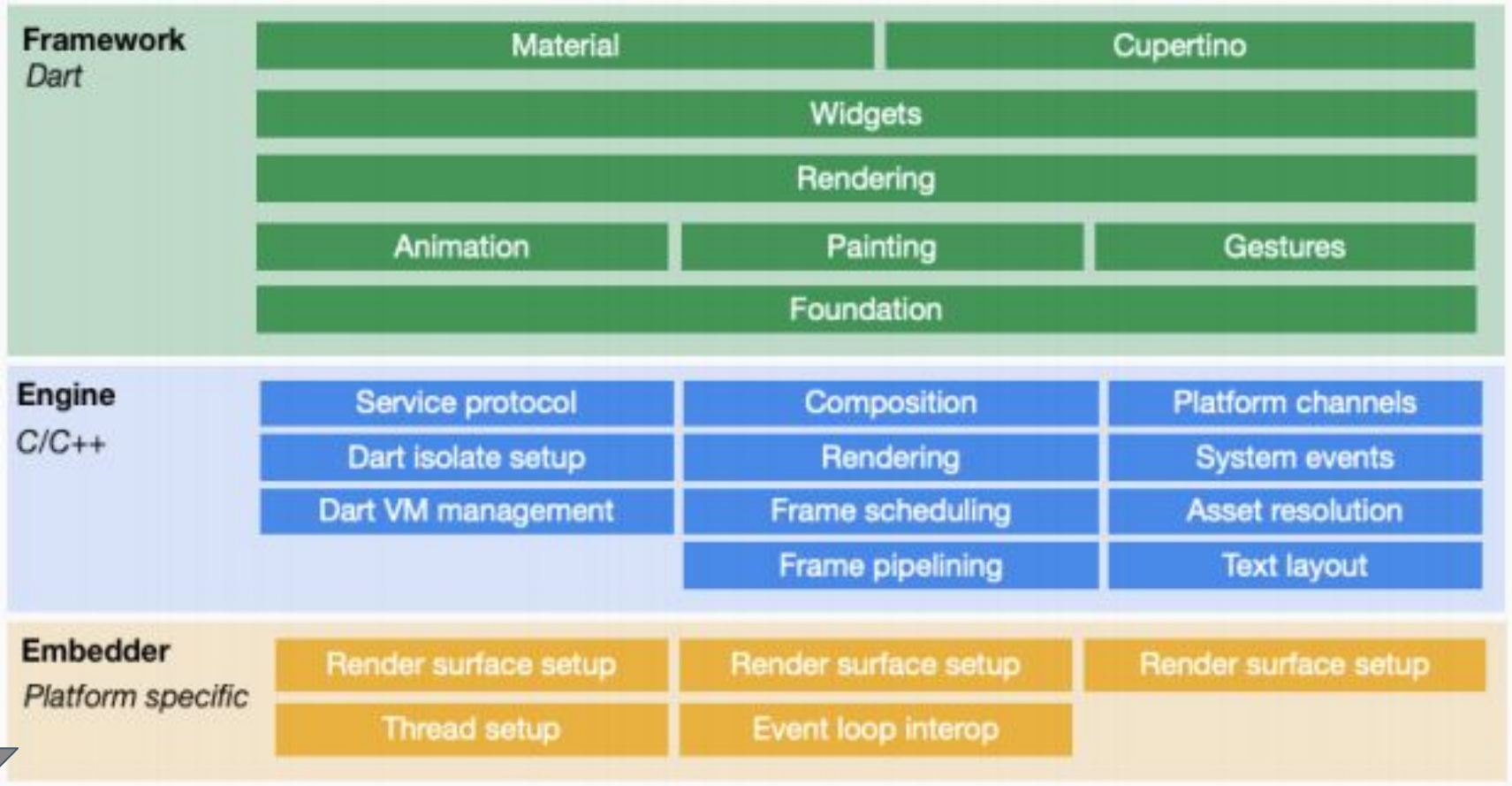
- Flutter là một framework phát triển ứng dụng đa nền tảng (Cross-Platform SDK) dành cho thiết bị di động, bao gồm:
  - Khung giao diện người dùng (UI Framework) hoàn chỉnh cho ứng dụng di động
  - Danh mục Widget khổng lồ
  - Các công cụ phát triển
- Cho phép xây dựng các ứng dụng di động độc lập với nền tảng, hiện tại hỗ trợ Android, iOS và Fuchsia

4.1 Giới thiệu về Flutter (2)

- Các đặc điểm nổi bật của Flutter:
  - Hiệu năng cao (High performance)
  - Kiểm soát hoàn toàn UI (Full control of the UI): hỗ trợ rất nhiều widget khác nhau, giao diện người dùng đẹp và linh hoạt, thể hiện cùng một UI trên nhiều nền tảng
  - Ngôn ngữ Dart (Dart programming language): đơn giản, dễ học, Dart có một kho lớn các gói phần mềm cho phép mở rộng
  - Được hỗ trợ bởi Google (Being backed by Google)
  - Framework mã nguồn mở (Open source framework)
  - Hot Reload: Flutter cung cấp khả năng ghi nhớ state của ứng dụng

