BÀI 2

CƠ BẢN VỀ NGÔN NGỮ DART

GV: ThS. BÙI PHÚ KHUYÊN

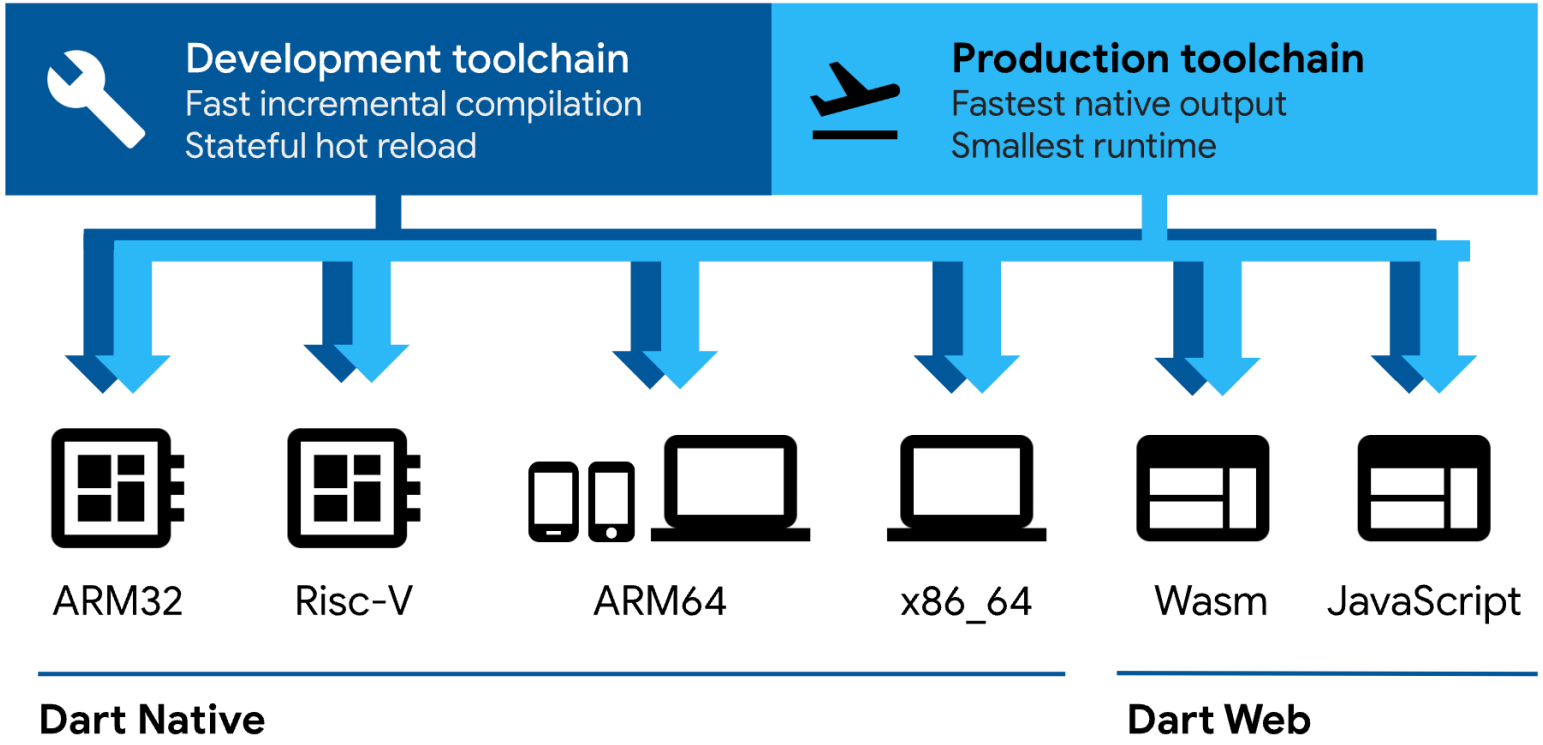
* Cập nhật: 09.09.2024

NỘI DUNG BÀI HỌC

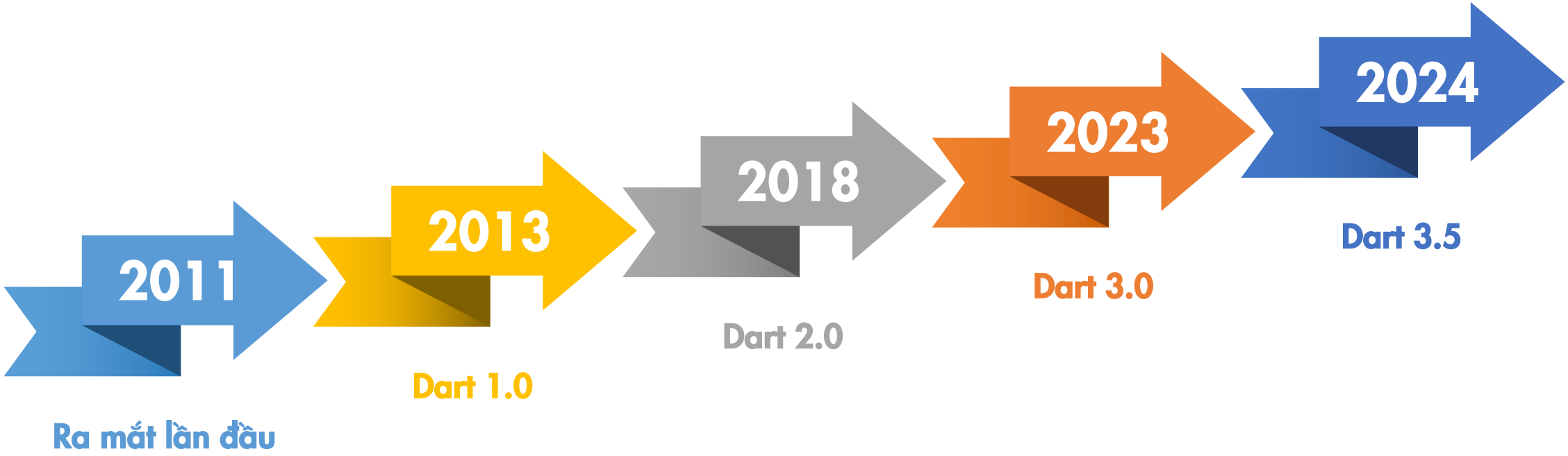
- Giới thiệu ngôn ngữ Dart
- Cài đặt môi trường Dart
- Tìm hiểu cơ bản về Dart
- Hàm
- Lập trình hướng đối tượng
- Xử lý lỗi và đồng thời

GIỚI THIỆU DART

- Dart là một ngôn ngữ lập trình mã nguồn mở, phát triển bởi Google.
- Mục đích: Xây dựng các ứng dụng hiện đại với hiệu suất cao.



