

Dart là gì

`Dart` là một ngôn ngữ lập trình được phát triển bởi `Google`. `Dart` khá giống với `java` nên đa phần các lập trình viên `Android` chuyển dịch sang làm `Flutter` khá thuận tiện vì sự giống nhau này.

Điểm đặt biệt ở `dart` đó là hỗ trợ biên dịch cả `Just in time(JIT)` và `Ahead of time(AOT)`.

- `Ahead Of Time (AOT)`: Với `AOT` thì trình biên dịch chuyển ngôn ngữ `Dart` thẳng sang Native Code giúp hiệu năng tốt nhất có thể (tức là khi chạy chương trình, nó sẽ biên dịch từ đầu đến cuối)
- `Just In Time (JIT)`: Còn với `JIT` cho phép **hot reloading** hoạt động, giúp phát triển sản phẩm nhanh và tiện dụng hơn (được hiểu như việc debug trong ngôn ngữ khác là debug hàm nào chạy hàm đó thì ở đây nó sẽ viết đến đâu biên dịch ngay đến đấy)

Để học Flutter, chúng ta phải học Dart

Dart là một ngôn ngữ thuần OOP (hướng đối tượng)

Ngôn ngữ Dart giống giống Java, là cải tiến lên từ Javascript, cũng là hệ C nên cú pháp tương tự nhau (tương tự cả C#)

Để nắm được ngôn ngữ Dart, chúng ta phải làm quen và ghi nhớ một số **CONCEPTS** (khái niệm) quan trọng:

- Do Dart là ngôn ngữ thuần OOP nên tất cả những thứ bạn gán cho biến đều là **object**, mọi **object** đều là **instance** (thể hiện) của một **class**.
- Kể cả **số**, **method (hàm)** và **null** cũng đều là **object**. Tất cả các object đều ***kế thừa*** từ `Object` class.
- **Type** của 1 **variable** (biến) là **optional** bởi vì Dart có thể tự suy ra type dựa trên giá trị truyền vào cho biến.
- Biến **number** được hiểu là **kiểu int**. Khi bạn muốn khai báo 1 variable mà type của nó không được xác định, hãy sử dụng **type dynamic**
- Như các ngôn ngữ khác, **Dart** hỗ trợ **generic type**, ví dụ như **List** (1 danh sách kiểu số nguyên) hoặc **List** (1 danh sách các object mà type không xác định, Dart có thể chấp nhận mọi loại type).
- **Dart** hỗ trợ **top-level function** (giống như **main()**), đồng nghĩa bạn có thể sử dụng hàm đó ở bất cứ đâu mà không cần thông qua tên **class** hay bất kì **instance** của class nào cả.
- Bạn cũng có thể tạo một hàm bên trong hàm (còn gọi là **nested function** hoặc **local function**).
- Tương tự **Dart** cũng hỗ trợ **top-level variable**.
- Không giống với **Java**, **Dart** không hỗ trợ **public**, **protected** và **private**. Nếu như **identifier** (tên biến, hàm,...) bắt đầu với dấu gạch dưới (_), thì nó **private** trong **library** của nó.
- Mỗi file **.dart** được coi là 1 **library**. Identifier có thể bắt đầu bằng một chữ cái hoặc dấu gạch dưới (_).
- **Dart tools** có thể báo cho bạn 2 loại vấn đề: **warning** và **errors**.

Một chương trình Dart cơ bản:



```

1 // Định nghĩa 1 hàm (function)
2 printInteger(int aNumber) {
3     print('The number is $aNumber.');// Print to console.
4 }
5
6 // Khi chương trình được excute, hàm main sẽ được thực thi đầu tiên
7 main() {
8     var number = 42; // Khai báo và gán giá trị cho biến.
9     printInteger(number); // Gọi hàm printInteger()
10 }

```

Một số keyword thường dùng trong Dart: [link](#)

Để tìm hiểu sâu hơn về ngôn ngữ Dart và Flutter, mời các bạn đón đọc các bài viết tiếp theo.

Các biến, kiểu dữ liệu trong Dart

```

1 void main(){
2     // Numbers - (int, double) num
3     // Strings - "Hello!" (single and doule quotes)
4     // Booleans - true or false
5     // Lists - collections of items (like arrays) List<int> 0 indexed
6     // Maps - collectiosn with associated Key Value Pairs Map<String, int>
7     // runes - unicode character points
8     // symbols - #symbol (simbolic metadata)
9 }

```

Strings

1. Khái niệm:

Kiểu String trong Dart sử dụng để biểu diễn chuỗi ký tự Unicode(UTF-16) (bạn có thể sử dụng các xâu có kí tự tiếng Việt hoặc bất cứ thứ tiếng nào mà sử dụng được với mã Unicode.

Nhìn chung string trong dart giống các ngôn ngữ khác.

Cách khai báo và sử dụng:

```

1 void main(List<String> args) {
2     var s1 = 'Single quotes work well for string literals.';
3     var s2 = "Double quotes work just as well.";
4     var s3 = 'It\'s easy to escape the string delimiter.';
5     var s4 = "It's even easier to use the other delimiter.";
6     // Dùng nháy đơn hay kép đều được
7 }

```



Một vài method cần lưu ý

```
1 void main(List<String> args) {
2     /// Có thể sử dụng nháy đơn '' hoặc nháy kép "" để khai báo 1 string
3     String s = 'Hello world ';
4     String s1 = "Hello world";
5
6     /// Tách kí tự trong 1 chuỗi
7     List<String> splittedStrs = s.split(" ");
8     print("splittedStrs -> $splittedStrs"); // sẽ in ra: splittedStrs -> [Hello,
world]
9
10    /// kiểm tra 1 string có chứa 1 string khác không
11    bool hasWorldStr = s.contains("world");
12    print("hasWorldStr -> $hasWorldStr"); // sẽ in ra: hasWorldStr -> true
13
14    /// chuyển đổi tất cả kí tự về kí tự viết hoa, viết thường
15    String sUpperCase = s.toUpperCase();
16    String sLowerCase = s.toLowerCase();
17    print("sUpperCase -> $sUpperCase"); // sẽ in ra: sUpperCase -> HELLO WORLD
18    print("sLowerCase -> $sLowerCase"); // sẽ in ra: sLowerCase -> hello world
19
20    /// lấy vị trí đầu tiên của kí tự 'o'
21    int sIndexOf = s.indexOf("o");
22    print("sIndexOf -> $sIndexOf"); // sẽ in ra: sIndexOf -> 4
23
24    /// kiểm tra xem string bắt đầu với kí tự 'H'
25    bool sStartWith = s.startsWith("H");
26    print("sStartWith -> $sStartWith"); // sẽ in ra: sStartWith -> true
27
28    /// thay thế các kí tự trong 1 chuỗi
29    String sReplaced = s.replaceAll("world", "everyone");
30    print("sReplaced -> $sReplaced"); // sẽ in ra: sReplaced -> Hello everyone
31
32    /// loại bỏ khoảng trắng ở đầu/cuối string
33    String sTrimmed = s.trim();
34    print("s -> \"$s\""); // sẽ in ra: s -> "Hello world "
35    print("sTrimmed -> $sTrimmed"); // sẽ in ra: sTrimmed -> Hello world
36 }
```

int

int là kiểu số nguyên. Tùy thuộc vào bộ nhớ máy, nền tảng máy (32 bit hay 64 bit) mà kiểu số nguyên trong Dart có giá trị từ -2^{63} đến $2^{63} - 1$. Ngoài kiểu số nguyên int, trong Dart còn có kiểu số nguyên BigInt để làm việc với các số nguyên lớn.

BigInt khá giống với BigInteger trong java, phù hợp để lưu trữ các số nguyên lớn, như số tiền lạm phát ở Venezuela

```
1 void main(List<String> args) {
2     int a = 10;
3     int b = -10;
4
5     print("a = $a"); // sẽ in ra: a = 10
6     print("b = $b"); // sẽ in ra: b = -10
7 }
```

double

double là kiểu số thực. Các phiên bản cũ của Dart thì kiểu số thực bắt buộc phải có dấu . ví dụ để khai báo biến số thực a có giá trị 10 chúng ta phải viết rõ double a = 10.0 Tuy nhiên các phiên bản hiện nay đã bỏ phần dấu . này đi, bạn có thể khai báo double a = 10

Trong dart sẽ không có Float như java.

```
1 void main(List<String> args) {
2     double a = 10;
3     double b = -10.0;
4
5     int c = 10;
6     int d = -10;
7
8     int _resultInt = (a + b + c + d).toInt();
9     double _resultDouble = (a + b + c + d);
10
11     print("_resultInt -> $_resultInt"); // sẽ in ra: _resultInt -> 0
12     print("_resultDouble -> $_resultDouble"); // sẽ in ra: _resultInt -> 0.0
13 }
```

Một vài phương thức cần nhớ

Chuyển đổi từ int, double sang string:

```

1 void main(List<String> args) {
2     int a = 10;
3     double b = -10.0;
4
5     String _aStr = a.toString();
6     String _bStr = b.toString();
7
8     print("int a toString: $_aStr"); // sẽ in ra: int a toString: 10
9
10    print("double b toString: $_bStr"); // sẽ in ra: double b toString: -10.0
11 }

```

Chuyển từ int qua double:

```

1 void main(List<String> args) {
2     int a = 10;
3     double _aDouble = a.toDouble();
4
5     print("int a toDouble: $_aDouble"); // sẽ in ra: int a toDouble: 10.0
6 }

```

Chuyển từ double qua int:

```

1 void main(List<String> args) {
2     double b = -10.0;
3     int _bInt = b.toInt();
4
5     print("double b toInt: $_bInt"); // sẽ in ra: double b toDouble: -10
6 }

```

Chuyển từ String qua double, int:

```

1 void main(List<String> args) {
2     String c = "20";
3
4     int _cInt = int.parse(c);
5     double _cDouble = double.parse(c);
6
7     print("Parse String c to int: $_cInt"); // sẽ in ra: Parse String c to int: 20
8     print("Parse String c to double: $_cDouble"); // sẽ in ra: Parse String c to
9     double: 20.0
10 }

```

bool

Kiểu dữ liệu bool trong Dart có 2 giá trị là true (đúng) và false (sai) được sử dụng để thể hiện kết quả của một mệnh đề logic (các phép toán so sánh, kiểm tra, các hàm...)

```
1 void main(List<String> args) {
2   bool a = true;
3   bool b = false;
4
5   if (a) { /// cách viết tắt khi so sánh bool == true
6     print("a is $a"); // sẽ in ra: a is true
7   }
8
9   if (a == true) { /// viết kiểu này rõ nghĩa hơn
10    print("a == true"); // sẽ in ra: a == true
11  }
12
13  if (!b) { /// cách viết tắt khi so sánh bool == false
14    print("b is $b"); // sẽ in ra: b is false
15  }
16
17  if (a && !b) { /// so sánh nhiều mệnh đề, viết tắt
18    print("a == true and b == false"); // sẽ in ra: a == true and b == false
19  }
20
21  if (a || b) { /// so sánh nhiều mệnh đề, viết tắt
22    print("a == true or b == true"); // sẽ in ra: a == true and b == false
23  }
24 }
```

Array

Kiểu dữ liệu array (danh sách - List) trong Dart được sử dụng để biểu diễn cho một tập hợp các đối tượng theo một thứ tự nhất định. Kiểu array trong Dart tương đương với kiểu mảng Array trong các ngôn ngữ khác (java, kotlin).

Cách khai báo

```
1 void main(List<String> args) {
2   List strs = ["a", "b", "c"]; /// cách viết tắt
3   List<String> strs_1 = <String>["a", "b", "c"]; /// cách viết tắt rõ nghĩa, sử dụng
   [generic]
4 }
```

Một vài phương thức cần nhớ



```

1 void main(List<String> args) {
2     List strs = ["a", "b", "c"]; /// cách viết tắt
3     List<String> strs_1 = <String>["a", "b", "c"]; /// cách viết tắt rõ nghĩa, sử dụng
    [generic]
4
5     strs.add("d");
6     print("strs after adding \"d\" -> $strs");
7     // sẽ in ra: strs after adding "d" -> [a, b, c, d]
8
9     strs.remove("d");
10    print("strs after removing \"d\" -> " + strs.toString());
11    // sẽ in ra: strs_1 after adding "d" -> [a, b, c, d]
12
13    /// đoạn code trên, nhưng sử dụng [Builder Design Pattern] để viết code ngắn gọn
    hơn
14    print("strs after adding \"d\" then remove \"d\" ->
    ${strs..add("d")..remove("d")}");
15    // sẽ in ra: strs after adding "d" then remove "d" -> [a, b, c]
16 }

```

Thêm 1 array khác:

```

1 void main(List<String> args) {
2     List<String> strs_1 = <String>["a", "b", "c"]; /// cách viết tắt rõ nghĩa, sử dụng
    [generic]
3
4
5     List<String> strs_2 = ["d", "e", "f"];
6     strs_1.addAll(strs_2);
7     print("strs_1 after addAll str_2 -> $strs_1");
8     // sẽ in ra: strs_1 after addAll str_2 -> [a, b, c, d, e, f]
9 }

```

Tách các phần tử thành 1 array từ 1 array:

```

1 void main(List<String> args) {
2     List<String> strs_1 = <String>["a", "b", "c"]; /// cách viết tắt rõ nghĩa, sử dụng
    [generic]
3
4
5     List<String> strs_1_subList = strs_1.sublist(2, 4);
6     print("strs_1_subList -> $strs_1_subList");
7     // sẽ in ra: strs_1_subList -> [c, d]
8 }