4.2 Kiến trúc Flutter

- Kiến trúc Flutter framework được chia làm các tầng:



4.2 Kiến trúc Flutter (2)

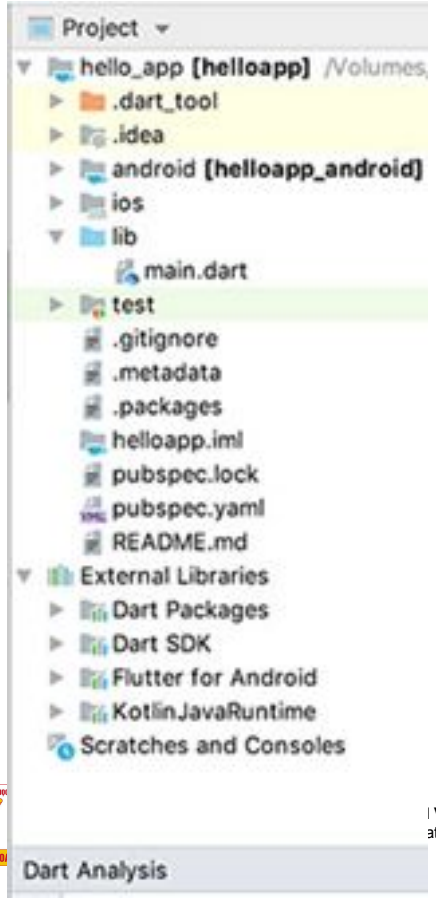
- Tầng Framework (Dart)
  - **Foundation**: core library
  - **Animation/Painting/Gesture**: thực hiện hóa các chuyển động trong Flutter (Animation), hỗ trợ xây dựng giao diện (Painting), nhận dạng tương tác cử chỉ của người dùng (Gesture).
  - **Rendering**: các UI components được tổ chức dưới dạng cây phân tầng với mỗi node là 1 RenderObject.
  - **Widgets**: được xây dựng theo dạng cây với những object là lớp kế thừa của RenderObject. Widgets nhận và lưu trữ trạng thái (state) của UI, và sẽ rebuild nếu state thay đổi.
  - **Material/Cupertino**: Được xây dựng từ Widgets library để triển khai ngôn ngữ thiết kế Material hoặc iOS (Cupertino)

4.2 Kiến trúc Flutter (3)

- Tầng Engine (C/C++)
  - Triển khai các API cốt lõi của Flutter bao gồm đồ họa (thông qua Skia), text layout, file và network I/O, trợ năng, kiến trúc plugin và Dart runtime
  - Đóng vai trò trung gian giữa mã nguồn được viết bởi Dart và thiết bị phần cứng (hoặc phần mềm nằm ngoài ứng dụng).
  - Thông dịch các đoạn mã Dart theo phương thức JIT hoặc AOT
  - Thực thi các đoạn mã đã được thông dịch hoặc biên dịch cũng như cung cấp các runtime system bao gồm garbage collector, các thư viện cần có của ngôn ngữ.
- Tầng Embedder
  - Cung cấp truy xuất vào các dịch vụ của nền tảng (Android, IOS,…)

4.3 Ứng dụng Flutter đầu tiên

- Ứng dụng Flutter được phát triển trên Android Studio
- Cấu trúc một dự án Flutter:



- **android** − Thư mục code sinh tự động cho ứng dụng Android
- **ios** − Thư mục code sinh tự động cho ứng dụng iOS
- **lib** − Main folder chứa Dart code được viết khi sử dụng flutter framework
- **ib/main.dart** − File đầu tiên là điểm khởi đầu của ứng dụng Flutter application
- **test** − Folder chứa Dart code để test flutter application
- **test/widget_test.dart** − Sample code
- **.gitignore** − Git version control file - File này chứa cấu hình cho project git
- **.metadata** − sinh tự động bởi flutter tools
- **.packages** − sinh tự động để theo dõi flutter packages
- **.iml** − project file của Android studio
- **pubspec.yaml** − Được sử dụng **Pub**, Flutter package manager
- **pubspec.lock** − Sinh tự động bởi Flutter package manager, **Pub**
- **README.md** − mô tả project được viết theo cấu trúc Markdown

4.3 Ứng dụng Flutter đầu tiên (2)