LỊCH SỬ PHÁT TRIỂN



https://dart.dev/guides/whats-new

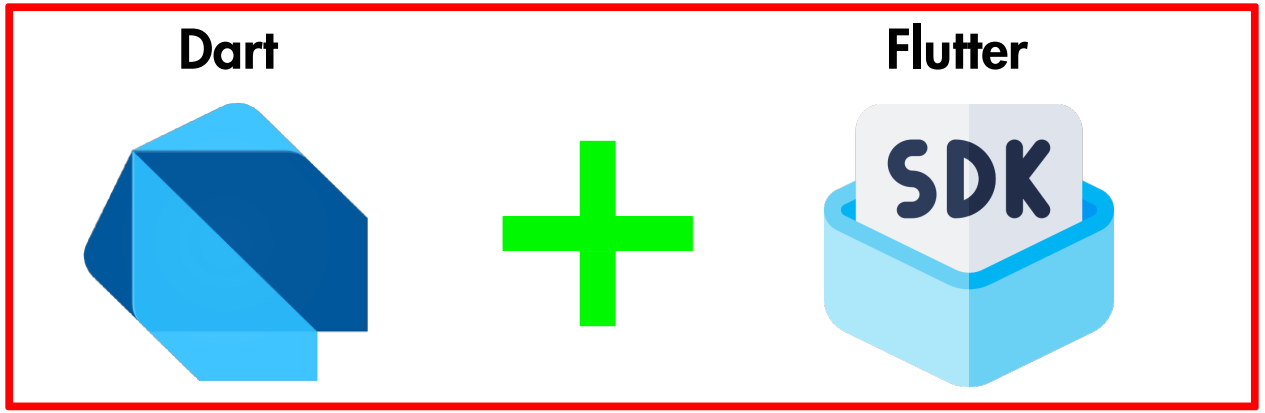
ƯU ĐIỂM CỦA DART {1}

- **Hiệu suất cao**

Mode	Compiler	Compile time	Execution time
Debug	Just-in-time (JIT)	✓ FAST (Good For Dev)	✗ SLOW
Release	Ahead-of-time (AOT)	✗ SLOW	✓ FAST (Good For End-users)