```

Lấy 1 phần tử từ 1 array:



```

1 void main(List<String> args) {
2   List<String> strs_1 = <String>["a", "b", "c"
3 ]; /// cách viết tắt rõ nghĩa, sử dụng [generic]
4
5
6   /// lấy 1 phần tử tại 1 vị trí index trong array
7   String _strByIndex = strs_1[0];
8   print("_strByIndex -> $_strByIndex");
9   // sẽ in ra: _strByIndex -> a
10 }

```

map

Kiểu dữ liệu map trong dart là một đối tượng lưu trữ dữ liệu dưới dạng một cặp khóa-giá trị. Mỗi giá trị được liên kết với khóa của nó và nó được sử dụng để truy cập giá trị tương ứng của nó. Cả khóa và giá trị đều có thể là bất kỳ loại nào

Khai báo 1 map

```

1 void main(List<String> args) {
2   /// cách viết ngắn gọn
3   Map map = {"key": "value"};
4   print("map -> $map");
5   // sẽ in ra: map -> {key: value}
6
7   /// cách viết rõ nghĩa
8   Map<String, String> map1 = Map<String, String>();
9   map1["key"] = "value";
10  print("map1 -> $map1");
11  // sẽ in ra: map1 -> {key: value}
12 }

```

Trong dart cũng có hỗ trợ HashMap như các ngôn ngữ lập trình khác: java, kotlin

1 vài phương thức cơ bản:

```

1 void main(List<String> args) {
2   Map<String, String> map1 = Map<String, String>();
3   map1["key"] = "value";
4   print("map1 -> $map1"); // sẽ in ra: map1 -> {key: value}
5
6   /// 1 vài phương thức cơ bản
7   Map<String, String> map2 = Map.from(map1);
8   print("map2 -> $map2"); // sẽ in ra: map2 -> {key: value}
9

```




```

10 String key = map1.keys.first;
11 print("key from map1 -> $key"); // key from map1 -> key
12
13 String value = map1.values.first;
14 print("value from map1 -> $value"); // sẽ in ra: value from map1 -> value
15
16 /// trong map key là duy nhất, không thể có 2 key cùng giá trị
17 Map map12 = map1..addAll(map2);
18 print("map12 -> $map12"); // sẽ in ra: map12 -> {key: value}
19 }

```

Map lưu trữ theo key-value. Vậy có thể thêm 1 cặp key-value trùng với cặp key-value đã tồn tại trong map không?

```

1 void main(List<String> args) {
2     /// điều gì sẽ xảy ra nếu add 1 map với key mới?
3     Map mapWithNewValue = {"key": "value"}..addAll({"key": "newValue"});
4     print("mapWithNewValue -> $mapWithNewValue");
5     // sẽ in ra: mapWithNewValue -> {key: newValue}
6 }

```

⚠ Cần chú ý:

- Map lưu trữ dữ liệu theo key-value
- Key trong map không thể trùng. Nếu thêm 1 map khác có cùng key, value sẽ được cập nhật lại.

2. Tổng kết:

- Map lưu trữ dữ liệu theo key-value
- Key trong map không thể trùng. Nếu thêm 1 map khác có cùng key, value sẽ được cập nhật lại.

dynamic và var

Trong Dart mọi thứ đều là object. Đã là object thì luôn phải là instance của một class nào đó. Vì tất cả là đều là object nên dù là số, chữ hay bất kể loại dữ liệu nào thì giá trị mặc định của nó đều là `null`. Nhờ vậy, mọi biến số trong Dart đều là `reference type`. Cũng chính vì thế mà Dart có một loại biến dynamic chấp nhận mọi kiểu dữ liệu.

Dùng `var` để khai báo các kiểu dữ liệu:

```

1 void main(List<String> args) {
2     /// Khai báo biến var
3     /// Ưu điểm: nhanh, và không cần quan tâm tới [runtimeType]
4     var intVar = 10;
5     print("intVar -> $intVar, loại dữ liệu (variableType) -> ${intVar.runtimeType}");

```



```

6 // sẽ in ra: intVar -> 10, loại dữ liệu (variableType) -> int
7
8 var doubleVar = 10.0;
9 print("doubleVar -> $doubleVar, loại dữ liệu (variableType) ->
  ${doubleVar.runtimeType}");
10 // sẽ in ra: doubleVar -> 10.0, loại dữ liệu (variableType) -> double
11
12 var stringVar = "A";
13 print("stringVar -> $stringVar, loại dữ liệu (variableType) ->
  ${stringVar.runtimeType}");
14 // sẽ in ra: stringVar -> A, loại dữ liệu (variableType) -> String
15
16 var boolVar = 10;
17 print("boolVar -> $boolVar, loại dữ liệu (variableType) ->
  ${boolVar.runtimeType}");
18 // sẽ in ra: boolVar -> 10, loại dữ liệu (variableType) -> int
19
20 var arrayVar = [1, 2, 3];
21 print("arrayVar -> $arrayVar, loại dữ liệu (variableType) ->
  ${arrayVar.runtimeType}");
22 // sẽ in ra: arrayVar -> [1, 2, 3], loại dữ liệu (variableType) -> List<int>
23
24 var mapVar = {"key": "value"};
25 print("mapVar -> $mapVar, loại dữ liệu (variableType) -> ${mapVar.runtimeType}");
26 // sẽ in ra: mapVar -> {key: value}, loại dữ liệu (variableType) ->
  _InternalLinkedHashMap<String, String>
27 }

```

Dùng dynamic để khai báo các kiểu dữ liệu

```

1 void main(List<String> args) {
2   /// Khai báo biến dynamic
3   /// Giống var, nhưng có thể thay đổi variable type
4   dynamic dynamicA = 10;
5   print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
  ${dynamicA.runtimeType}");
6   // sẽ in ra: dynamicA -> 10, loại dữ liệu (variableType) -> int
7
8   dynamicA = 10.0;
9   print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
  ${dynamicA.runtimeType}");
10  // sẽ in ra: dynamicA -> 10.0, loại dữ liệu (variableType) -> double
11
12  dynamicA = "A";
13  print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
  ${dynamicA.runtimeType}");
14  // sẽ in ra: dynamicA -> A, loại dữ liệu (variableType) -> String
15
16  dynamicA = true;

```



```

17     print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
${dynamicA.runtimeType}");
18     // sẽ in ra: dynamicA -> true, loại dữ liệu (variableType) -> bool
19
20     dynamicA = [1, 2, 3];
21     print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
${dynamicA.runtimeType}");
22     // sẽ in ra: dynamicA -> [1, 2, 3], loại dữ liệu (variableType) -> List<int>
23
24     dynamicA = {"key": "value"};
25     print("dynamicA -> $dynamicA, loại dữ liệu (variableType) ->
${dynamicA.runtimeType}");
26     // sẽ in ra: dynamicA -> {key: value}, loại dữ liệu (variableType) ->
_InternalLinkedHashMap<String, String>
27
28
29
30     /// error khi chạy, vì dynamicA hiện tại là Map, không có phương thức toInt()
31     var a = dynamicA.toInt();
32     print("a -> $a");
33     // sẽ in ra: Class '_InternalLinkedHashMap<String, String>' has no instance
method 'toInt'
34
35     /*
36     - dynamic đúng với cái tên của nó, có thể sử dụng linh hoạt trong rất nhiều
trường hợp
37     - Nhưng đây vừa là ưu điểm, vừa là nhược điểm:
38     - Trong ví dụ trên, nếu không check [runtimeType] thì khó mà biết được
variable type của dynamic,
39     điều này dễ dẫn tới việc sử dụng sai phương thức
40     */
41 }

```

Tác dụng của var/dynamic, cùng xem ví dụ sau:

```

1 void main(List<String> args) {
2     Test test = Test().instance(); // phải chỉ rõ rằng hàm `instance` trả về đối
    tượng `Test`
3
4     var a = Test().instance(); // không cần quan tâm tới `variable type` của hàm
    `instance`
5
6     /// sử dụng var/dynamic thích hợp cho việc trung chuyển các `variable`
7     /// tức chuyển từ hàm này qua hàm khác, class này qua class khác
8 }
9
10 class Test{
11     Test instance(){return this;}
12 }

```

Const / final / static

cost

Từ khoá "const" được dùng khi giá trị của biến được biết tại thời điểm compile time và không đổi. Nói cách khác trình biên dịch sẽ biết trước giá trị nào được lưu vào biến đó.

```

1 const int x = 1;
2 //Tại thời điểm compile time, giá trị của biến x là 1 và không đổi.
3

```

Flutter cũng tự động suy ra kiểu của biến khi bạn chỉ cần khai báo biến và khởi tạo giá trị dưới dạng const.

```

1 const name = "Vietcombank";

```

Việc sử dụng từ khoá này ngoài phục vụ cho việc báo cho trình biên dịch biết biến này sẽ không bao giờ thay đổi trong suốt thời gian tồn tại của nó mà còn tác dụng cải thiện hiệu suất bằng cách khi gọi lại biến này trình biên dịch ko cần phải tạo ra 1 bản sao mới mà chỉ cần tham chiếu lại bản sao mà bạn đã tạo trước đó.

Từ khoá này được sử dụng cho compile time constant, String, number và kể cả Class. Bạn hiểu ý mình chứ, chính là Class. Hãy tưởng tượng chúng ta có 1 class Widget, và nó thực hiện việc gì đó ví dụ như loading hoặc show một dialog gì đó và không có nhu cầu thay đổi. Thì việc bạn sử dụng khai báo class widget đó là một `const` sẽ giúp tiết kiệm rất nhiều bộ nhớ.



final

final và **const** trên thực tế thì rất giống nhau, đều không thay đổi giá trị của biến, nhưng **final** ít nghiêm ngặt hơn, nó chứa các giá trị không thay đổi nhưng giá trị đó có thể không xác định trong 1 khoảng thời gian ngay cả sau khi biên dịch nhưng một khi đã xác định thì giá trị đó không bao giờ thay đổi.

```
1 final response =  
2   await http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));  
3
```

Ở ví dụ trên giá trị của biến response chưa được biết đến cho đến khi chúng ta call xong api và api trả về giá trị. Một điều cần lưu ý nữa là một instance variables chỉ có thể là **final** không thể là **const** và static variables chỉ có thể là **const**.

static

Từ khóa static dùng để khai báo biến lớp và phương thức. Nó thường quản lý bộ nhớ cho biến dữ liệu toàn cục. Các biến và phương thức static là thành viên của class thay vì một cá thể riêng lẻ. Biến hoặc các phương thức static giống nhau đối với mọi thể hiện của lớp, vì vậy nếu chúng ta khai báo thành viên dữ liệu là static thì chúng ta có thể truy cập nó mà không cần tạo một đối tượng. Đối tượng lớp không bắt buộc phải truy cập vào phương thức hoặc biến static, chúng ta có thể truy cập nó bằng cách đặt tên lớp trước biến hoặc phương thức tĩnh. Sử dụng tên lớp, chúng ta có thể gọi phương thức của lớp từ các lớp khác.

Function

Hàm là một khối lệnh thực hiện một tác vụ, khối lệnh này được dùng nhiều lần nên gom chúng lại thành một hàm. Trong Dart mọi thứ đều là đối tượng nên hàm cũng là một đối tượng (kế thừa Function).

```
1  /// viết và gọi function (hàm)  
2  /// cách gọi hàm phổ biến  
3  test("cách gọi hàm phổ biến"); // sẽ in ra: Đây là hàm test, params test: cách gọi  
   hàm phổ biến  
4  
5  void test(String test){  
6    print("Đây là hàm test, params test: $test");  
7  }
```

Cách viết khác:

```

1  /// vì trong dart, mọi thứ đều là đối tượng,
2  /// và hàm kế thừa Function, nên có thể viết như này
3  test.call("gọi qua method call()"); // sẽ in ra: Đây là hàm test, params test: gọi
qua method call()
4
5  void test(String test){
6      print("Đây là hàm test, params test: $test");
7  }

```

Function cũng có thể truyền vào như 1 params:

```

1  void main(List<String> args) {
2      /// hoặc có thể truyền vào như 1 parameter....
3      functionTest(() => test("function call function"));
4      // sẽ in ra: Đây là hàm test, params test: function call function
5
6
7      var _ret = functionTestCallback(parseStringFromInt);
8      print("Kết quả khi gọi functionTestCallback -> $_ret");
9      // sẽ in ra: Kết quả khi gọi functionTestCallback -> Kết quả thực thi function:
-1
10
11
12      var _ret1 = functionTestCallback((intStr) => parseStringFromInt("10"));
13      print("Kết quả khi gọi functionTestCallback -> $_ret1");
14      // sẽ in ra: Kết quả khi gọi functionTestCallback -> Kết quả thực thi function:
10
15  }
16
17  void test(String test){
18      print("Đây là hàm test, params test: $test");
19  }
20
21  int parseStringFromInt(String number){
22      return int.tryParse(number) ?? -1;
23  }
24
25  void functionTest(Function function){
26      function.call();
27  }
28
29  /// [functionTestCallback] sẽ thực thi như sau:
30  /// - Yêu cầu truyền vào 1 function với parameter là 1 string
31  /// - Thực thi function [callback] và trả về dữ liệu kiểu int
32  /// - Trả về kiểu dữ liệu sau khi thực thi function [functionTestCallback] dạng
String
33  String functionTestCallback(int Function(String) callback){
34      int result = callback.call(""); // "" là default value