- Mã nguồn *lib/main.dart*

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
   // This widget is the root of your application.
   @override
   Widget build(BuildContext context) {
      return MaterialApp(
         title: 'Hello World Demo Application',
         theme: ThemeData(
            primarySwatch: Colors.blue,
         ),
         home: MyHomePage(title: 'Home page'),
      );
   }
}
```

import flutter package, tên là *material*. **Material** là một flutter package được sử dụng để tạo giao diện người dùng theo Material design cho Android

Điểm khởi đầu của Flutter application là hàm main của ứng dụng.
Phương thức *runApp* được gọi và truyền vào đối tượng của lớp *MyApp*. Mục đích chính của phương thức *runApp* là đưa giao diện widget vào hiển thị trên màn hình

4.3 Ứng dụng Flutter đầu tiên (3)

- Mã nguồn *lib/main.dart*

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
   // This widget is the root of your application.
   @override
   Widget build(BuildContext context) {
      return MaterialApp(
         title: 'Hello World Demo Application',
         theme: ThemeData(
            primarySwatch: Colors.blue,
         ),
         home: MyHomePage(title: 'Home page'),
      );
   }
}
```

Widget được sử dụng để tạo UI (giao diện người dùng).
StatelessWidget là một widget, nó không bao gồm trạng thái nào của widget.

- *MyApp* kế thừa *StatelessWidget* và ghi đè phương thức *build()*.
- Mục đích của phương thức *build* là tạo một phần UI cho ứng dụng. Ở đây, phương thức *build* sử dụng *MaterialApp*, một widget để tạo layout UI gốc cho ứng dụng.
- Bao gồm 3 thành phần chính là - *title (tiêu đề), theme (chủ đề)* và *home (trang chủ hay phần màn hình)*

4.3 Ứng dụng Flutter đầu tiên (4)

- Mã nguồn *lib/main.dart*

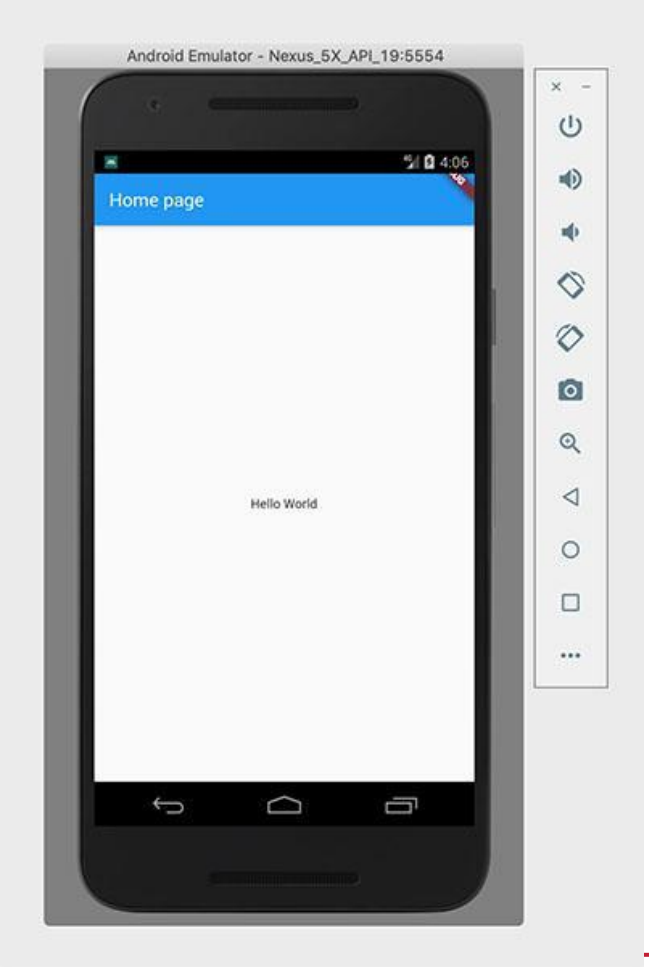
```
class MyHomePage extends StatelessWidget {
   MyHomePage({Key key, this.title}) : super(key: key);
   final String title;
   @override
   Widget build(BuildContext context) {
      return Scaffold(
         appBar: AppBar(
            title: Text(this.title),
         ),
         body: Center(
            child:
            Text(
               'Hello World',
            )
         ),
      );
   }
}
```

- *MyHomePage* tương tự *MyApp* ngoại trừ nó trả về *Scaffold* Widget.
- *Scaffold* là một top level widget đứng sau *MaterialApp* widget được sử dụng để tạo UI theo material design.
- Nó có hai thuộc tính quan trọng nhất là *appBar* để hiển thị phần đầu của ứng dụng và *body* để hiển thị nội dung chính của ứng dụng.

AppBar là một widget khác để tạo phần đầu của ứng dụng và sử dụng các thuộc tính của *appBar*.
Trong các thuộc tính của *body*, chúng ta sử dụng một *Center* widget, đây là một widget con. *Text* là một widget cuối cùng, phổ biến để hiển thị một văn bản giữa màn hình

4.3 Ứng dụng Flutter đầu tiên (5)

- Kết quả thực thi ứng dụng trên máy giả lập Android Emulator



4.4 Flutter widgets

- Widget là thành phần cơ bản nhất tạo nên toàn bộ giao diện người dùng của ứng dụng Flutter.
- Mỗi ứng dụng là một top-level widget bao gồm một hoặc nhiều các widget con, mỗi widget này lại có thể chứa một hoặc nhiều widget con khác.
- Nhờ sự kết hợp linh hoạt theo cấu trúc cây phân cấp cho phép phát triển các ứng dụng có giao diện phức tạp.