ƯU ĐIỂM CỦA DART {2}

- **Đa nền tảng**



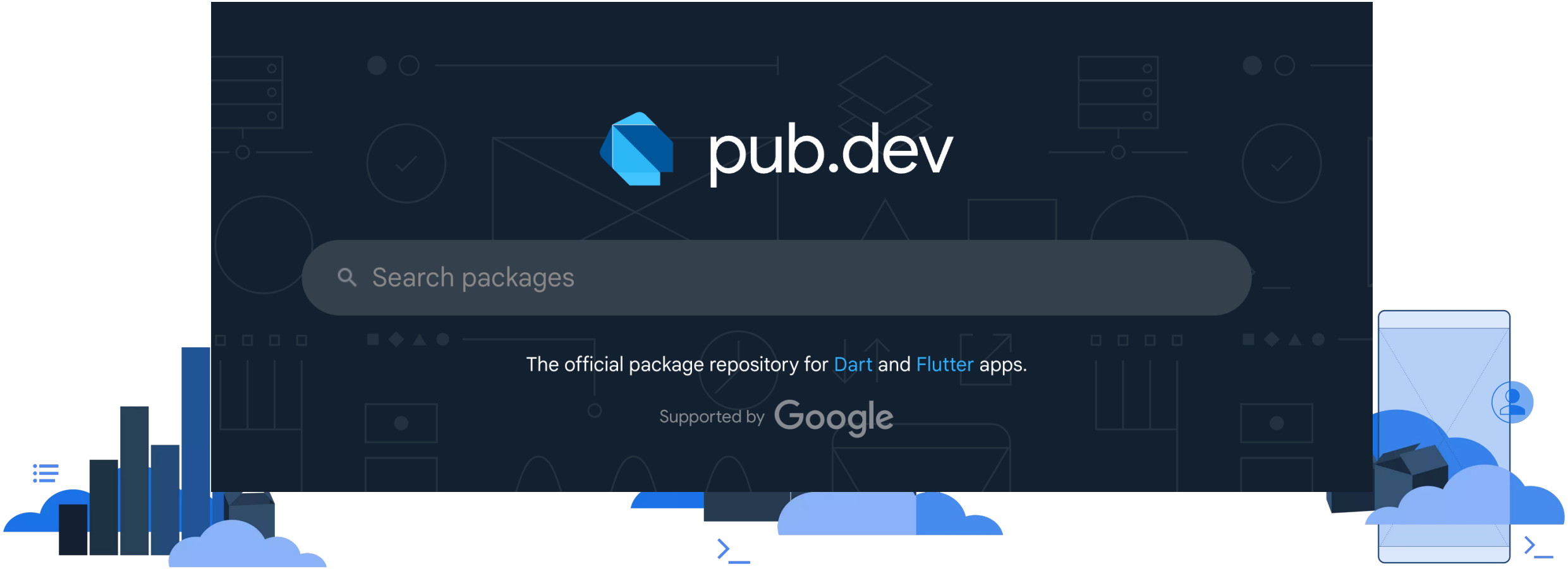
Viết một lần, chạy trên mọi nơi

ƯU ĐIỂM CỦA DART {3}

- Ngôn ngữ dễ học

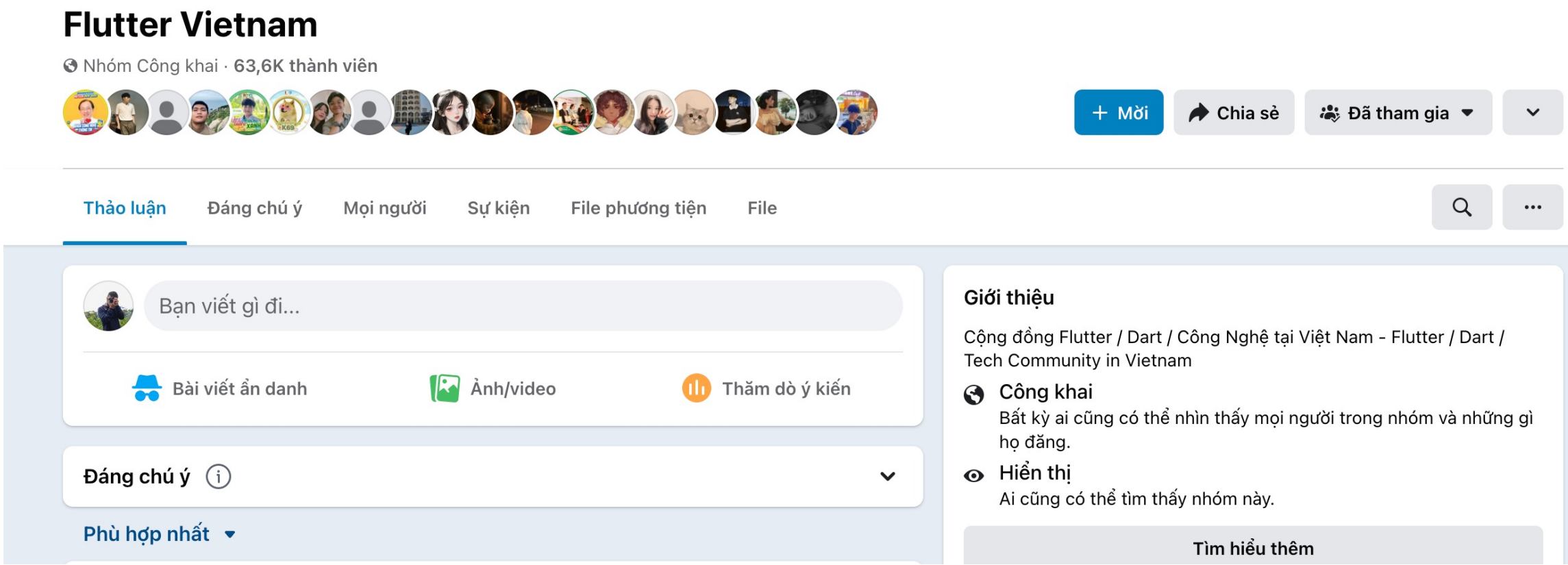
ƯU ĐIỂM CỦA DART {4}

- **Thư viện phong phú**



ƯU ĐIỂM CỦA DART {4}

- **Cộng đồng mạnh mẽ**



THIẾT LẬP MÔI TRƯỜNG

PHÁT TRIỂN DART

DART SDK

System requirements

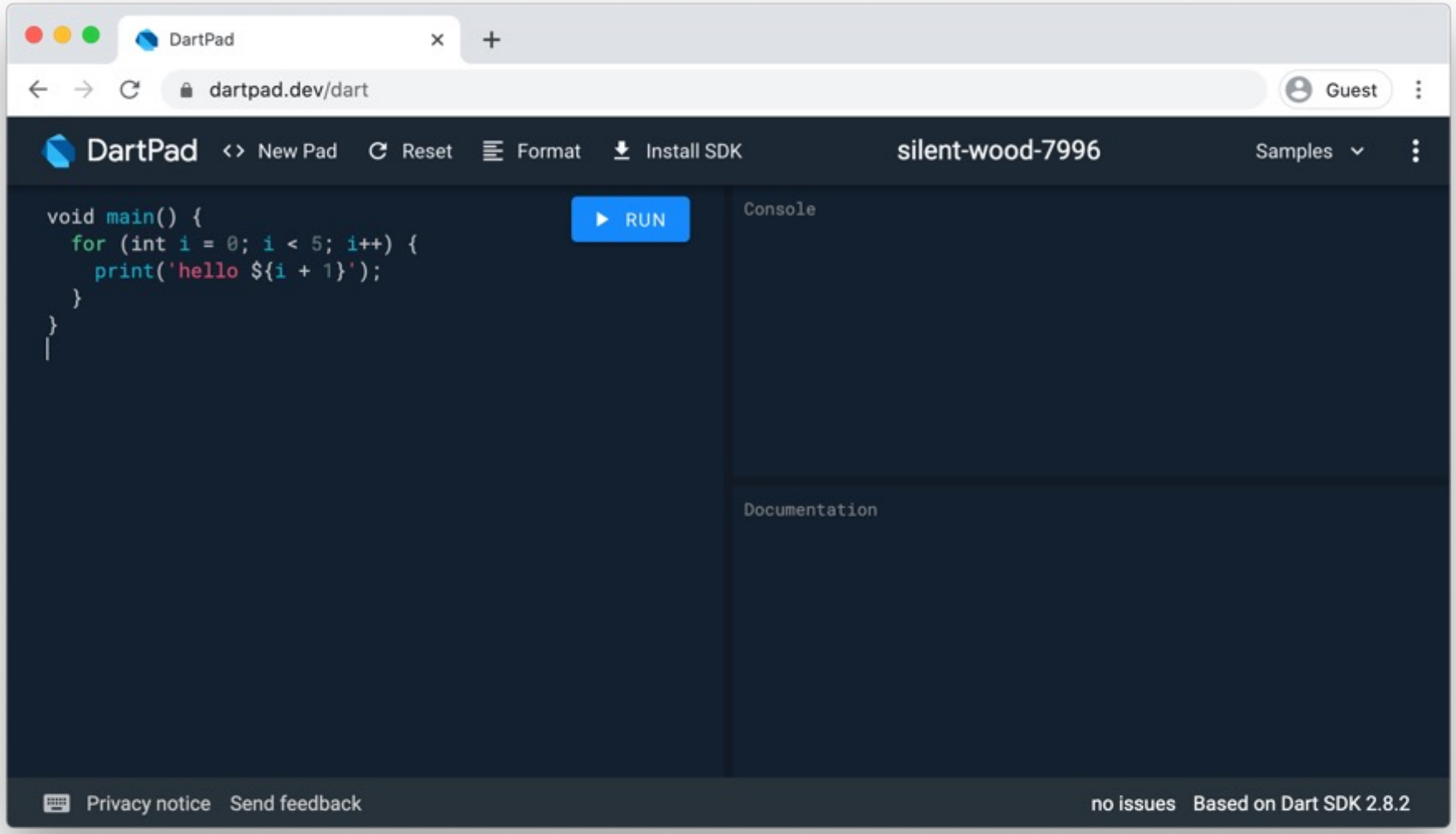
Dart supports the following hardware architectures and platform versions to develop and run Dart code.