```



Vietcombank

```

35
36     return "Kết quả thực thi function: $result";
37 }

```

Params trong function

```

1  void main(List<String> args) {
2      /// function trong dart hỗ trợ optional params,
3      /// - nếu không truyền params vào function: params sẽ nhận giá trị mặc định
4      /// - nếu truyền params vào function: params sẽ nhận giá trị được truyền vào từ
    function
5      functionWithOptionalParam();
6      // sẽ in ra: params a=a, b=b
7
8      functionWithOptionalParam(
9          b: "không phải giá trị b",
10         a: "không phải giá trị a"
11     );
12     // sẽ in ra: params a=không phải giá trị a, b=không phải giá trị b
13
14     functionWithPositionalParam("a", "b", "c");
15     // sẽ in ra: params a=a, b=b, b=c, d=
16     /// vì không truyền vào d, nên d=""
17 }
18
19 void functionWithOptionalParam({String a = "a", String b = "b"}){
20     print("params a=$a, b=$b");
21 }
22 void functionWithPositionalParam(String a, [String b = "", c = "", d = ""]){
23     print("params a=$a, b=$b, b=$c, d=$d");
24 }

```

Inline function: function trong function

```

1  void main(List<String> args) {
2      /// Inline function (function lồng function trong dart)
3      /// [_inlineFunction2] được định nghĩa bên trong function [_inlineFunction1]
4      /// Lúc này chỉ có thể call function [_inlineFunction2] trong function
    [_inlineFunction2]
5      void _inlineFunction1(){
6          print("_inlineFunction1");
7
8          void _inlineFunction2(){
9              print("_inlineFunction2");
10         }
11
12         _inlineFunction2();
13     }

```



```

14  _inlineFunction1();
15  // sẽ in ra: _inlineFunction1
16  // sẽ in ra: _inlineFunction2
17  }

```

Mở rộng function từ 1 class

```

1  void main(List<String> args) {
2    /// Dart có hỗ trợ extension như kotlin, swift...:
3    /// function [test] được viết thêm vào lớp String
4    String a = "a";
5    a.test();
6    // sẽ in ra: Mở rộng function trên lớp String
7  }
8
9  extension stringExt on String{
10   void test(){
11     print("Mở rộng function trên lớp String");
12   }
13 }

```

2. Tổng kết

- Function cũng là 1 object trong dart.
- Khi tạo inline function, cần chú ý tới việc đặt tên hàm và nên tuân thủ clean code (1 function không dài quá 20 line, 1 line không nên quá 80 kí tự)
- Function với optional params linh hoạt trong hầu hết các trường hợp.
- Function với positioned params sẽ khó mở rộng trong tương lai.

Ví dụ 1 function với positioned params, sẽ không khả thi khi cần truyền thêm params khác variable type.

Enum

Kiểu enum trong Dart còn gọi là kiểu liệt kê (kiểu liệt kê thứ tự enumerated) được sử dụng để liệt kê các giá trị hằng số. Kiểu liệt kê được khai báo bằng cách sử dụng từ khóa enum. Kiểu liệt kê enum cũng là một kiểu iterable, tức là cũng có thể duyệt tuần tự qua từng phần tử của nó.

Ví dụ khai báo enum và thêm function `getName()`

```

1  enum DayOfWeeks {
2    Monday,
3    Tuesday,
4    Wednesday,
5    Thursday,

```




```

6   Friday,
7   Saturday,
8   Sunday,
9 }
10
11 extension DayOfWeeksExt on DayOfWeeks {
12     String getName() {
13         switch (this) {
14             case DayOfWeeks.Monday:
15                 return "Thứ 2";
16             case DayOfWeeks.Tuesday:
17                 return "Thứ 3";
18             case DayOfWeeks.Wednesday:
19                 return "Thứ 4";
20             case DayOfWeeks.Thursday:
21                 return "Thứ 5";
22             case DayOfWeeks.Friday:
23                 return "Thứ 6";
24             case DayOfWeeks.Saturday:
25                 return "Thứ 7";
26             case DayOfWeeks.Sunday:
27                 return "Chủ nhật";
28             default:
29                 return "Không xác định";
30         }
31     }
32 }
33
34 void main(List<String> args) {
35     List<DayOfWeeks> dayOfWeeks = DayOfWeeks.values;
36     dayOfWeeks.forEach((element) {
37         print("day is: ${element.getName()}");
38         // sẽ in ra: các ngày từ thứ 2 -> chủ nhật
39         // day is: Thứ 2
40         // day is: Thứ 3
41         // day is: Thứ 4
42         // day is: Thứ 5
43         // day is: Thứ 6
44         // day is: Thứ 7
45         // day is: Chủ nhật
46     });
47 }

```

Vào ngày 3 tháng 3 năm 2021, team Flutter đã công bố [Flutter 2](#) and [Dart 2.12](#). Trong số đó, có một thay đổi rất quan trọng là **Sound Null Safety**. Bài viết này sẽ hướng dẫn cho bạn mọi thứ cần bản bạn cần phải hiểu rõ để chuyển code của mình sang **Sound Null Safety**.

Null safety là gì

Null safety còn được biết đến với tên gọi là `Non Nullable By Default` ở các ngôn ngữ lập trình khác. Null safety đảm bảo cho lập trình viên biết rằng, trừ khi họ muốn biến đó `null`, còn lại thì sẽ không có bất kì biến nào `null` cả. Trường hợp lập trình viên muốn biến đó `null` thì họ sẽ tạo nó là một biến `nullable`, trình phân tích và biên dịch sẽ đảm bảo rằng lập trình viên sẽ được nhắc nhở để xử lý các trường hợp mà giá trị của biến có thể `null`.

Tại sao cần dùng

Trong trường hợp thực tế chúng ta gặp phải là một giá trị của một biến bị null nhưng thực tế thì chúng không nên null trong lúc đang chạy chương trình. Điều này sẽ gây ra lỗi tương tự `... was called on null`.

Null types

Bạn có thể khai báo giá trị nullable bằng cách thêm `?`, ví dụ sau sẽ giúp bạn hiểu hơn:

```
1 | int x = 0; //Can never be null
2 | int? y = 1; //Can have null value
3 | int? z; //Can have null value, currently has null value
4 | x = null; //This line will throw an error as x can never be null
5 | y = null;
6 | z = 2;
7 | z = null;
```

Bây giờ bạn muốn tạo một class có tên là `User`, trong đó biến `name` không bao giờ được null nhưng biến `phone` có thể null.

```
1 | class User {
2 |     final String Name = "name";
3 |     String? phone;
4 | }
```

Null Checking

Giả sử bây giờ ta sử dụng biến `phone` của class `User` và lưu trữ nó ở một biến khác

```
1 | String _phone = User().phone;
```

Trình biên dịch lúc này sẽ báo lỗi

type of String? cannot be assigned to a variable of type String



Điều này là vì `phone` có thể null nhưng biến bạn vừa tạo là `_phone` thì lại không.

Để xử lý trường hợp này, chúng ta có một vài cách để giải quyết:

- Thay đổi type của `_phone` từ `String` thành `String?`

```
1 String? _phone = User().phone;
```

- Kiểm tra null cho `User().phone`

```
1 final User _user = User();
2 if(_user.phone != null){
3     String _phone = _user.phone;
4 }else{
5     debugPrint("Error");
6 }
```

Cẩn thận với biến kiểu var và dynamic

Biến `dynamic` phá vỡ null safety vì biến dynamic có thể là null mà không cần phải chỉ định thêm `?`. Thí dụ:

```
1 dynamic x;
2 int y = x; //Will compile but throw an error at runtime
3 print(y.toString()); //Will compile but throw an error at runtime
```

Tương tự như vậy nếu bạn khai báo một biến với `var` mà không khởi tạo nó và không cung cấp một kiểu rõ ràng, biến đó sẽ suy ra kiểu `dynamic` và phá vỡ toàn bộ null safety.

Các biến bắt buộc và mặc định

Giả sử bạn không muốn biến `phone` là null, và bạn muốn biến này được khởi tạo khi bất kỳ `User` nào được đăng ký. Chúng ta sẽ làm điều này ở hàm khởi tạo:

```
1 class User {
2     final String name;
3     final String phone;
4     int? age;
5     User(this.name, this.phone)
6 }
```

Ngoài ra chúng ta còn xuất hiện thêm một trường mới là `age`. Như bạn thấy thì `age` là một `nullable`, nên khi khởi tạo chúng ta có thể truyền nó là 1 thông số tùy chọn. Có cũng được mà không cũng được



```

1 class User {
2     final String name;
3     final String phone;
4     int? age;
5     User({this.name, this.phone, this.age}) // This will throw an error
6 }

```

Nếu chúng ta viết như ở trên thì sẽ hiểu `name` và `phone` là một `optional`, nó có thể là null, nhưng theo khai báo của chúng ta, nó không thể là null được. Vì vậy chúng ta cần thêm từ khóa `required` dành cho `name` và `phone`.

```

1 class User {
2     final String name;
3     final String phone;
4     int? age;
5     User({required this.name, required this.phone, this.age})
6 }

```

Làm việc với !

Nếu bạn chắc chắn rằng một biến `nullable` sẽ không null trong thời gian chạy, bạn có thể thêm `!` để biểu thị điều đó.

```

1 int? x;
2 x = 5;
3 int y = x; //Will throw type int? cannot be assigned to a variable of type int
4 int y = x!; // Will work but you have to be COMPLETELY sure

```

late

Trường hợp bạn muốn thêm một biến `id` vào class `User`. Và bạn quá mệt mỏi để xử lý `nullable` cho nó trong việc khai báo. Bạn muốn trình biên dịch tin rằng `id` sẽ không bao giờ null. Và hãy cứ kệ nó(`id`) đi. Kiểu gì cũng sẽ truyền data cho nó mà. Thì bạn có thể cân nhắc sử dụng `late`

```

1 class User {
2     late String id;
3     final String name;
4     final String phone;
5     int? age;
6     User({required this.name, required this.phone, this.age})
7 }
8 void main(){
9     final User _user = User(name: 'vcb', phone: '0900000000');
10    print(_user.id); // This won't give you any warning but will throw a
LateInitializationException during runtime and crash your app
11

```



```
12  _user.id = '1';
13  print(_user.id); // will print 1
14 }
```

Nhìn chung thì các bạn cần nhắc kỹ trước khi dùng `late` nhé.

For loop

Trong các ngôn ngữ lập trình, vòng lặp cho phép một khối mã được thực thi lặp đi lặp lại nhiều lần.

Dart cũng như các ngôn ngữ lập trình khác, đều hỗ trợ các vòng lặp

- for
- for in
- while
- do while

For: áp dụng theo các điều kiện

```
1  void main() {
2    /// vòng for cơ bản, theo index
3    for (int i = 0; i < 10; i = i + 1) {
4        print('i= $i');
5        // sẽ in ra giá trị của i từ 0 -> 9
6        /*
7            i= 0
8            i= 1
9            i= 2
10           i= 3
11           i= 4
12           i= 5
13           i= 6
14           i= 7
15           i= 8
16           i= 9
17        */
18    }
19 }
```

For in: áp dụng lặp từng phần tử trong mảng

```
1  void main(List<String> args) {
2    List ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
3    /// cũng là vòng lặp nhưng lặp qua từng giá trị trong 1 mảng
4    for (int i in ints) {
5        print("i = $i");
6        // sẽ in ra giá trị của i 0 -> 9
7    }
```



```

7      /*
8      i = 0
9      i = 1
10     i = 2
11     i = 3
12     i = 4
13     i = 5
14     i = 6
15     i = 7
16     i = 8
17     i = 9
18     */
19 }
20 }

```

While: áp dụng cho 1 điều kiện chưa thể xác định, chỉ có thể xác định khi thực thi 1 hoặc nhiều khối lệnh

```

1 void main(List<String> args) {
2     int x = 0;
3     // trong khi 1 điều kiện đúng, thực thi 1 khối lệnh
4     while (x < 10){
5         x++;
6         print("x -> $x");
7         // sẽ in ra các giá trị từ 1 -> 10,
8         // khi x = 10, while sẽ dừng, vì điều kiện để chạy while là x < 10
9     }
10 }

```

do while:

```

1 void main(List<String> args) {
2     int x = 0;
3     /// thực thi khối lệnh trước khi kiểm tra điều kiện trong while
4     do {
5         x++;
6         print("x -> $x"); // sẽ in ra giá trị của x từ 1 -> 10
7     } while (x > 0 && x < 10);
8 }

```

⚠️ Chú ý

Khi sử dụng vòng lặp, cần chú ý các điều kiện lặp để hạn chế `infinity loop` gây treo, lag ứng dụng.

Tổng kết:

Vòng lặp trong dart giống các ngôn ngữ lập trình phổ biến khác như: java, kotlin, swift, python....



Decision Making trong dart

Mệnh đề if trong dart được sử dụng để kiểm tra giá trị dạng boolean của điều kiện.

Mệnh đề này trả về giá trị **True** hoặc **False**.

Có các kiểu của mệnh đề if-else trong dart như sau:

- Mệnh đề if
- Mệnh đề if-else
- Mệnh đề if-else-if

Mệnh đề if

Mệnh đề if được sử dụng để kiểm tra giá trị dạng boolean của điều kiện. Khối lệnh sau if được thực thi nếu giá trị của điều kiện là **True**.

```
1 int a = 10;  
2 if (a == 10)  
3 { print("a == 10"); }
```

Mệnh đề if else

Mệnh đề if-else cũng kiểm tra giá trị dạng boolean của điều kiện. Nếu giá trị điều kiện là **True** thì chỉ có khối lệnh sau if sẽ được thực hiện, nếu là **False** thì chỉ có khối lệnh sau else được thực hiện.

```
1 int a = 8;  
2 if (a == 10) {  
3     print("a == 10");  
4 } else {  
5     print("a != 10");  
6 }
```

Mệnh đề if else if

Mệnh đề if-else-if cũng kiểm tra giá trị dạng boolean của điều kiện.