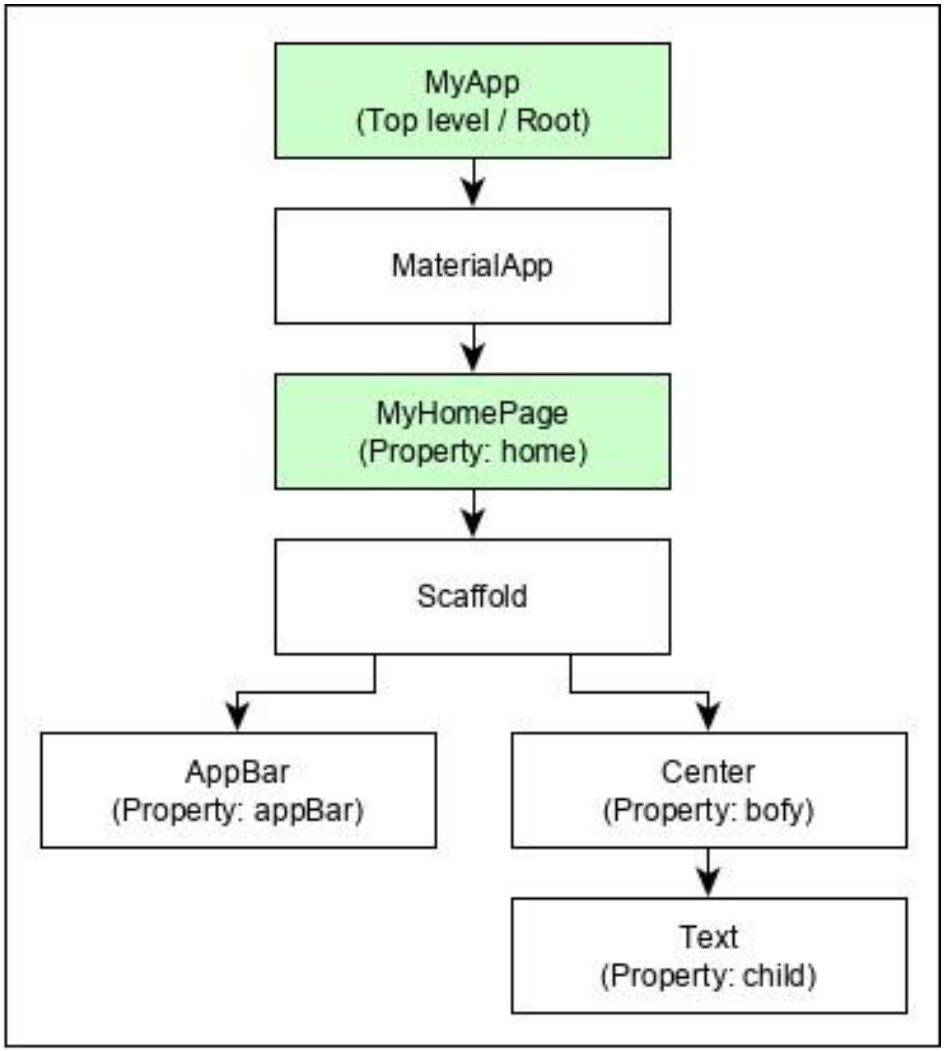
4.4 Flutter widgets (2)

- Widgets có thể là:
  - Một phần tử cấu trúc (ví dụ: nút, menu,…)
  - Một phần tử kiểu dáng (themes, styles, fonts,…)
  - Một đặc trưng của layout (padding, center,…)
  - …

```
class PaddedText extends StatelessWidget {

 final String _data;

 PaddedText(this._data, {Key key})
      : super(key: key);

 @override
 Widget build(BuildContext context) {
   return new Padding(
       padding: const EdgeInsets.all(4.0),
       child: new Text(_data)
   );
 }
}
```

4.4 Flutter widgets (3)

- Cấu trúc widget của ứng dụng hello world (minh hoạ ở phần trước)