Platform	x64	IA32 (x86)	Arm32	Arm64	RISC-V (RV64GC)	OS Versions
Windows	✓	!	✗	✓	–	10, 11
Linux	✓	!	✓	✓	✓	Debian stable, Ubuntu LTS under standard support
macOS	✓	✗	–	✓	–	Latest three versions of macOS: Sonoma (14), Ventura (13), Monterey (12)

✓ Supported on all channels.
! Support is deprecated and might be dropped in a future Dart release.
✗ Unsupported on all channels.
– Unsupported by the operating system.

https://dart.dev/get-dart

DARTPAD



https://dartpad.dev/

NGÔN NGỮ DART

CÚ PHÁP CƠ BẢN

CHƯƠNG TRÌNH DART CƠ BẢN


```dart
import 'dart:io';
// Định nghĩa lớp Person
class Person {
  // Thuộc tính
  String name;
  int age;
  // Phương thức khởi tạo
  Person(this.name, this.age);
  // Phương thức để in thông tin
  void printInfo() {
    print('Name: $name');
    print('Age: $age');
  }
}
// Hàm để tính tổng hai số
int addNumbers(int a, int b) {
  return a + b;
}
```

```dart
// Hàm chính của chương trình
void main() {
  // Nhập tên từ người dùng
  stdout.write('Enter your name: ');
  String name = stdin.readLineSync()!;
  // Nhập tuổi từ người dùng
  stdout.write('Enter your age: ');
  int age = int.parse(stdin.readLineSync()!);
  // Tạo đối tượng Person
  Person person = Person(name, age);
  // Gọi phương thức để in thông tin
  person.printInfo();
  // Sử dụng hàm để tính tổng hai số
  int sum = addNumbers(10, 20);
  print('Sum of 10 and 20 is: $sum');
}
```

BIẾN TRONG DART

Biến trong Dart được sử dụng để lưu trữ các tham chiếu tới các đối tượng.

```
// Biến name lưu trữ tham chiếu tới đối tượng String với giá trị 'Khuyên'.
var name = 'Khuyên';
```

Hoặc khai báo rõ ràng kiểu dữ liệu:

```
// Biến name có kiểu String
String name = 'Khuyên';
```

NULL SAFETY

- Dart thực thi tính năng **Null Safety**, giúp ngăn chặn lỗi **Null Dereference**.
- Với **Null Safety**, Dart phát hiện các lỗi tiềm ẩn tại thời gian biên dịch.
- Để khai báo biến có thể nhận giá trị **null**, thêm ký hiệu **?** vào sau kiểu dữ liệu:

```
String? name; // Kiểu dữ liệu có thể null
String name = 'Khuyên'; // Kiểu dữ liệu không thể null (phải được khởi tạo giá trị trước khi sử dụng)
```

BIẾN KIỂU LATE

- Khi ta chắc chắn biến sẽ được khởi tạo trước khi sử dụng nhưng Dart không thể xác định điều đó, ta có thể sử dụng từ khóa **late**

```
late String description;


void main() {
  description = 'Trường HUTECH!';
  print(description); // In ra: Trường HUTECH!
}
```

BIẾN FINAL VÀ CONST

- Biến **final** chỉ được gán một lần và không thể thay đổi giá trị sau khi gán:

```
final name = 'Khuyên';      // Biến final
name = 'Cường';             // Lỗi: không thể thay đổi giá trị của biến final
```

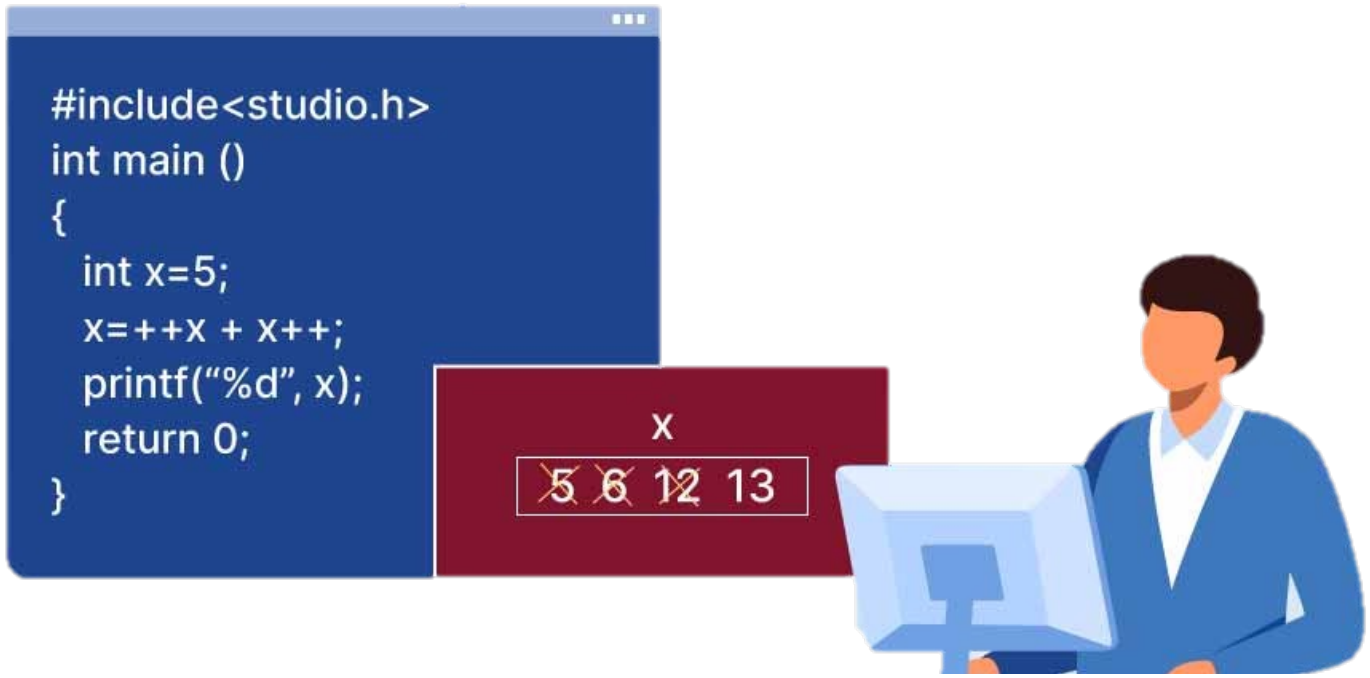
- Biến **const** là hằng số tại thời gian biên dịch và không thể thay đổi giá trị:

```
const bar = 1000000;                  // Đơn vị áp suất (dynes/cm2)
const double atm = 1.01325 * bar;     // Áp suất tiêu chuẩn
```