Nếu giá trị điều kiện if là **True** thì chỉ có khối lệnh sau if sẽ được thực hiện.

Nếu giá trị điều kiện if else nào là **True** thì chỉ có khối lệnh sau else if đó sẽ được thực hiện...

Nếu tất cả điều kiện của if và else if là **False** thì chỉ có khối lệnh sau else sẽ được thực hiện.



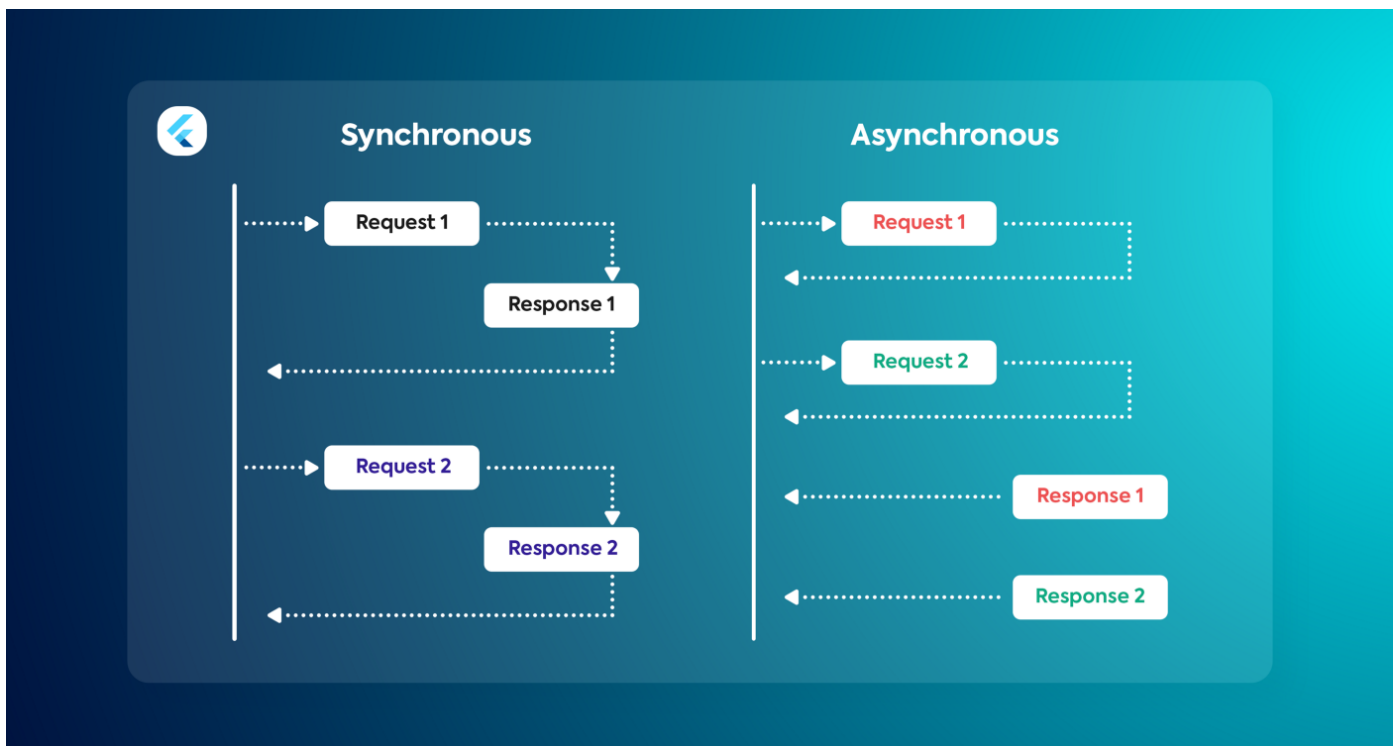
```
1  int a = 8;
2  if (a == 1) {
3    print("a == 1");
4  } else if (a == 2){
5    print("a == 2");
6  } else {
7    print("else");
8  }
```

Feature & Async/await Dart

Bài viết được dịch và viết lại từ ngữ cho dễ hiểu từ nguồn bài gốc: [Exploring Asynchronous Programming In Dart & Flutter](#)

Asynchronous Programming (Lập trình bất đồng bộ):

Lập trình bất đồng bộ (**Asynchronous Programming**) giúp bạn có thể thực hiện nhiều action trong cùng 1 lúc mà không cần theo tuần tự (xem hình phía dưới). Lập trình đồng bộ (**Synchronous Programming**) mất khoảng thời gian lâu hơn để ứng dụng phản hồi kết quả. Tuy nhiên, việc lập trình bất đồng bộ cũng có một số đánh đổi, đặc biệt là phần hiệu năng ứng dụng. Do đó, chúng ta không nên sử dụng lập trình bất đồng bộ trong tất cả các trường hợp.



Mặc dù Dart chạy single-threaded, nhưng nó có thể kết hợp với các code khác trong các thread chạy song song, riêng biệt.

Tại sao chúng ta nên sử dụng lập trình bất đồng bộ

Cải thiện hiệu năng: Một trong những lý do lớn nhất mà chúng ta cần cân nhắc sử dụng lập trình bất đồng bộ. Trường hợp chúng ta cần tính toán và xử lý mà không phải chặn việc running app hiện tại. Trong trường hợp này nếu sử dụng lập trình bất đồng bộ, bạn có thể thực hiện công việc khác và ngồi đợi tác vụ đó hoàn thành.

Code clean hơn: Hãy nhìn qua các ngôn ngữ khác mà xem. Việc bạn muốn làm việc với bất đồng bộ, bạn phải tự tạo thread riêng theo tiêu chuẩn được đặt ra.

Future

Future trong dart có 2 trạng thái. `Uncompleted` và `Completed`. `Completed Future` sẽ cho chúng ta giá trị trả về hoặc lỗi xảy ra. Với `Uncompleted Future`, nó là trạng thái chờ đợi hoạt động bất đồng bộ của hàm sẽ hoàn thành hoặc xảy ra lỗi.

Future với async và await

Hàm bất đồng bộ được đánh dấu bằng từ khoá **async**

Để chờ 1 hàm bất đồng bộ, ta có thể sử dụng từ khoá `**await**`, tuy nhiên `**await**` phải đặt trong function được khai báo với từ khoá **async**.

```
1 void main(List<String> args) {
2   test();
3 }
4 void test() async {
5   Future<String> testAsync() async {
6     await Future.delayed(const Duration(seconds: 3)); // sẽ chờ 3s trước khi chạy
    phương thức tiếp theo
7     return "testAsync has been done!";
8   }
9
10  print("begin");
11  await testAsync(); // sẽ chờ testAsync thực thi xong mới thực thi phương thức tiếp
    theo
12  print("end");
13 }
14
```

Lấy dữ liệu trả về từ Future

Có thể sử dụng `await` hoặc `Future.then`, để lấy kết quả khi Future hoàn thành.

```
1 String result = await testAsync(); // use await
2
3 String result;
4 await testAsync().then((value) => result = value); // use then
```



Xử lý lỗi trong Future

Có thể sử dụng try catch bọc ngoài Future, hoặc sử dụng `Future.catchError`, `Future.onError` **onError** trong then sẽ được ưu tiên xử lý lỗi từ Future. Nếu không dùng **onError**, **catchError** sẽ xử lý toàn bộ lỗi. Nếu dùng **onError**, **catchError** sẽ xử lý lỗi trong **onError**.

```
1 Future<String> testAsync() async {
2   int.parse("source");
3   await Future.delayed(const Duration(
4     seconds: 3)); // sẽ chờ 3s trước khi chạy phương thức tiếp theo
5   return "testAsync has been done!";
6 }
7
8 print("begin");
9 await testAsync().then((value) => null, onError: (error) {
10   print("onError -> $error");
11   throw Exception("from onError");
12 }).catchError((catchError) {
13   print("catchError -> $catchError");// catchError sẽ bắt được exception
14 }).whenComplete(() => print('onCompleted'));
15 // sẽ chờ testAsync thực thi xong mới thực thi phương thức tiếp theo
16 print("end");
17
18
```

Future.foreach

Cách thức hoạt động khá đơn giản: for từng phần tử và thực thi future.Future.foreach trả về `null` sau khi hoàn thành.

```
1 Future delayed(int second) async {
2   await Future.delayed(Duration(seconds: second));
3   return second;
4 }
5
6 List<int> steps = [1, 2, 3, 4, 5];
7 var result = await Future.forEach<int>(steps, (step) async {
8   print("current step: $step");
9
10   /// in ra step và chờ `step` second trước khi tiếp tục print
11   return await delayed(step);
12 });
13
14 print("result -> $result"); // result -> null
15
16
```

Future.wait

Future.wait sẽ thực thi 1 list future, và trả về 1 list object tương ứng. Có trả về kết quả khi hoàn thành, tùy trường hợp mà ta nên áp dụng `Future.foreach` hay `Future.wait`.

```
1 Future delayed(int second) async {
2     await Future.delayed(Duration(seconds: second));
3     return second;
4 }
5
6 Future noDelay() async { return true; }
7
8 List result = await Future.wait([
9     delayed(1), delayed(2), delayed(3),
10    noDelay(), noDelay(),
11    ]);
12
13 print("result -> $result"); // result -> [1, 2, 3, true, true]
14
15
```

Future.timeout

Giới hạn thời gian 1 Future được hoạt động. Cũng giống Future.onError. Nếu truyền vào Future.onTimeout thì sẽ ưu tiên xử lý lỗi tại onTimeout(). Nếu onTimeout có lỗi sẽ xử lý exception tại catchError

```
1 Future delayed(int second) async {
2     await Future.delayed(Duration(seconds: second));
3     return second;
4 }
5
6 var result = await
7     delayed(10).timeout(const Duration(seconds: 2)).catchError((error) {
8     return error.toString();
9 });
10 print("result -> $result"); // sẽ in ra: result -> TimeoutException after
    0:00:02.000000: Future not completed
11
12
```

```

1 var result =
2     await delayed(10).timeout(const Duration(seconds: 2), onTimeout: () {
3         return "onTimeout";
4     }).catchError((error) {
5         return error.toString();
6     });
7 print(
8     "result -> $result"); // sẽ in ra: result -> onTimeout
9

```

Future.whenComplete

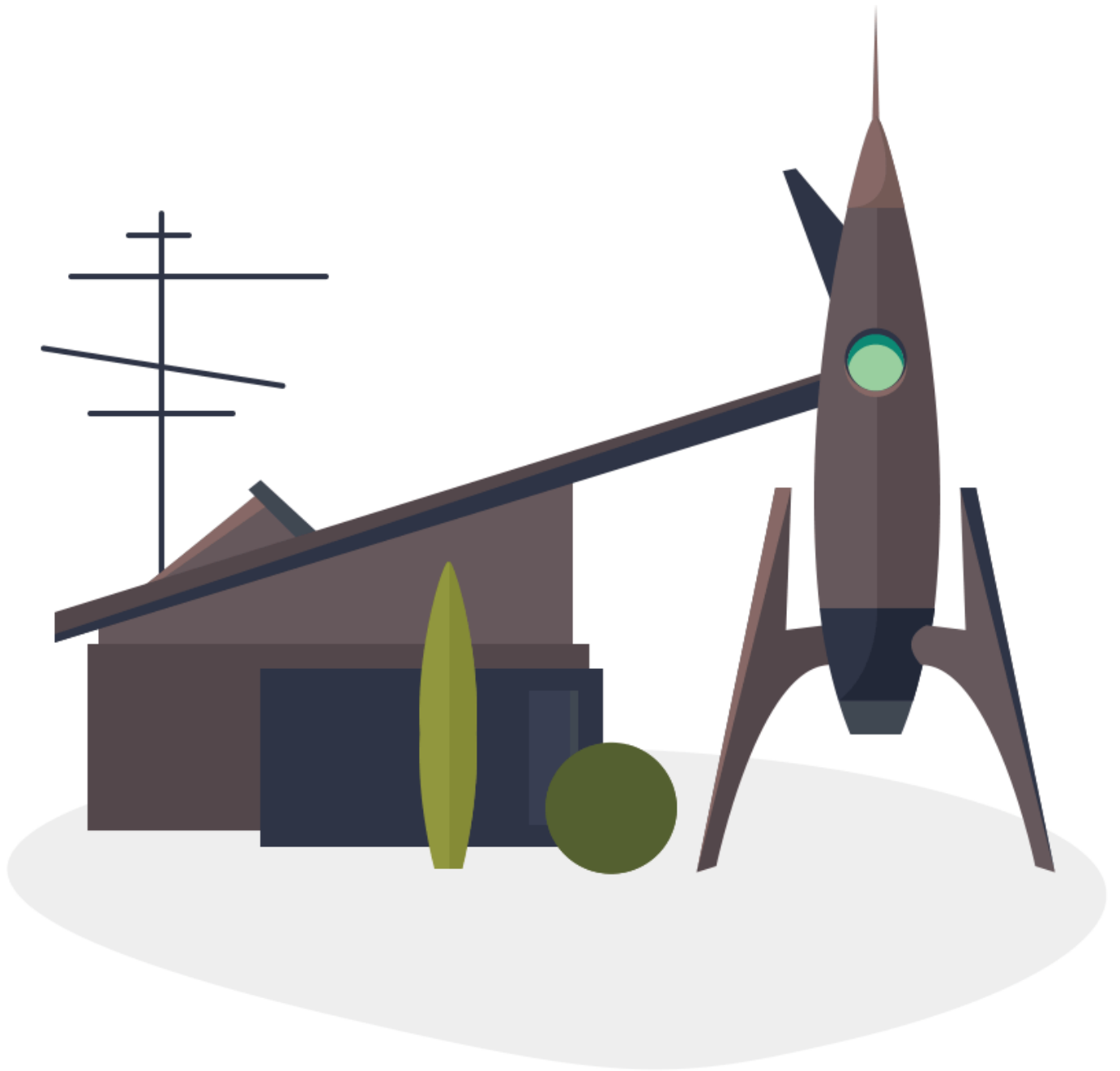
Cũng giống như finally, Future.whenComplete() luôn được gọi kể cả Future timeout, error....

```

1 Future delayed(int second) async {
2     await Future.delayed(Duration(seconds: second));
3     return second;
4 }
5
6 var result =
7     await delayed(2).whenComplete(() => print("Future has been completed!"));
8 print("result -> $result"); // will print 2
9
10

```

Stream



Stream hiểu đơn giản là một chuỗi các **events** bất đồng bộ.