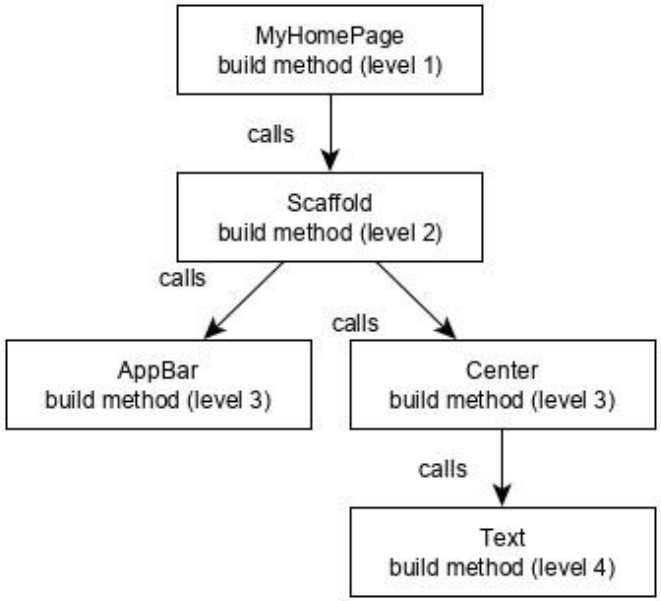
4.4 Flutter widgets (4)

- Cách tiếp cận của Flutter trong việc hiển thị giao diện người dùng mang tính chất khai báo (declarative).
  - Nghĩa là: "đây là trạng thái hiện tại của ứng dụng, hãy vẽ thứ gì đó trên màn hình cho phù hợp"
- Ưu điểm: chỉ có một đường dẫn mã cho bất kỳ trạng thái nào của giao diện người dùng
  - Các nhà phát triển chỉ cần mô tả cách màn hình trông như thế nào đối với một trạng thái nhất định
- Công thức giao diện người dùng:

4.4 Flutter widgets (5)

- Lời gọi phương thức build() trong cây widget

```
class MyHomePage extends StatelessWidget {
   MyHomePage({Key key, this.title}) : super(key: key);

   final String title;
   @override
   Widget build(BuildContext context) {
      return Scaffold(
         appBar: AppBar(title: Text(this.title), ),
         body: Center(child: Text( 'Hello World',)),
      );
   }
}
```



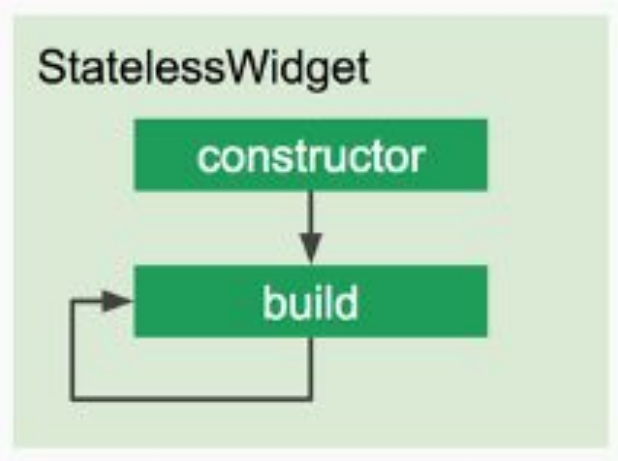
4.5 Quản lý trạng thái

- Quản lý trạng thái (state) là một trong những nhiệm vụ quan trọng và cần thiết trong vòng đời của ứng dụng.
- Phân loại state:
  - **Local state** – Trạng thái cục bộ có thể được gắn vào một widget. Ví dụ, nó có thể là tab hiện tại trong "tab selector" widget hoặc trạng thái của checkbox (được chọn hoặc bỏ chọn)…
  - **Application state** (trạng thái ứng dụng) – trạng thái không cục bộ, chia sẻ trên nhiều widget, trạng thái có thể được giữ trong phiên người dùng. Ví dụ là trạng thái đăng nhập của người dùng, các sản phẩm trong giỏ hàng hoặc tin nhắn trò chuyện,…

4.5 Quản lý trạng thái (2)

- Stateless vs. Stateful Widget
  - **Stateless Widget**– là một widget không quản lý trạng thái của chính nó. Một khi nó nhận được các tham số và được xây dựng thông qua phương thức **build()**, nó sẽ không thể thay đổi được.
    - Vòng đời của stateless widget

Được khởi tạo qua một constructor và vẽ lên giao diện với hàm build(BuildContext)