TOÁN TỬ - OPERATORS {1}

- Dart hỗ trợ một loạt các toán tử, từ **toán tử cơ bản** như cộng trừ nhân chia đến các **toán tử logic** và **toán tử thao tác bit**.

- Các toán tử này được sử dụng để thực hiện các phép tính và thao tác trên các giá trị và biến trong chương trình.



TOÁN TỬ SỐ HỌC

Toán tử	Ý nghĩa
+	Cộng
-	Trừ
*	Nhân
/	Chia
~/	Chia (lấy phần nguyên)
%	Chia (lấy phần dư)
-expr	Phủ định (đổi dấu biểu thức)

```
assert(2 + 3 == 5);

assert(2 - 3 == -1);

assert(2 * 3 == 6);

assert(5 / 2 == 2.5); // Kết quả là double

assert(5 ~/ 2 == 2);  // Kết quả là int

assert(5 % 2 == 1);   // Phần dư là 1
```

*Hàm **assert()** sử dụng để kiểm tra các điều kiện khi Dev

TOÁN TỬ TĂNG GIẢM TIỀN TỐ/HẬU TỐ

Toán tử	Ý nghĩa
++var	var = var + 1 (giá trị của biểu thức là var+1)
var++	var = var + 1 (giá trị của biểu thức là var)
--var	var = var - 1 (giá trị của biểu thức là var-1)
var--	var = var - 1 (giá trị của biểu thức là var)

```
int a; int b;
a = 0;
b = ++a; // Tăng a trước khi b nhận giá trị của nó.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Tăng a sau khi b nhận giá trị của nó.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Giảm a trước khi b nhận giá trị của nó.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Giảm a sau khi b nhận giá trị của nó.
assert(a != b); // -1 != 0
```

TOÁN TỬ SO SÁNH VÀ QUAN HỆ

Toán tử	Ý nghĩa
==	Bằng
!=	Không bằng
>	Lớn hơn
<	Nhỏ hơn
>=	Lớn hơn hoặc bằng
<=	Nhỏ hơn hoặc bằng

```
assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);
```

TOÁN TỬ KIỂM TRA KIỂU

Toán tử	Ý nghĩa
as	Ép kiểu
is	Kiểm tra kiểu
!is	Kiểm tra không phải kiểu

```
if (employee is Person) {
  // Kiểm tra kiểu
  employee.firstName = 'Khuyên';
}
// Ép kiểu
(employee as Person).firstName = 'Khuyên';
```

TOÁN TỬ GÁN

Toán tử	Ý nghĩa
=	Gán giá trị
??=	Gán giá trị nếu biến hiện tại là **null**
+=, -=,...	Gán kết hợp với các toán tử khác

```
int value = 10;
int? b;                 // b ban đầu là null
b ??= value;            // b sẽ có giá trị là 10

int? c = 5;             // c không phải là null
c ??= value;            // c vẫn giữ giá trị là 5

var a = 2;              // Gán giá trị
a *= 3;                 // Gán và nhân: a = a * 3
assert(a == 6);
```

TOÁN TỬ LOGIC

Toán tử	Ý nghĩa
`!expr`	Đảo ngược biểu thức (biến `false` thành `true` và ngược lại)
`||`	**OR** logic
`&&`	**AND** logic

```
if (!done && (col == 0 || col == 3))
{
  // Thực hiện hành động nào đó...
}
```

TOÁN TỬ ĐIỀU KIỆN

Toán tử	Ý nghĩa
`expr1 ? expr2 : expr3`	Nếu **expr1** đúng, trả về **expr2**; nếu không, trả về **expr3**
`expr1 ?? expr2`	Nếu **expr1** không **null**, trả về giá trị của nó; nếu không, trả về **expr2**

```
var visibility = isPublic ? 'public' : 'private';

String playerName(String? name) => name ?? 'Guest';
```

TOÁN TỬ CASCADE {1}

- Toán tử cascade (..,?..) cho phép thực hiện một loạt các thao tác trên cùng 1 đối tượng.

```
var paint = Paint();
paint.color = Colors.black;
paint.strokeCap = StrokeCap.round;
paint.strokeWidth = 5.0;
```

→

```
var paint = Paint()
  ..color = Colors.black
  ..strokeCap = StrokeCap.round
  ..strokeWidth = 5.0;
```

TOÁN TỬ CASCADE {2}

- Nếu đối tượng mà toán tử cascade thao tác có thể null, hãy sử dụng cascade có kiểm tra null **(?..)** cho thao tác đầu tiên

```
var button = querySelector('#confirm');
button?.text = 'Xác nhận';
button?.classes.add('important');
button?.onClick.listen((e) =>
      window.alert('Đã xác nhận!'));
button?.scrollIntoView();
```

→

```
querySelector('#confirm')
  ?..text = 'Xác nhận'
  ..classes.add('important')
  ..onClick.listen((e) =>
          window.alert('Đã xác nhận!'))
  ..scrollIntoView();
```