Ở hình ảnh bên, **Stream** hiểu đơn giản chính là cái băng chuyền kia, thứ để trung chuyển hàng hoá (events).

Những thứ đang trượt trên băng chuyền kia chính là các **events**.

Event có thể là một data (kiểu dữ liệu bất kì), cũng có thể là một error, hoặc một trạng thái done

Stream có 2 loại:

- Single - Subscription Stream: chỉ có thể được listen 1 lần

- Broadcast Streams: có thể được listen nhiều lần

Nếu 1 Single Stream mà listen nhiều hơn 1 lần sẽ báo lỗi **"Bad state: Stream has already been listened to"**

Khởi tạo Stream với value

- **.value()** sẽ tạo ra 1 stream với giá trị mặc định.
- Khi sử dụng **.listen()** sẽ nhận được callback thay đổi của giá trị mặc định này.
- **.listen()** chỉ có thể gọi 1 lần, nếu muốn **.listen()** nhiều stream, cần sử dụng **StreamController**.
- Áp dụng cho các trường hợp có giá trị khởi tạo ban đầu.

Khởi tạo empty Stream

```
1 Stream stream1 = const Stream.empty();
2 stream1.listen((event){
3     print("onEvent -> $event");// sẽ chưa in ra gì, chỉ chạy trong tương lai, khi có
    event được add(emit) vào
4 });
```

.empty() sẽ tạo ra 1 stream không có giá trị mặc định. Áp dụng cho các trường hợp không có giá trị khởi tạo ban đầu.

Khởi tạo Stream từ 1 Future

.fromFuture() sẽ tạo ra 1 stream từ 1 Future. Event sẽ được add (emit) vào mỗi khi 1 future complete. Event có thể là data, error...

```
1 Future testDelayed()async{
2     await Future.delayed(const Duration(seconds: 3));
3     return 10;
4 }
5
6 Stream stream2 =
7     Stream.fromFuture(testDelayed());
8 stream2.listen((event) {
9     print("onEvent -> $event");// sau khi Future complete (chờ 3s) sẽ in ra:
    onEvent -> 10
10 });
```

Khởi tạo Stream từ nhiều Future

```
1 Future testDelayed(int second)async{
2     await Future.delayed(Duration(seconds: second));
3     return second;
4 }
5
6 Iterable<Future> futures = [testDelayed(1), testDelayed(2), testDelayed(3)];
7 Stream stream3 = Stream.fromFutures(futures);
```



```

8   stream3.listen((event) {
9       print("onEvent -> $event");
10      /*
11         Sau 1s sẽ chạy xong testDelayed(1) và in ra: onEvent -> 1
12         Sau 2s sẽ chạy xong testDelayed(2) và in ra: onEvent -> 2
13         Sau 3s sẽ chạy xong testDelayed(3) và in ra: onEvent -> 3
14      */
15  });

```

.fromFutures() sẽ tạo ra 1 stream từ nhiều Future. Các event sẽ được add (emit) vào mỗi khi 1 future complete. Event có thể là data, error...

Khởi tạo Stream từ 1 Stream khác

```

1   Stream sourceStream = Stream.periodic(const Duration(seconds: 1), (i) {
2       print("i -> $i");
3       return i;
4   });
5
6   Stream s = Stream.castFrom(sourceStream);
7   s.listen((event) {
8       print("onEvent -> $event");
9   });

```

.castFrom() sẽ tạo ra 1 stream từ 1 Stream khác.

Chú ý: Stream gốc phải chưa gọi listen().

StreamSubscription

Khi sử dụng `stream.listen` sẽ trả về 1 `instance` của `StreamSubscription`. `StreamSubscription` có các phương thức để điều khiển các events ra, vào stream như:

- `onError()`
- `onData()`
- `onDone`
- `pause()`
- `resume()`

StreamSubscription.pause(), .resume()

```

1   Stream stream = Stream.periodic(const Duration(seconds: 1), (i) {
2       print("after $i (s)");
3       return i;
4   });
5
6   late StreamSubscription subscription;

```



```

7 subscription = stream.listen((event) async {
8   print("onEvent -> $event");
9   if (event == 3) {
10    subscription.pause(); // sẽ dừng lại, không listen nữa
11    await Future.delayed(const Duration(seconds: 3));
12    subscription.resume(); // sẽ tiếp tục listen
13   }
14 });

```

.pause() sẽ tạm ngưng việc listen các events.

resume() sẽ tiếp tục việc listen các events.

Các events sẽ không được push tới hàm onListen() nếu không truyền **resumeSignal**

```

1 StreamController streamController = StreamController();
2 StreamSubscription streamSubscription = streamController.stream.listen((event) {
3   print("stream onListen: $event");
4 });
5 streamController.add("event"); // StreamController sẽ giới thiệu ở slide sau
6 streamController.add("event1"); // sẽ add "event" vào stream
7
8 /// nếu sử dụng streamSubscription.pause() không truyền params resumeSignal,
9 /// toàn bộ events đã add trước đó sẽ không được push to listen()
10 /// ở đây ngữ cảnh là đã add events vào stream, sau đó mới call .pause()
11 streamSubscription.pause();
12
13 streamController.add("1111");
14 streamController.add("2222");

```

Khi truyền **resumeSignal()**, hàm listen() sẽ được gọi khi **resumeSignal()** thực thi xong. Ở ví dụ là sẽ chờ 5s rồi mới print ra các events đã được add.

```

1 StreamController streamController = StreamController();
2 StreamSubscription streamSubscription = streamController.stream.listen((event) {
3   print("stream onListen: $event");
4 });
5 streamController.add("event"); // StreamController sẽ giới thiệu ở slide sau
6 streamController.add("event1"); // sẽ add "event" vào stream
7
8 /// nếu sử dụng streamSubscription.pause() không truyền params resumeSignal,
9 /// toàn bộ events đã add trước đó sẽ không được push to listen()
10 /// ở đây ngữ cảnh là đã add events vào stream, sau đó mới call .pause()
11 streamSubscription.pause(Future.delayed(const Duration(seconds: 5)));
12
13 streamController.add("1111");

```



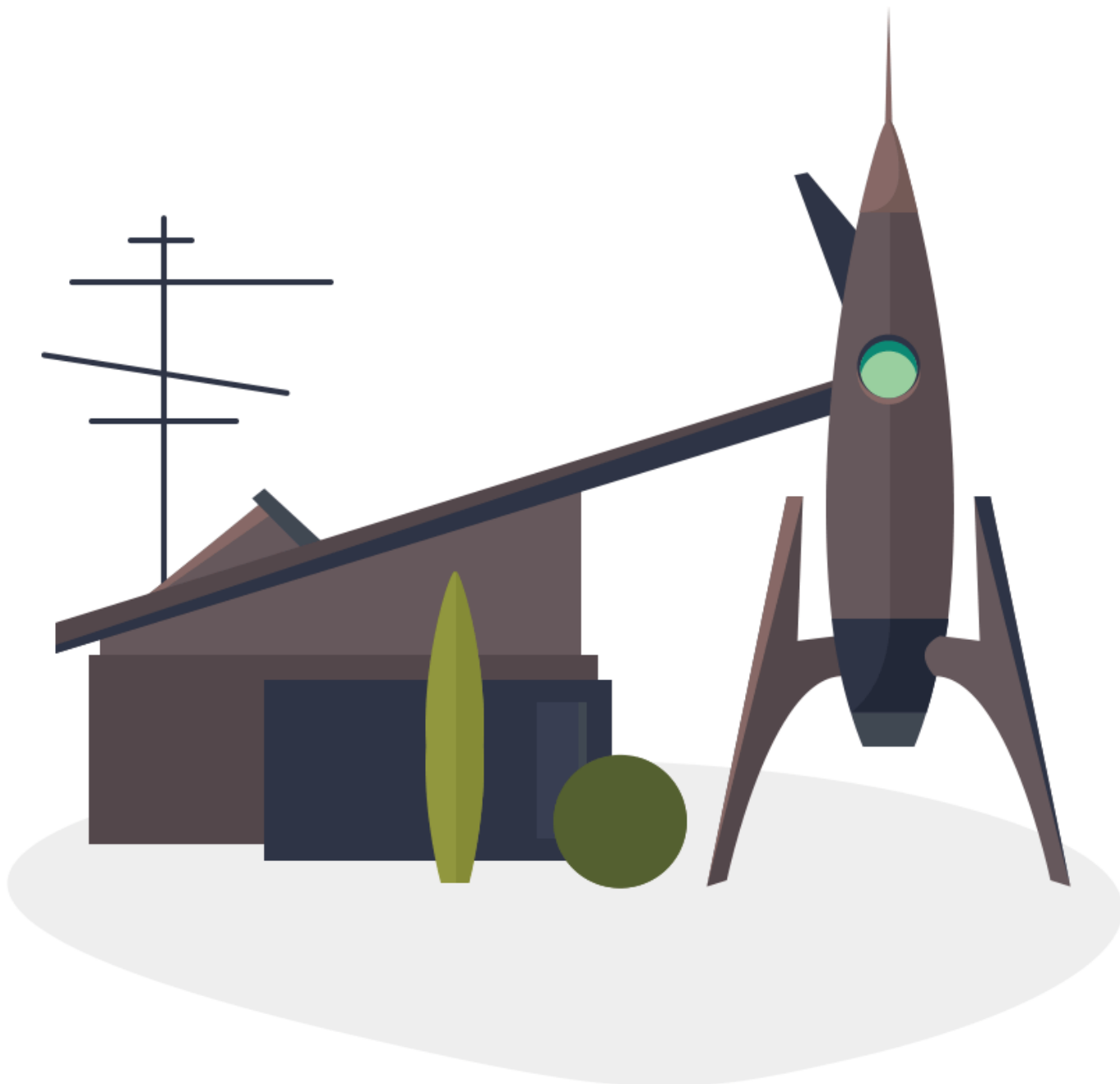
```
14 streamController.add("2222");
```

StreamSubscription.onData()

```
1 Stream stream = Stream.periodic(const Duration(seconds: 1), (i) {
2   print("after $i (s)");
3   return i;
4 });
5
6 late StreamSubscription subscription;
7 subscription = stream.listen((event) async {
8   print("onEvent -> $event");
9 });
10
11 subscription.onData((data) {
12   print("onData $data");
13 });
```

- Khi sử dụng StreamSubscription.onData(), sẽ thay thế phần onListen() của stream hiện tại.
- Áp dụng cho việc muốn thay đổi phần callback listen event, nhưng không muốn khởi tạo lại 1 stream mới.
- Trong ví dụ bên, chỉ chạy .onData() chứ không chạy .listen()

StreamController



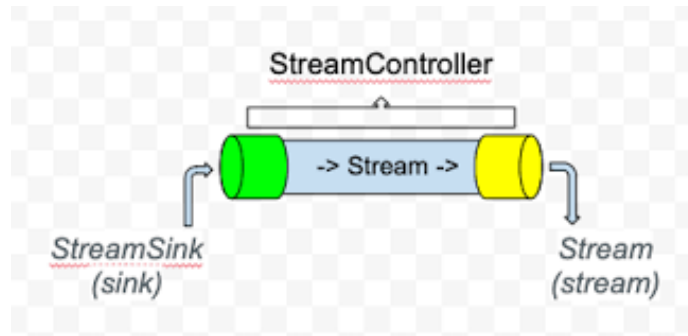
\

Ở hình ảnh trên, **Stream** hiểu đơn giản chính là cái băng truyền kia, thứ để trung chuyển hàng hoá (events). **Stream** chỉ nhận event 1 cách thụ động (từ khi khởi tạo)

Vậy có cách nào để làm cho **Stream** linh hoạt hơn, có thể chủ động nhận, thêm events?