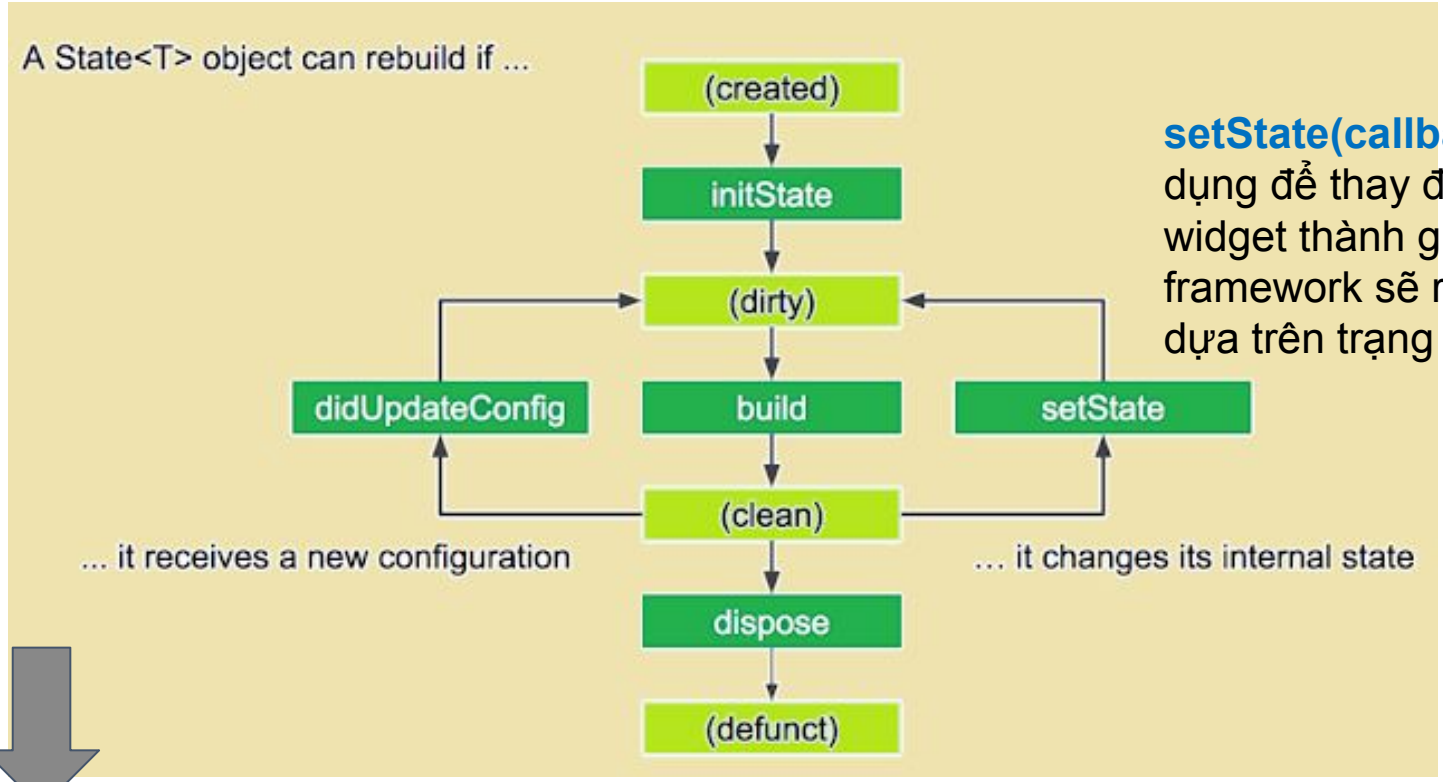


4.5 Quản lý trạng thái (3)

- Stateless vs. Stateful Widget
  - **Stateful Widget**– là một widget được gắn với đối tượng state object.
    - Stateful Widget không chỉ có phương thức build() mà có đối tượng State được liên kết xác định một số phương thức để hỗ trợ vòng đời của widget.
    - Ví dụ các phương thức này là initState() cho khởi tạo trạng thái và dispose() để xóa tài nguyên được cấp phát.

4.5 Quản lý trạng thái (4)

- Stateless vs. Stateful Widget
  - **Stateful Widget**– Vòng đời của stateful widget



setState(callback) được sử dụng để thay đổi trạng thái của widget thành giá trị mới và framework sẽ rebuild widget dựa trên trạng thái mới đó.

Ví dụ

Chức năng của chương trình đơn giản là khi ấn vào button "+" ở góc dưới bên phải, con số ở chính giữa màn hình sẽ tăng lên 1 đơn vị.

Ví dụ

- Ví dụ:

MyHomePage là 1 stateful widget

```
class MyHomePage extends statefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
```

định nghĩa 1 đối tượng State là _MyHomePageState

Một widget State hợp lệ là một lớp kế thừa từ lớp State

state của widget ở trên được xác định trong thuộc tính **_counter**. Thuộc tính này lưu trữ số lần nhấn nút vào button

MyHomePage phải trả về một đối tượng State hợp lệ trong phương thức createState(), đó là 1 thể hiện của _MyHomePageState