CÁC TOÁN TỬ KHÁC

Toán tử	Ý nghĩa
`()`	Biểu thị một lệnh gọi hàm/phương thức
`[] hoặc ?[]`	Truy cập phần tử mảng. Ví dụ: `fooList[1]`
`. hoặc ?.`	Tham chiếu đến một thuộc tính của biểu thức/lớp. Ví dụ: `foo.bar`
`!`	Ép kiểu một biểu thức thành **`không-null`**, ném một ngoại lệ tại **`runtime`** nếu ép kiểu thất bại. Ví dụ: `foo!.bar` xác nhận `foo` **`không-null`** và truy cập thuộc tính **`bar`**

CHÚ THÍCH - COMMENTS

Dart hỗ trợ các loại chú thích một dòng, nhiều dòng và chú thích tài liệu.

```dart
// Kiểm tra xem giá trị có phải là số dương
bool isPositive(int number) {
  return number > 0;
}

/// Hàm tính tổng của hai số.
///
/// Trả về tổng của [a] và [b].
int sum(int a, int b) {
  return a + b;
}
```

```dart
/*
  Hàm này sẽ tính toán và trả về tổng của hai số.
  Đây là ví dụ về cách sử dụng comment nhiều dòng.
*/
int add(int a, int b) {
  return a + b;
}
```

THƯ VIỆN - LIBRARIES

- Sử dụng thư viện

Để sử dụng một thư viện trong Dart, ta sử dụng từ khóa **import**. Việc này cho phép ta truy cập các lớp, hàm và các thành phần khác từ thư viện đó trong mã.

```
import 'dart:html';

import 'package:test/test.dart';
```

- Đặt tiền tố cho thư viện

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// Sử dụng Element từ lib1.
Element element1 = Element();

// Sử dụng Element từ lib2.
lib2.Element element2 = lib2.Element();
```

TỪ KHOÁ - KEYWORDS

abstract [2]	as [2]	assert	async [3]
await [1]	base [3]	break	case
catch	class	const	continue
covariant [2]	default	deferred [2]	do
dynamic [2]	else	enum	export [2]
extends	extension [2]	external [2]	factory [2]
false	final (var)	final (class)	finally
for	Function [2]	get [2]	hide [3]
if	implements [2]	import [2]	in
interface [2]	is	late [2]	library [2]
mixin [2]	new	null	of [3]
on [3]	operator [2]	part [2]	required [2]
rethrow	return	sealed [3]	set [2]
show [3]	static [2]	super	switch
sync [3]	this	throw	true
try	type [2]	typedef [2]	var
void	when [3]	with	while
yield [1]			

Bảng thể hiện một số từ khoá trong Dart

NGÔN NGỮ DART

KIỂU DỮ LIỆU - DATATYPE

KIỂU SỐ (NUMBERS) {1}

- Dart có 2 loại kiểu số chính: số nguyên (**int**) và số thực (**double**).

```
var x = 10;          //int
var y = 1.1;         //double
```

- Ta cũng có thể khai báo biến dưới dạng num. Khi đó, biến có thể chứa cả giá trị nguyên và giá trị thực.

```
num x = 1;           // x có thể chứa cả giá trị int và double
x += 2.5;
```

KIỂU SỐ (NUMBERS) {2}

- Một số ví dụ về cách chuyển đổi giữa chuỗi và số

```
// Chuỗi -> số nguyên
var one = int.parse('1');
assert(one == 1);

// Chuỗi -> số thực
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);
```

```
// Số nguyên -> chuỗi
String oneAsString = 1.toString();
assert(oneAsString == '1');

// Số thực -> chuỗi
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

KIỂU CHUỖI {1}

- Chuỗi trong Dart là một tập hợp các ký tự **UTF-16**. Ta có thể sử dụng dấu nháy đơn hoặc dấu nháy kép để tạo chuỗi

```
var s1 = 'Dấu nháy đơn cho chuỗi.';
var s2 = "Dấu nháy kép cũng được sử dụng.";
```

- Ta có thể chèn giá trị của một biểu thức vào trong chuỗi bằng cách sử dụng cú pháp **${expression}.** Nếu biểu thức chỉ là một biến đơn, ta có thể bỏ qua dấu ngoặc nhọn.

```
var name = 'Khuyên';
print(Xin chào, $name!');
```

KIỂU CHUỖI {2}

- Ở một số trường hợp, ta có thể tạo chuỗi với dữ liệu thô (tức không có bất kỳ ký tự đặc biệt nào được xử lý trong chuỗi) này, hãy thêm ký tự r trước chuỗi.

```
var s = r'Đây là một chuỗi thô, ngay cả \n cũng không được xử lý.';
```

- Ta có thể nối các chuỗi bằng cách sử dụng các chuỗi liên tiếp hoặc sử dụng toán tử +

```
var s1 = 'Chuỗi '
    'liên tiếp'
    " hoạt động qua cả dòng mới.";
var s2 = 'Toán tử + ' + 'cũng hoạt động.';
```

KIỂU LOGIC (BOOLEAN)

- Kiểu dữ liệu **boolean** chỉ có hai giá trị: **true** và **false**

```
bool isTrue = true;

bool isFalse = false;
```

- Ta có thể sử dụng **boolean** trong các câu lệnh điều kiện:

```
if (isTrue) {
  print('Đúng!');
} else {
  print('Sai!');
}
```

KIỂU DANH SÁCH (LIST)

- Danh sách trong Dart được biểu thị bằng các giá trị hoặc biểu thức được phân tách bằng dấu phẩy, nằm trong dấu ngoặc vuông ([ ])

```
var list = [1, 2, 3];      //Hoặc List<int> list = [1, 2, 3]
```

- Danh sách sử dụng chỉ số bắt đầu từ **0**. Ta có thể lấy độ dài của danh sách bằng thuộc tính **.length** và truy cập các giá trị của danh sách bằng toán tử chỉ số ([ ]):

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

```
list[1] = 1;
assert(list[1] == 1);
```

CẤU TRÚC SET

Cấu trúc **Set** là một tập hợp các phần tử không trùng lặp trong danh sách

```
var set = {1, 2, 3};          // Hoặc Set<int> set = {1,2,3}
set.add(4);
set.add(1);                   // Không thêm phần tử trùng lặp
print(set);                   // {1, 2, 3, 4}
```