StreamController bao gồm 2 thành phần chính:

- Sink: kiểm soát đầu vào (input)
- Stream: nơi trung chuyển dữ liệu



.sink.add() tương đương với .add()

```

1 StreamController streamController = StreamController();
2 StreamSubscription streamSubscription = streamController.stream.listen((event) {
3     print("stream onListen: $event");
4 });
5 streamController.sink.add("event");
6 streamController.sink.add("event1"); // sẽ add "event1" vào stream
7 streamController.add("1111");
8 streamController.add("2222");
9 streamController.addStream(Stream.fromIterable([1,2,3]));
10 streamController.addError(Stream.fromIterable([1,2,3]));

```

StreamTransform

StreamTransformer giúp biến đổi data của 1 stream trước khi trả về dữ liệu

Nhớ chú ý các generic in, out.

```

1 import 'dart:async';
2
3 void main(List<String> args) {
4     Stream<int> s = Stream.fromIterable([1, 2, 3, 4, 5, 6, 7, 8, 9]);
5
6     StreamTransformer<int, String> streamTransformer =
7         StreamTransformer.fromHandlers(
8             handleData: (data, sink) => sink.add("_$data"),
9         );
10
11     Stream<String> stream = s.transform(streamTransformer);
12
13     stream.listen((event) {
14         print(event);
15     });
16 }

```



Vietcombank

Các từ khoá `async`, `sync`, `yield`, `yield*`

- **`async*`** cũng giống từ khoá **`async`**, để đánh dấu hàm bất đồng bộ, nhưng variable type trả về là **`Stream`**.
- **`sync*`** để đánh dấu hàm đồng bộ, nhưng variable type trả về là **`Iterable`**.
- **`yield`** sử dụng trong function được đánh dấu với từ khoá **`async*`** hoặc **`sync*`** để phát ra 1 giá trị. Nếu dùng với từ khoá **`sync*`** sẽ phát ra 1 giá trị được add vào **`Iterable`**. Nếu sử dụng với từ khoá **`async*`** sẽ phát ra 1 giá trị được add vào **`Stream`**.
- **`yield*`** sử dụng trong function được đánh dấu với từ khoá **`async*`** hoặc **`sync*`**. Nếu dùng với từ khoá **`sync*`** sẽ phát ra 1 giá trị **`Iterable`**, và add toàn bộ giá trị vào **`Iterable`** trả về. Dùng để phát ra 1 giá trị **`Stream`** được add vào **`Stream`** trả về của function.

Ví dụ về `async*`, `yield`

```
1 import 'dart:async';
2
3 Stream<int> countStream(int max) async* {
4   for (int i = 1; i <= max; i++) {
5     yield i; // lặp từ 1 -> max, mỗi lần lặp add 1 giá trị vào stream
6   }
7 }
8
9 Future<int> sumStream(Stream<int> stream) async {
10  int sum = 0;
11  await for (int i in stream) {
12    sum += i;
13  }
14  return sum;
15 }
16
17 main() async {
18   var stream = countStream(20); // sau khi chạy sẽ có stream từ 1 > 20
19   var sum = await sumStream(stream);
20   print(sum); // sẽ in ra 210
21 }
22 /// async* cũng giống từ khoá `async`, để đánh dấu hàm bất đồng bộ, nhưng variable
23 /// type trả về là Stream.
24 /// yield sử dụng trong function được đánh dấu với từ khoá `async*` hoặc `sync*` để
25 /// phát ra 1 giá trị.
26 /// Nếu sử dụng với từ khoá `async*` sẽ phát ra 1 giá trị được add vào Stream.
```

Ví dụ về `async*`, `yield*`

```
1 import 'dart:async';
2
3 Stream<int> countStream(int max) async* {
4   List<int> arrs = [];
```



```

5   for (var i = 0; i < max; i++) { arrs.insert(i, i); }
6   yield* Stream.fromIterable(arrs); // lấy từng giá trị của
Stream.fromIterable(arrs) và trả về
7   yield* Stream.fromIterable([17, 18, 19, 20]);
8   }
9
10  Future<int> sumStream(Stream<int> stream) async {
11    int sum = 0;
12    await for (int i in stream) {
13      sum += i;
14    }
15    return sum;
16  }
17
18  main() async {
19    var stream = countStream(17);
20    var sum = await sumStream(stream);
21    print(sum); // sẽ in ra 210
22  }
23  /// async* cũng giống từ khoá `async`, để đánh dấu hàm bất đồng bộ, nhưng variable
type trả về là Stream.
24  /// yield* sử dụng trong function được đánh dấu với từ khoá `async*` hoặc `sync*`.
25  // Nếu dùng với từ khoá `async*` sẽ phát ra 1 giá trị Stream, và add toàn bộ giá
trị của Stream này vào Stream trả về.

```

Ví dụ về sync*, yield

```

1   void main(List<String> args) {
2     print(testYield()); // sẽ in ra: (1, 2, 3, 4, 5)
3   }
4
5   Iterable testYield() sync* {
6     yield 1; // add 1 vào Iterable
7     print("xử lý logic ở đây");
8     yield 2;
9     print("xử lý logic tiếp trước khi yield");
10    yield 3;
11    yield 4;
12    yield 5;
13  }
14  /// sync* cũng giống từ khoá `sync`, để đánh dấu hàm đồng bộ, nhưng variable type
trả về là Iterable.
15  /// yield sử dụng trong function được đánh dấu với từ khoá `async*` hoặc `sync*` để
phát ra 1 giá trị.
16  /// Nếu dùng với từ khoá `sync*` sẽ phát ra 1 giá trị được add vào Iterable.

```

Ví dụ về sync*, yield*

```

1 void main(List<String> args) {
2     int max = 100;
3     List<int> arrs = [];
4     for (var i = 0; i < max; i++) { arrs.insert(i, i); }
5     print(yieldWay(arrs));
6 }
7
8 Iterable yieldWay(List<int> arrs) sync* {
9     // yield* Iterable: add toàn bộ data từ Iterable con sang Iterable cha
10    yield* arrs.where((element) => element % 2 == 0);
11    yield* arrs.where((element) => element % 3 == 0);
12    yield* arrs.where((element) => element % 5 == 0);
13 }
14 /// sync* cũng giống từ khoá, để đánh dấu hàm đồng bộ, nhưng variable type trả về
    là Iterable.
15 /// yield* sử dụng trong function được đánh dấu với từ khoá `async*` hoặc `sync*`.
16 /// Nếu dùng với từ khoá `sync*` sẽ phát ra 1 giá trị Iterable, và add toàn bộ giá
    trị vào Iterable trả về.
17

```

So sánh với cách viết truyền thống

Cách viết cơ bản

```

1 void main(List<String> args) {
2     int max = 100;
3     List<int> arrs = [];
4     for (var i = 0; i < max; i++) { arrs.insert(i, i); }
5     print(normalWay(arrs));
6 }
7
8 Iterable normalWay(List<int> arrs) {
9     return [
10    // [...Iterable] dùng cho 1 Iterable, sẽ add toàn bộ data của Iterable con vào
    Iterable cha
11    ...arrs.where((element) => element % 2 == 0), // thêm các phần tử của mảng các
    số chia hết cho 2
12    ...arrs.where((element) => element % 3 == 0), // thêm các phần tử của mảng các
    số chia hết cho 3
13    ...arrs.where((element) => element % 5 == 0), // thêm các phần tử của mảng các
    số chia hết cho 2
14    ];
15 }

```

Cách viết sử dụng sync*



```

1 void main(List<String> args) {
2     int max = 100;
3     List<int> arrs = [];
4     for (var i = 0; i < max; i++) { arrs.insert(i, i); }
5     print(yieldWay(arrs));
6 }
7
8 Iterable yieldWay(List<int> arrs) sync* {
9     yield* arrs.where((element) => element % 2 == 0);
10    yield* arrs.where((element) => element % 3 == 0);
11    yield* arrs.where((element) => element % 5 == 0);
12 }

```

1 vài Ví dụ khác

```

1 void main() {
2     testStream().listen(println, onError: println, onDone: () => println('Done!'));
3 }
4
5 Stream<int> testStream() async* {
6     yield 10; // emit ra giá trị 10
7     await Future.delayed(const Duration(seconds: 2)); // chờ 2s
8     yield* Stream.fromIterable([1, 2, 3]); // add toàn bộ phần tử của 1 stream
9     throw const FormatException('FormatException'); // thử throw Exception
10    yield 13; // hàm này đã xảy ra Exception nên số 13 không được phát ra
11 }
12
13 void println(Object value) {
14     print(value.toString());
15 }

```

Sử dụng **yield*** kết hợp với **sync*** sẽ dễ dàng viết code hơn, dễ xử lý hơn.

Thích hợp cho các bài toán xử lý logic nhiều. Không cần tạo Iterable, khi cần phát ra giá trị, chỉ cần **yield***.

Tổng kết

- 1 Cách nhớ đơn giản:
- 2 **async*** sử dụng cho các hàm bất đồng bộ.
- 3 **sync*** sử dụng cho các hàm đồng bộ.
- 4 **yield** sử dụng trong function với từ khoá **async*** hoặc **sync*** kèm 1 giá trị. Giá trị này được add vào **Iterable** hoặc **Stream**.
- 5 **yield*** sử dụng trong function với từ khoá **async*** hoặc **sync*** kèm 1 **Iterable** hoặc 1 **Stream**. Toàn bộ giá trị sẽ được add vào **Iterable** và **Stream** trả về của function.

StreamBuilder

StreamBuilder là một Widget trong Flutter nhận vào một Stream và trả ra kết quả nhận được mỗi khi stream listen 1 event để người dùng render Widget tương ứng lên màn hình. StreamBuilder gồm 2 thành phần chính:

- stream: là stream sẽ listen
- builder: là widget hiển thị kết quả lên trên giao diện cho người dùng.

```
1 StreamBuilder(  
2   stream: stream,  
3   builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {  
4     return Text(snapshot.toString());  
5   },  
6 );
```

AsyncSnapshot

Khi sử dụng StreamBuilder, có thể sử dụng AsyncSnapshot để kiểm tra các trạng thái của Future:

- ConnectionState.none: khi `future` không được truyền vào
- ConnectionState.waiting: khi `future` đang được thi
- ConnectionState.done: khi `future` hoàn thành. Có thể sử dụng `.hasData`, `.hasError` để kiểm tra dữ liệu trả về có hay không

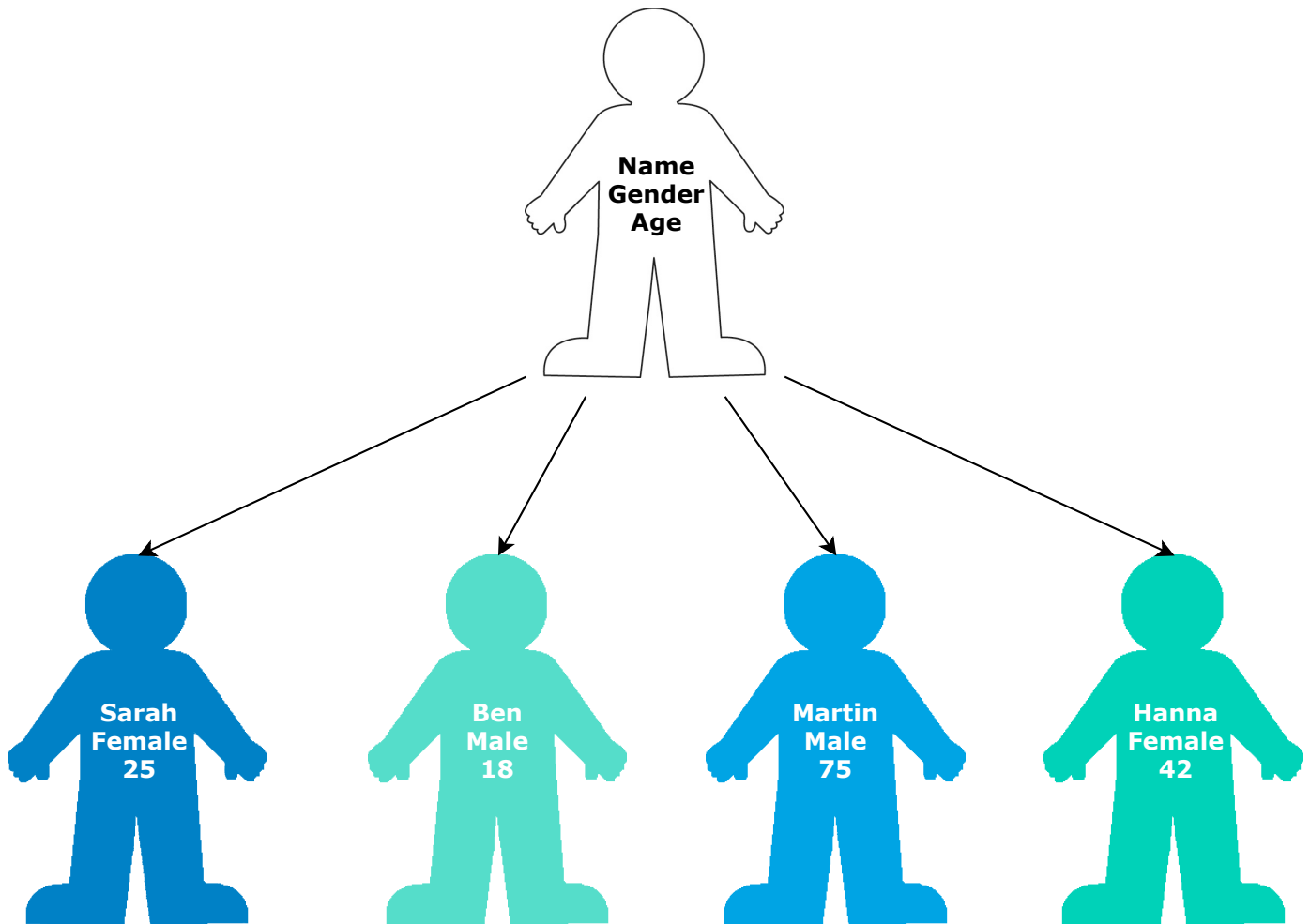
```
1 final Stream s = Stream.periodic(const Duration(seconds: 2), (i) => i);  
2  
3 @override  
4 Widget build(BuildContext context) {  
5   return StreamBuilder(  
6     stream: s,  
7     builder: (BuildContext _, AsyncSnapshot<dynamic> snapshot) {  
8       if(snapshot.connectionState == ConnectionState.waiting){  
9         return const Text("Loading, please wait...");  
10      }  
11      if (snapshot.hasData) { return Text("After ${snapshot.data}s"); }  
12      return Text(snapshot.toString());  
13    },  
14  );  
15 }
```

Dart: Classes

1. Classes là những bản vẽ thiết kế



PERSON



Mỗi người đều có một cái tên, giới tính và tuổi đó được gọi là thuộc tính hay là biến. Và con người cũng thực hiện những hoạt động như là đi bộ và nói chuyện,... đó được gọi là phương thức. Class của con người sẽ bao gồm những phương thức và thuộc tính đó. Để tạo ra một người instance thì chúng ta cần thuộc tính và phương thức riêng biệt cho từng người. Ví dụ: Tên: Sarah, tuổi: 25, giới tính: nữ có thể thực hiện những hoạt động như đi lại, nói chuyện,...