Ví dụ

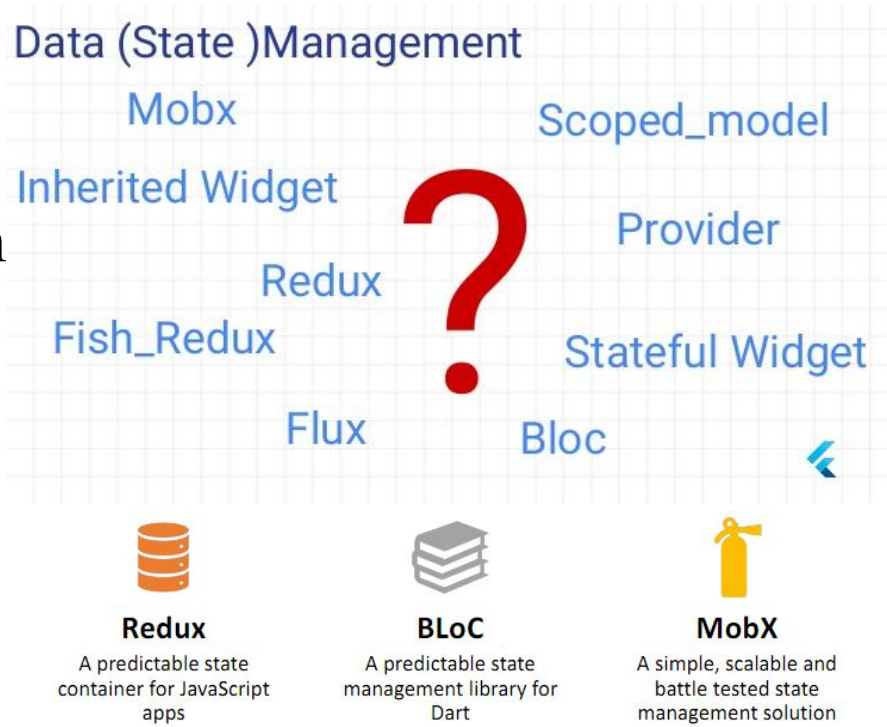
onPressed của widget FloatingActionButton, thuộc tính này nhận 1 hàm (ở đây là **_incrementCounter()**), và hàm sẽ được thực thi khi nút được nhấn

hàm **_incrementCounter()** được thực thi sẽ gọi phương thức **setState()** làm trạng thái widget thay đổi framework được thông báo rằng nó cần phải rebuild widget con

```
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // This trailing comma makes auto-formatting nicer.
    );
  }
}
```

4.5 Quản lý trạng thái (6)

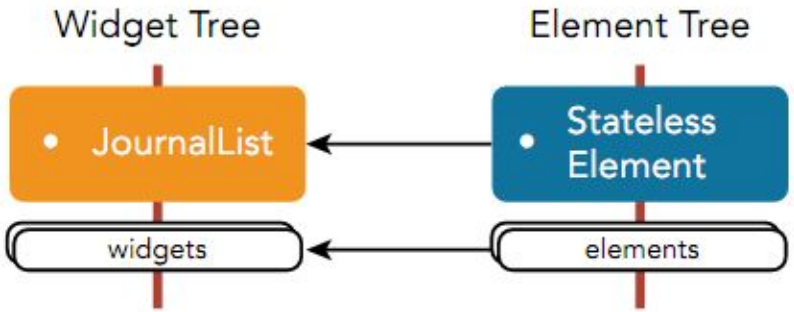
- Các giải pháp quản lý trạng thái ứng dụng (Application state)
  - Inherited Widget
  - Provider Package
  - Business Logic Componen
  - flutter_bloc Package
  - Redux
  - …



- Các giải pháp này sẽ được mô tả chi tiết hơn trong Chương 7.