CẤU TRÚC MAP

Map là một tập hợp các cặp **khóa-giá trị** (**key-value**), cho phép lưu trữ và truy xuất giá trị theo khóa.

```
var map = {'name': 'Khuyên', 'age': 26};  //Hoặc Map<String,dynamic> map = {…}
map['language'] = 'Tiếng Việt';
print(map); // {'name': 'Khuyên', 'age': 26, 'language': 'Tiếng Việt'}
```

KIỂU RUNE (CHUỖI UNICODE 32BIT)

Trong Dart, rune là các điểm mã UTF-32 của một chuỗi. Unicode xác định một giá trị số duy nhất cho mỗi chữ cái, chữ số và ký hiệu được sử dụng trong tất cả các hệ thống chữ viết của thế giới.

```
Runes input = Runes( '\u2665 \u{1f605} \u{1f60e} \u{1f47b}');
print(String.fromCharCodes(input));
```

♥ 😅 😎 👻

GENERICS {1}

Generics trong Dart là một cách để tạo ra các lớp, hàm và cấu trúc dữ liệu có thể làm việc với nhiều loại dữ liệu khác nhau. Các lợi ích của **Generics** bao gồm:

- Kiểm tra lỗi trong quá trình viết mã: Khi sử dụng **Generics**, Dart có thể kiểm tra lỗi ngay trong quá trình viết mã nguồn, giúp phát hiện lỗi sớm hơn.
- Giảm trùng lặp mã: Ta không cần viết lại mã nguồn cho mỗi loại dữ liệu khác nhau. Thay vào đó, ta chỉ cần viết một lần và sử dụng cho nhiều loại dữ liệu.
- Cải thiện hiệu suất: Khi sử dụng **Generics**, mã nguồn được tối ưu hóa tốt hơn, giúp cải thiện hiệu suất của chương trình.

- Ta có thể khai báo **Generics** cho một lớp bằng cách sử dụng cú pháp **<T>** ngay sau tên lớp

```
class Box<T> {
  T? value;

  Box(this.value);

  T getValue() => value!;
  void setValue(T value) => this.value = value;
}
```

```
void main() {
  var intBox = Box<int>(123);
  print(intBox.getValue()); // Output: 123

  var stringBox = Box<String>("Hello");
  print(stringBox.getValue()); // Output: Hello
}
```

GENERICS {3}

- Khi sử dụng **Generics**, ta có thể chỉ định rõ loại dữ liệu mà một **Collection** chứa. Điều này giúp kiểm tra lỗi ngay trong quá trình biên dịch, tránh việc thêm các loại dữ liệu không phù hợp vào **Collection**

```
void main() {
  var names = <String>[];
  names.addAll(['Seth', 'Kathy', 'Lars']);
  names.add(42); // Lỗi: Không thể thêm số vào danh sách Generic chuỗi
}
```

TYPEDEF

- **Typedefs** trong Dart là một cách ngắn gọn để tạo các **alias** (bí danh) cho các loại dữ liệu.
- Giúp mã nguồn trở nên rõ ràng và dễ đọc hơn, đặc biệt khi làm với các loại dữ liệu phức tạp.

```
typedef IntList = List<int>;

void main() {
  IntList il = [1, 2, 3];
  print(il); // Output: [1, 2, 3]
}
```

NGÔN NGỮ DART

LOOPS – BRANCHES

CẤU TRÚC LẶP {1}

FOR

Vòng lặp **for** truyền thống

```
// Ví dụ: In ra từ 1 đến 5
for (var i = 1; i <= 5; i++) {
  print(i);
}
// Kết quả: 1 2 3 4 5
```

Vòng lặp **for-in**

```
// Ví dụ: In ra các tên trong danh sách
var names = ['An', 'Bình', 'Chi'];
for (var name in names) {
  print(name);
}
// Kết quả: An Bình Chi
```

WHILE / DO-WHILE

Vòng lặp **while**

```
// Ví dụ: In ra từ 1 đến 5
var i = 1;
while (i <= 5) {
  print(i);
  i++;
}
// Kết quả: 1 2 3 4 5
```

Vòng lặp **do-while**

```
// Ví dụ: In ra từ 1 đến 5
var i = 1;
do {
  print(i);
  i++;
} while (i <= 5);
// Kết quả: 1 2 3 4 5
```

BREAK / CONTINUE

```
//Thoát khỏi vòng lặp khi giá trị bằng 3
for (var i = 1; i <= 5; i++) {
  if (i == 3) {
    break;
  }
  print(i);
}
// Kết quả: 1 2
```

```
//Bỏ qua giá trị 3 và tiếp tục vòng lặp
for (var i = 1; i <= 5; i++) {
  if (i == 3) {
    continue;
  }
  print(i);
}
// Kết quả: 1 2 4 5
```

CẤU TRÚC RẼ NHÁNH {1}

IF / IF-ELSE

```
void main() {
  bool isRaining = false;
  bool isSnowing = true;

  if (isRaining) {
    print('Mang theo áo mưa.');
  } else if (isSnowing) {
    print('Mặc áo khoác.');
  } else {
    print('Hạ mui xe xuống.');
  }
}
```

```
void main() {
  var pair = [3, 4];

  if (pair case [int x, int y]) {
    print('Cặp tọa độ là: $x, $y');
  } else {
    print('Dữ liệu không hợp lệ.');
  }
}
// Kết quả: Cặp toạ độ là 3, 4
```

SWITCH - CASE

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    print('Đã đóng');
    break;
  case 'PENDING':
    print('Đang chờ xử lý.');
    break;
  case 'APPROVED':
    print('Đã phê duyệt.');
    break;
  case 'DENIED':
    print('Bị từ chối.');
    break;
  case 'OPEN':
    print('Mở');
    break;
  default:
    print('Lệnh không xác định.');
}
```

BÀI TẬP CÁ NHÂN 1

TRẮC NGHIỆM 15 PHÚT

NGÔN NGỮ DART

HÀM – FUNCTION

HÀM MAIN

- Mọi ứng dụng Dart đều phải có hàm **`main()`** là điểm khởi đầu của ứng dụng
- Hàm **`main()`** trả về **`void`** và có thể có tham số tùy chọn **`List<String>`** để nhận các đối số dòng lệnh

```
void main() {
  print('Hello, World!');
}

void main(List<String> arguments) {
  print(arguments); // In ra các đối số dòng lệnh trong danh sách
  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

HÀM CƠ BẢN/RÚT GỌN

Đối với các hàm chỉ chứa duy nhất một biểu thức, ta có thể sử dụng cú pháp rút gọn với =>

```
double tinhtong(var a, var b) {
    return a + b;
}

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

1 Biểu thức

HÀM VỚI TÊN THAM SỐ {1}

- Khi gọi 1 hàm, ta có truyền giá trị thông qua tên biến **paramName: value**, việc này giúp việc đọc code dễ dàng hơn trong các trường hợp hàm có quá nhiều tham số (**parameter**).
- Hàm phải khai báo tham số ở trong **{ }**.