2. Classes tích hợp sẵn và classes do người dùng tự định nghĩa.

Classes trong Scala được chia thành hai loại: các classes tích hợp sẵn và các classes do người dùng tự định nghĩa.

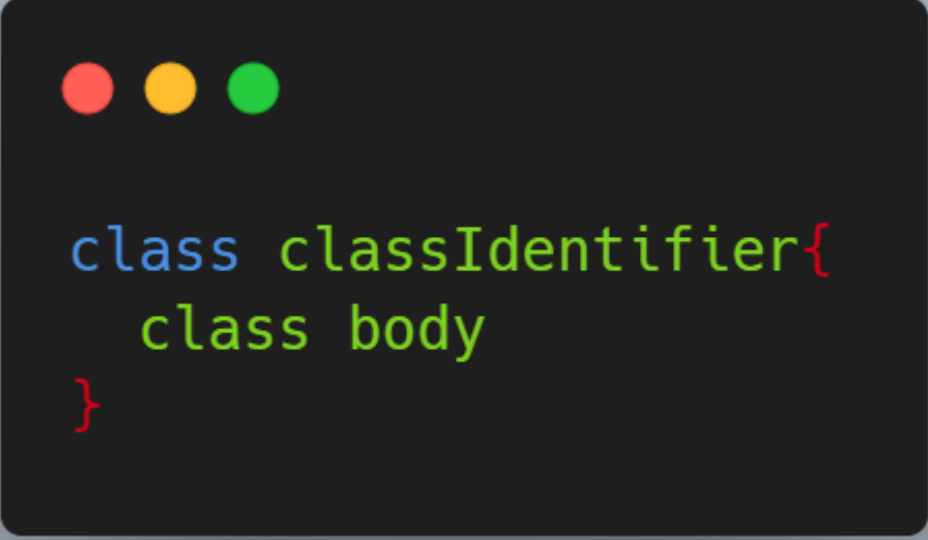
Bạn có nhớ khi chúng ta thảo luận về `List`? Tất cả các phương thức mà chúng ta đã thảo luận thực sự là một phần của class `List` tích hợp sẵn của ngôn ngữ Dart.

Khi chúng ta định nghĩa một danh sách thực ra chúng ta đang tạo một instance, tức là đối tượng, của lớp `List`.

Mọi đối tượng là một instance của một class và tất cả các class đều đi xuống từ class cao nhất trong hệ thống phân cấp class, class `Object`.

3. Tạo một Class trong Dart

Từ khóa `class` sẽ được sử dụng để định nghĩa một class trong ngôn ngữ Dart. Sau từ khóa là một nhận dạng, một cái tên do bạn tự chọn. Nội dung của class sẽ nằm bên trong dấu ngoặc nhọn `{ }`



```
class classIdentifier{  
  class body  
}
```

Nội dung của class sẽ bao gồm các biến instance và các phương thức.

Các biến instance

Class `Person` của chúng ta có 3 biến instance. Đây là cách mà bạn sẽ khai báo biến instance trong ngôn ngữ Dart.

```
1 class Person{  
2   String name; // Declare name, initially null.  
3   String gender; // Declare gender, initially null.  
4   int age = 0; // Declare age, initially 0.  
5 }
```

Tất cả các biến instance không được gán giá trị đều có giá trị là `null`

Các phương thức instance

Có rất nhiều kiểu của phương thức mà bạn có thể sử dụng trong class tuy nhiên trong series này chúng ta chỉ tập trung vào các phương thức instance.

Các phương thức instance tập trung vào các đối tượng có thể truy cập các biến instance

Phương thức của chúng ta là walking và talking sẽ chỉ in tên của người đang đi bộ hoặc nói chuyện tương ứng.

```
1 class Person{
2     String name; // Declare name, initially null.
3     String gender; // Declare gender, initially null.
4     int age = 0; // Declare age, initially 0.
5
6     walking() => print('$name is walking');
7     talking() => print('$name is talking');
8 }
```

Và chúng ta đã tạo được class đầu tiên với ba biến instance và hai phương thức.

4. Đối tượng của một class

Sau khi một class đã được định nghĩa, bạn có thể tạo một đối tượng bằng cách sử dụng từ khóa `new` theo sau là nhận dạng của class



```
new classIdentifier()
```

OR

```
classIdentifier()
```

Chúng ta tạo ra các đối tượng vì muốn làm việc với chúng theo một cách nào đó. Vì lý do này, chúng ta gán cho đối tượng một biến.

Hãy khởi tạo lớp `Person` nhé!

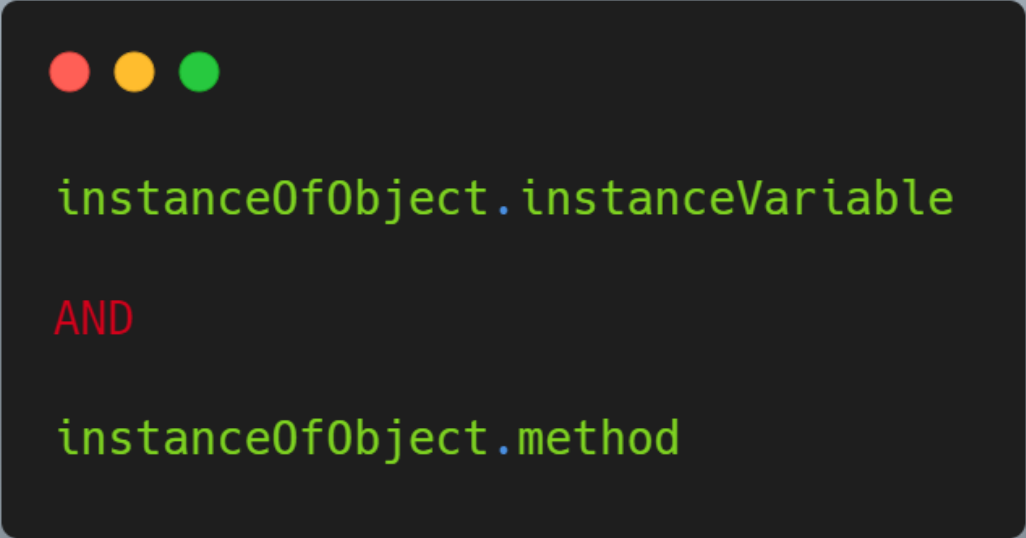
```

1  class Person{
2      String name;
3      String gender;
4      int age = 0;
5
6      walking() => print('$name is walking');
7      talking() => print('$name is talking');
8  }
9
10 int main() {
11     // Creating an object of the Person class
12     var firstPerson = Person();
13 }

```

Sử dụng class members

Bây giờ chúng ta đã có đối tượng `firstPerson`. Tiếp theo chúng ta sẽ tìm hiểu cách sử dụng các biến và phương thức instance. Trong ngôn ngữ Dart, hãy sử dụng dấu chấm (.) để tham chiếu đến một biến hoặc phương thức instance



```

instanceOfObject.instanceVariable

AND

instanceOfObject.method

```

Nếu bạn khởi tạo một biến instance ở nơi nó được khai báo, giá trị sẽ được đặt khi instance được tạo. Vì vậy, ngay sau khi `firstPerson` được tạo, giá trị của `age` đã được đặt bằng 0. Hãy đặt giá trị của `name` và `gender` bằng toán tử dấu chấm trong khi cũng chỉ định lại `age` một giá trị mới.

```

1  class Person{
2      String name;
3      String gender;
4      int age = 0;
5
6      walking() => print('$name is walking');
7      talking() => print('$name is talking');
8  }
9
10 int main() {
11     var firstPerson = Person();
12
13     firstPerson.name = "Sarah";
14     firstPerson.gender = "female";
15     firstPerson.age = 25;
16
17     print(firstPerson.name);
18     print(firstPerson.gender);
19     print(firstPerson.age);
20 }

```

```

1  Output:
2  Sarah
3  female
4  25

```

Khi bạn gọi một phương thức tức là bạn gọi nó trên một đối tượng. Phương thức đó có quyền truy cập vào các phương thức và biến instance của đối tượng đó. Chúng ta hãy gọi phương thức walking và talking và xem điều gì sẽ xảy ra.

```

1  class Person{
2      String name;
3      String gender;
4      int age = 0;
5
6      walking() => print('$name is walking');
7      talking() => print('$name is talking');
8  }
9
10 int main() {
11     var firstPerson = Person();
12
13     firstPerson.name = "Sarah";
14     firstPerson.gender = "female";
15     firstPerson.age = 25;

```



```
16
17     firstPerson.walking();
18     firstPerson.talking();
19 }
```

```
1 Output:
2 Sarah is walking
3 Sarah is talking
```

`walking()` được gọi trên đối tượng `firstPerson` và in cùng biến `name`. Vì phương thức được gọi trên một đối tượng nên nó xác định xem biến được sử dụng có phải là một trong các biến instance của đối tượng đó hay không. Vì `name` được định nghĩa trên đối tượng `firstPerson`, phương thức sẽ in ra giá trị của nó.

Trong đoạn code trên, `firstPerson.walking()` lấy giá trị của `Sarah` và in ra `Sarah is walking`. `firstPerson.talking()` cũng thực hiện tương tự và in ra `Sarah is talking`.

Nhiều đối tượng trên cùng một class

Vì các class cung cấp code có thể sử dụng lại, nên có nghĩa là chúng ta có thể tạo nhiều đối tượng bằng cách sử dụng cùng một class. Cùng với `firstPerson`, hãy tạo thêm các đối tượng của class `Person`.

```
1  class Person{
2      String name;
3      String gender;
4      int age = 0;
5
6      walking() => print('$name is walking');
7      talking() => print('$name is talking');
8  }
9
10 int main() {
11     var firstPerson = Person();
12
13     firstPerson.name = "Sarah";
14     firstPerson.gender = "female";
15     firstPerson.age = 25;
16
17     // Creating an object of the Person class
18     var secondPerson = Person();
19     secondPerson.name = "Ben";
20
21     // Creating an object of the Person class
22     var thirdPerson = Person();
23     thirdPerson.name = "Martin";
24 }
```




```

25 // Creating an object of the Person class
26 var fourthPerson = Person();
27 fourthPerson.name = "Hannah";
28
29 // Driver Code
30 print(firstPerson.name);
31 print(secondPerson.name);
32 print(thirdPerson.name);
33 print(fourthPerson.name);
34 }

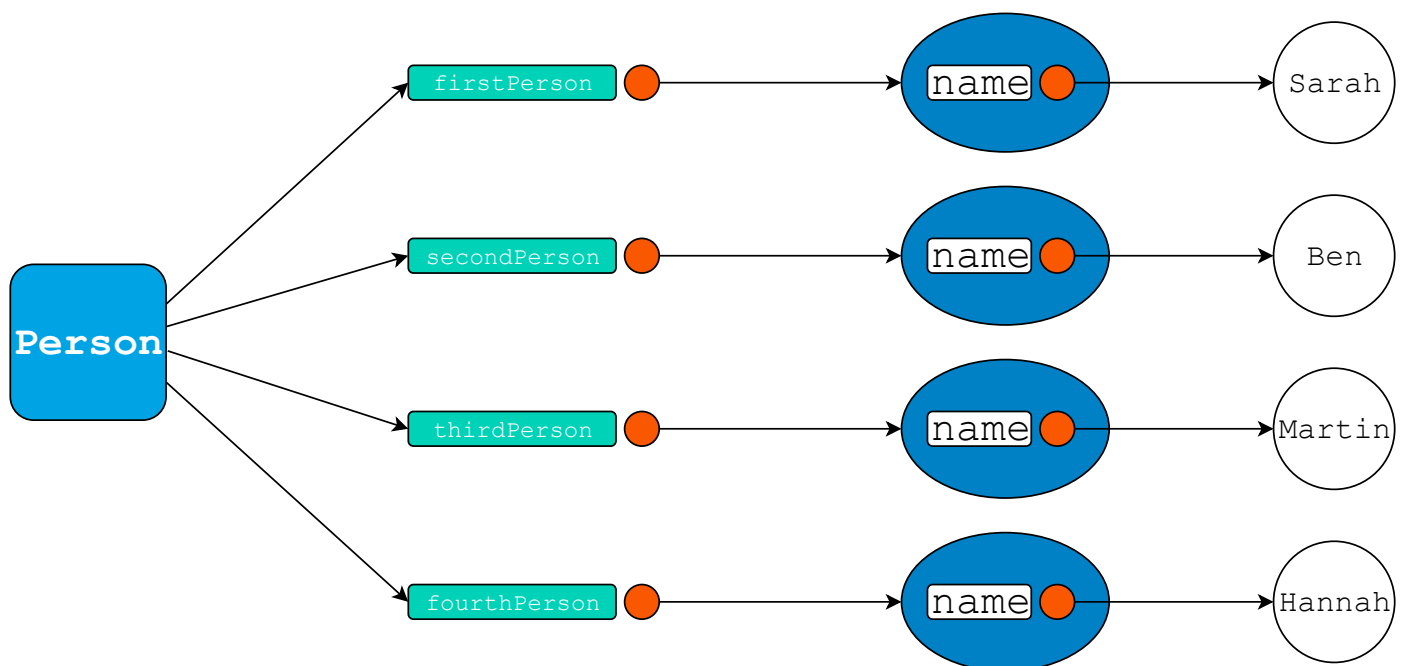
```

```

1 Output:
2 Sarah
3 Ben
4 Martin
5 Hannah

```

Mặc dù có nhiều biến `name`, nhưng tất cả chúng đều được tham chiếu bởi các đối tượng khác nhau, do đó, việc sửa đổi một biến sẽ không sửa đổi các biến khác. Đây là lý do tại sao các thuộc tính được gọi là các biến instance, bởi vì mỗi đối tượng có một tập hợp các biến đó riêng của mình.