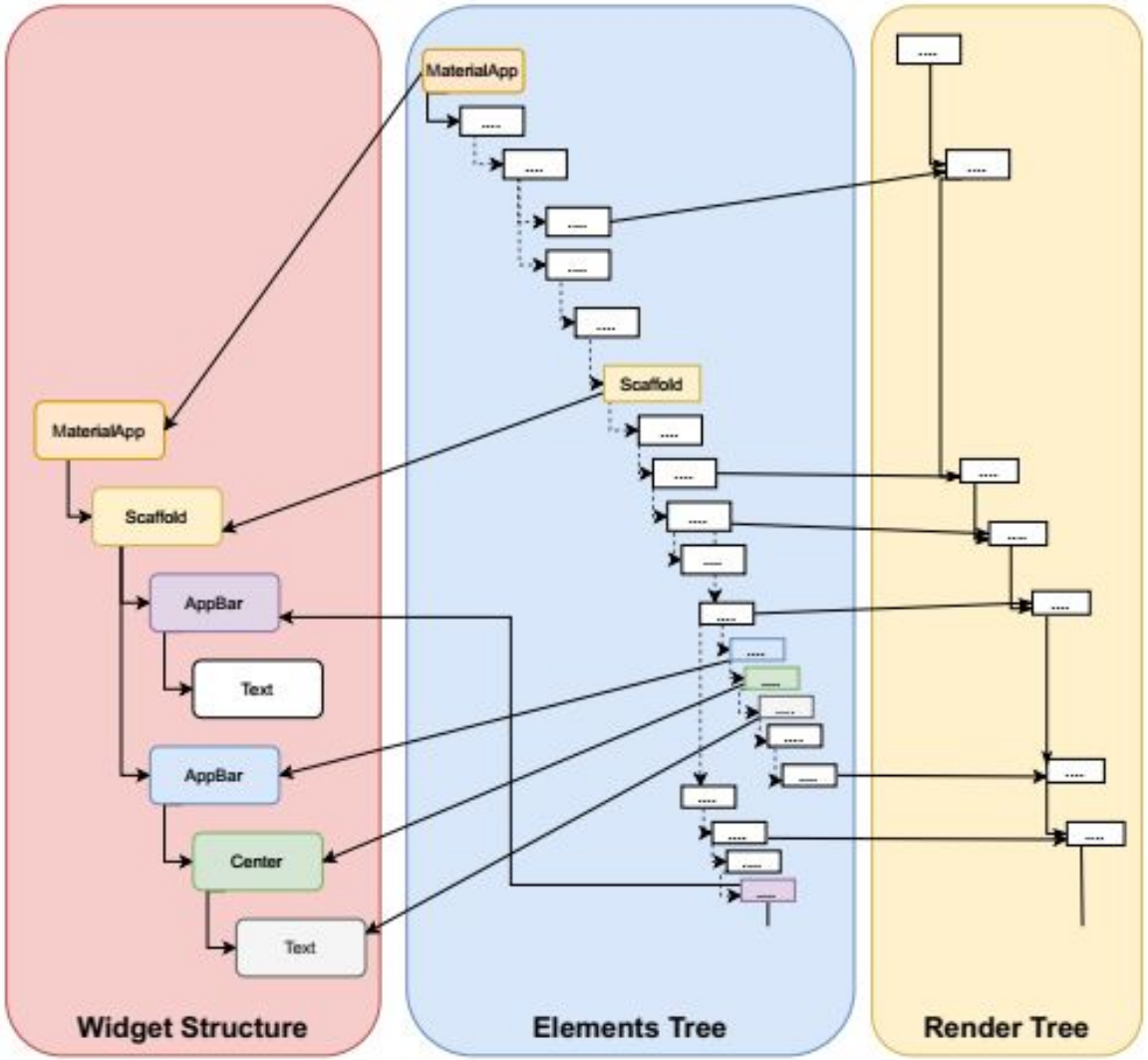
4.6 RenderObject và RenderTree

- Các widget chứa các hướng dẫn tạo giao diện người dùng và khi lồng các widget với nhau, chúng sẽ tạo ra cây widget.
  - Các widget làm cấu hình cho từng phần tử được kết xuất trên màn hình.
- Các phần tử được gắn kết hiển thị trên màn hình tạo ra cây phần tử (element tree).
  - Các element có tham chiếu đến widget và chịu trách nhiệm so sánh sự khác biệt của widget.



- Tuy nhiên, bản thân widget không biết cách nó được hiển thị trên màn hình

4.6 RenderObject và RenderTree (3)



4.6 RenderObject và RenderTree (4)

Sao lại cần đến 3 tree?

Element không giữ cấu hình widget, nhưng có thể nhìn cấu hình chi tiết thông qua việc tham chiếu đến widget tương ứng.

RenderObject chứa toàn bộ logic để **rendering** widget, nên nó khá nặng nề để khởi tạo.

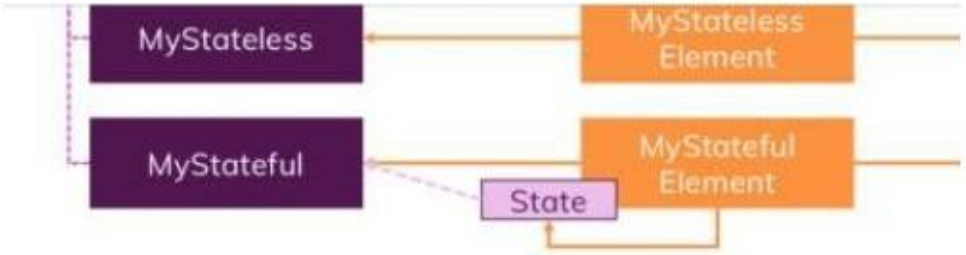
>> Khi widget trong tree thay đổi, Flutter sử dụng **Elements** để so sánh widget mới với **RenderObject** đã tồn tại, -> nếu type của widget không đổi, Flutter sẽ không tạo lại cái RenderObject nặng nề đó, thay vào đó chỉ cập nhật mutable configuration.
Widget rất nhẹ để khởi tạo, nên nó sẽ được dùng để mô tả state hiện tại.

4.6 RenderObject và RenderTree (5)

Lưu ý

There's one last important detail I'd like to touch on: State objects are actually managed by elements, not widgets. In fact, under the hood, Flutter renders based on Elements and state objects, and isn't as concerned with widgets. In the next section, we'll look deeper into how the state object interacts with elements and widgets.

StateObject thực sự được quản lý bởi element, không phải widget. Flutter render dựa trên Elements và state object, chứ nó không liên quan gì đến widget.



HẾT CHƯƠNG 3

HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.