5. Constructors

Trong ngôn ngữ Dart, **constructor** là những hàm đặc biệt của một class có nhiệm vụ khởi tạo các biến instance của class đó.

Một constructor phải có cùng tên với class mà nó đang được khai báo và vì nó là một hàm nên nó có thể được tham số hóa.

Tuy nhiên, không giống như các hàm thông thường, các constructor không có giá trị trả về, do đó không thể có kiểu trả về.

Dart cung cấp nhiều loại constructor. Trong series này, chúng ta sẽ tìm hiểu về hai trong số chúng

1. Generative Constructor
2. Named Constructor

Generative Constructor

Dạng phổ biến nhất của một constructor đó chính là generative constructor, tạo ra một instance mới của một class.

```
1 class Person{
2     String name;
3     String gender;
4     int age;
5
6     // Generative Constructor
7     Person(String nameC, String genderC, int ageC){
8         this.name = nameC;
9         this.gender = genderC;
10        this.age = ageC;
11    }
12
13    walking() => print('$name is walking');
14    talking() => print('$name is talking');
15 }
16
17 int main() {
18     var firstPerson = Person("Sarah", "Female", 25);
19     print(firstPerson.name);
20     print(firstPerson.gender);
21     print(firstPerson.age);
22 }
```

TEXT

copy

```
1 Output:
2 Sarah
3 Female
4 25
```

Như đã thảo luận ở trên, chúng ta có thể tạo nhiều instance của một class duy nhất. Từ khóa `this` để cập đến instance hiện tại.

Trên dòng 8 (`this.name = nameC;`), chúng ta đang gán giá trị `nameC` cho biến instance `name` của instance hiện tại.

Ở dòng 18, chúng ta đang tạo một instance của class `Person` bằng cách sử dụng generative constructor. Bây giờ thay vì gán riêng từng giá trị cho các biến instance, tất cả những gì chúng ta phải làm là chuyển chúng cho constructor và nó sẽ thực hiện phần còn lại.

Một generative constructor không yêu cầu bất kỳ nội dung hàm nào. Chúng ta có thể gán `this.name` làm tham số. Điều này giúp viết mã ngắn gọn hơn.

```
1 class Person{
2     String name;
3     String gender;
4     int age;
5
6     // Generative Constructor
7     Person(this.name, this.gender, this.age);
8
9     walking() => print('$name is walking');
10    talking() => print('$name is talking');
11 }
12
13 int main() {
14     var firstPerson = Person("Sarah", "Female", 25);
15     print(firstPerson.name);
16     print(firstPerson.gender);
17     print(firstPerson.age);
18 }
```

```
1 Output:
2 Sarah
3 Female
4 25
```

Named Constructor

Chúng ta có thể tạo nhiều constructor trong Dart dựa trên các tình huống khác nhau. Trong những trường hợp như vậy, tốt hơn hết là nên đặt tên cho các constructor để rõ ràng hơn.

Syntax như sau:





```
ClassName.constructorName{  
    Constructor Body  
}
```

```
1  class Person{  
2      String name;  
3      String gender;  
4      int age;  
5  
6      // Generative Constructor  
7      Person(this.name, this.gender, this.age);  
8  
9      // Named Constructor  
10     Person.newBorn(){  
11         this.age = 0;  
12     }  
13  
14     walking() => print('$name is walking');  
15     talking() => print('$name is talking');  
16 }  
17  
18 int main() {  
19     var firstPerson = Person("Sarah","Female",25);  
20     var secondPerson = Person.newBorn();  
21     print(secondPerson.age);  
22 }
```



Vietcombank

```
1 | Output:
2 | 0
```

Nếu bạn không khai báo một constructor, một constructor mặc định sẽ được cung cấp cho bạn.

Một constructor mặc định không có tham số. Nó tạo một instance của một class mà không cần khởi tạo các biến instance của nó.

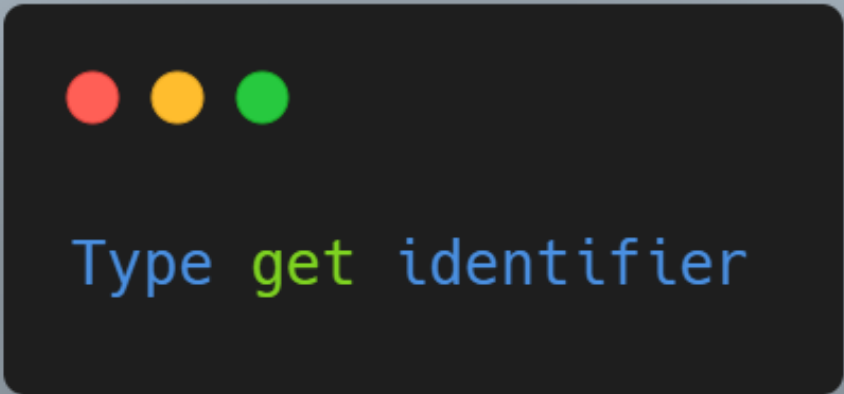
6. Getter and Setters

Getters và setters là các phương thức đặc biệt cung cấp quyền truy cập đọc và ghi vào các thuộc tính của một đối tượng. Bạn có nhớ cách chúng ta có thể truy xuất và đặt giá trị của các biến instance bằng toán tử dấu chấm (`.`) không?. Mỗi biến instance có một getter và setter ngầm mà chúng ta đã sử dụng cho đến bây giờ.

Getter

Getters là các hàm được sử dụng để truy xuất các giá trị thuộc tính của một đối tượng và được định nghĩa bằng cách sử dụng từ khóa `get`.

Syntax như sau:



```
Type get identifier
```

Hãy xem một ví dụ rất đơn giản về `get`

```
1 | class Person{
2 |     String name;
```

```

3   String gender;
4   int age;
5
6   Person(this.name, this.gender, this.age);
7
8   Person.newBorn(){
9       this.age = 0;
10  }
11
12  // Getter function getting the value of name
13  String get personName => name;
14
15  walking() => print('$name is walking');
16  talking() => print('$name is talking');
17  }
18
19  int main() {
20      var firstPerson = Person("Sarah", "Female", 25);
21      print(firstPerson.personName);
22  }

```

```

1 | Output:
2 | Sarah

```

Trên dòng `String get personName => name;`, chúng ta đang tạo một hàm getter trả về giá trị `name` của phiên bản hiện tại. Trên dòng `print(firstPerson.personName);`, chúng ta đang gọi hàm getter và output sẽ hiển thị `Sarah`.

Setters

Setters là các hàm được sử dụng để viết các giá trị thuộc tính của một đối tượng và được định nghĩa bằng cách sử dụng từ khóa `set`

Syntax như sau:



Type set identifier(parameterList)

```
1 class Person{
2     String name;
3     String gender;
4     int age;
5
6     String get personName => name;
7
8     // Setter function for setting the value of age
9     void set personAge(num val){
10         if(val < 0){
11             print("Age cannot be negative");
12         } else {
13             this.age = val;
14         }
15     }
16
17     walking() => print('$name is walking');
18     talking() => print('$name is talking');
19 }
20
21 int main() {
22     var firstPerson = Person();
23     firstPerson.personAge = -5;
24     print(firstPerson.age);
25 }
```

```
1 Output:
2 Age cannot be negative
3 null
```

Từ dòng 9 đến dòng 15, chúng ta đang tạo một hàm setter để đặt giá trị cho `age`. Setter có một điều kiện đảm bảo rằng người dùng không nhập độ tuổi âm.

Ở dòng `firstPerson.personAge = -5;`, chúng ta đang đặt giá trị tuổi của `firstPerson` bằng cách sử dụng hàm setter `personAge`.

Hãy xem một ví dụ phức tạp hơn.

Class `Figure` có 4 thuộc tính là `left`, `top`, `width` và `height`. Chúng ta sẽ tạo hai hàm getter tính toán giá trị của hai thuộc tính mới, `right` và `bottom`.

Chúng ta cũng sẽ tạo một hàm setter cho `right`. Các thuộc tính `right` và `left` phụ thuộc lẫn nhau. Có nghĩa là dựa trên giá trị được đặt cho `right`, giá trị của bên `left` cần được sửa đổi. Điều này đang được xử lý bởi hàm setter.

Tương tự như vậy, `bottom` và `height` phụ thuộc lẫn nhau. Do đó, setter cho `bottom` sửa đổi giá trị của `height` theo giá trị của `bottom`.

```
1 class Figure {
2     num left, top, width, height;
3
4     Figure(this.left, this.top, this.width, this.height);
5
6     // Define two calculated properties: right and bottom.
7     num get right => left + width;
8     set right(num value) => left = value - width;
9     num get bottom => top - height;
10    set bottom(num value) => top = value + height;
11 }
12
13 main() {
14     var fig = Figure(3, 4, 20, 15);
15     print(fig.left);
16     print(fig.right);
17     fig.right = 12;
18     print(fig.left);
19 }
```



```
1 Output:
2 3
3 23
4 -8
```

Trên dòng 15, giá trị của `left` là 3. Tuy nhiên, khi chúng ta gọi hàm setter `right` trên dòng `fig.right = 12;`, giá trị của `left` được sửa đổi thành -8 và được hiển thị trên dòng `print(fig.left);`.

7. Inheritance

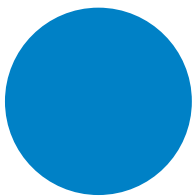
Bây giờ bạn đã được làm quen với các đối tượng và class, chúng ta hãy nói về inheritance, một trong những khái niệm cốt lõi của lập trình hướng đối tượng.

Inheritance là một khái niệm mà qua đó bạn có thể tạo một class mới từ class đã tồn tại. Class mới kế thừa các thuộc tính (biến instance) và các phương thức của lớp hiện có. Class kế thừa được gọi là subclass, trong khi lớp đang được kế thừa được gọi là superclass.

Mỗi class là một subclass của superclass Object. Nó nằm ở đầu của hệ thống phân cấp lớp và không có superclass của chính nó.

Có thể bạn sẽ đặt ra câu hỏi là khi nào chúng ta sử dụng inheritance? Câu trả lời là bất cứ khi nào chúng ta bắt gặp mối quan hệ "IS A" giữa các đối tượng thì chúng ta có thể sử dụng inheritance.

Circle



is a Shape

DART



is a Programming Language

Car



is a Vehicle

Trong hình minh họa trên, chúng ta có thể thấy các đối tượng có mối quan hệ "IS A" giữa chúng:

- Circle is a shape
- DART is a programming language
- Car is a vehicle

Class trước "is a" là subclass và class sau "is a" là superclass

Superclass	Subclass
Shape	Circle
Programming Language	DART
Vehicle	Car

FutureBuilder

FutureBuilder là một Widget trong Flutter nhận vào một function Future và trả ra kết quả nhận được để người dùng render Widget tương ứng lên màn hình. FutureBuilder gồm 2 thành phần chính:

future: là future cần thực thi

builder: là widget hiển thị kết quả lên trên giao diện cho người dùng.

```
1 FutureBuilder(  
2     future: testDelay(),  
3     builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {  
4         return Text(snapshot.data.toString());  
5     },  
6 );
```

Khởi tạo FutureBuilder

```
1 @override  
2 Widget build(BuildContext context) {  
3     return MaterialApp(home: Scaffold( appBar: AppBar(title: const  
4 Text("FutureBuilder")),  
5         body: Center(  
6             child: FutureBuilder(  
7                 future: testDelay(),  
8                 builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {  
9                     return Text(snapshot.data.toString());  
10                },  
11            ),  
12        ),  
13    );  
14 }  
15  
16 Future testDelay() async {  
17     await Future.delayed(const Duration(seconds: 3));  
18     return 1;  
19 }
```

AsyncSnapshot là gì

Khi sử dụng FutureBuilder, có thể sử dụng AsyncSnapshot để kiểm tra các trạng thái của Future:

- ConnectionState.none: khi `future` không được truyền vào
- ConnectionState.waiting: khi `future` đang được thi
- ConnectionState.done: khi `future` hoàn thành. Có thể sử dụng `.hasData`, `.hasError` để kiểm tra dữ liệu trả về có hay không

Ví dụ về AsyncSnapshot

```
1 FutureBuilder(  
2   future: testDelay(),  
3   builder: (BuildContext context, AsyncSnapshot<dynamic> snapshot) {  
4     if (snapshot.connectionState == ConnectionState.active) {  
5       return const Text("Please wait...");  
6     } else {  
7       if (snapshot.hasData) {  
8         return Text("Success: ${snapshot.data}");  
9       } else {  
10        return Text("Error: ${snapshot.data}");  
11      }  
12    }  
13  },  
14 );
```

Tổng kết Future

Future thích hợp sử dụng cho 1 tác vụ bất đồng bộ

Có thể sử dụng FutureBuilder để cập nhật tiến độ, kết quả của 1 future lên giao diện người dùng.