```
String createFullName({String surName, String midName, String firstName}) {
  return surName + midName + firstName;
}

void main() {
  var result = createFullName(surName: 'Bui ', midName: 'Phu ', firstName: 'Khuyen ');
  print(result);  // Output: Bui Phu Khuyen
}
```

HÀM VỚI TÊN THAM SỐ {2}

- Để đặt giá trị mặc định cho 1 tham số, ta có thể sử dụng cú pháp **paramName=defaultValue**. Điều này giúp tránh các lỗi do tham số không được cung cấp giá trị khi gọi hàm

```
String createFullName({String surName = 'Bui ', String midName = 'Phu ', String firstName = 'Khuyen '}) {
  return surName + midName + firstName;
}
void main() {
  var result = createFullName();
  print(result);  // Output: Bui Phu Khuyen
  var result2 = createFullName(surName: 'Nguyen ');
  print(result2);  // Output: Nguyen Phu Khuyen
}
```

HÀM VỚI TÊN THAM SỐ {2}

- Để đặt giá trị mặc định cho 1 tham số, ta có thể sử dụng cú pháp **paramName=defaultValue**. Điều này giúp tránh các lỗi do tham số không được cung cấp giá trị khi gọi hàm

```
String createFullName({String surName = 'Bui ', String midName = 'Phu ', String firstName = 'Khuyen '}) {
  return surName + midName + firstName;
}

void main() {
  var result = createFullName();
  print(result);  // Output: Bui Phu Khuyen
  var result2 = createFullName(surName: 'Nguyen ');
  print(result2);  // Output: Nguyen Phu Khuyen
}
```

HÀM VỚI TÊN THAM SỐ {3}

- Để bắt buộc người gọi phải cung cấp giá trị cho tham số, ta có thể sử dụng từ khóa **required**. Điều này đảm bảo rằng hàm sẽ nhận được các giá trị cần thiết khi được gọi.

```
String createFullName({required String surName, required String midName, required String firstName}) {
  return surName + midName + firstName;
}
void main() {
  var result = createFullName(surName: 'Bui ', midName: 'Phu ', firstName: 'Khuyen ');
  print(result);  // Output: Bui Phu Khuyen

  // Gọi hàm mà không cung cấp đủ các giá trị bắt buộc sẽ gây lỗi biên dịch
  var result2 = createFullName(surName: 'Nguyen ');
}
```

HÀM VỚI THAM SỐ TUỲ CHỌN

- Các tham số tùy chọn trong hàm cho phép ta gọi hàm mà không cần cung cấp đầy đủ tất cả các tham số.
- Khi các tham số tùy chọn không được cung cấp, chúng sẽ nhận giá trị **null** theo mặc định.
- Để khai báo các tham số tùy chọn, ta sử dụng dấu ngoặc vuông **[]**.

```
double sum(double a, [double? b, double? c]) {
  var result = a;
  if (b != null) {
    result += b;
  }
  result += (c != null) ? c : 0;
  return result;
}
```

```
void main() {
  print(sum(1));       // Output: 1.0
  print(sum(1, 2));    // Output: 3.0
  print(sum(1, 2, 3)); // Output: 6.0
}
```

HÀM ẨN DANH

- Trong Dart, ta có thể tạo các hàm không có tên gọi là hàm ẩn danh **lambda** hoặc **closure**
- Chúng thường được sử dụng trong các ngữ cảnh mà ta không cần sử dụng tên hàm, ví dụ như khi truyền hàm như tham số cho hàm khác
- Để tạo hàm ẩn danh, ta có thể sử dụng cú pháp tương tự như khi khai báo hàm có tên, nhưng bỏ qua phần kiểu trả về và tên hàm

```
(var a, var b) {
  return a + b;
};
```

⟶

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

NGÔN NGỮ DART

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

KIẾN THỨC CƠ BẢN {1}

- Dart là một ngôn ngữ lập trình hướng đối tượng, nơi mọi thứ đều là đối tượng (object) và các đối tượng này được tạo ra từ các lớp (class).
- Mỗi đối tượng trong Dart có thể có các thuộc tính (biến) và phương thức (hàm).

```dart
class Point {
  double x, y;    //Thuộc tính
  Point(this.x, this.y); //Phương thức khởi tạo
  double distanceTo(Point other) {  //Phương thức
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

```dart
void main() {
  //Khởi tạo đối tượng Point
  var p = Point(2, 2);

  // Gọi phương thức distanceTo() trên p.
  double distance = p.distanceTo(Point(4, 4));
  print(distance); // In ra khoảng cách
}
```

KIẾN THỨC CƠ BẢN {2}

- Trong Dart, thuộc tính và phương thức tĩnh (**static**) được sử dụng để chia sẻ dữ liệu và chức năng giữa tất cả các đối tượng của một lớp
- Ta không cần phải tạo ra một đối tượng để truy cập vào các thuộc tính hoặc phương thức này

```
class Queue {
  static const int initialCapacity = 16; // Thuộc tính tĩnh
}
void main() {
  // Truy cập thuộc tính tĩnh mà không cần tạo đối tượng
  print(Queue.initialCapacity); // Kết quả: 16
}
```

KIẾN THỨC CƠ BẢN {3}

- Trong Dart, **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

```
class Rectangle {
  double left, top, width, height;
  Rectangle(this.left, this.top, this.width, this.height);
  // Định nghĩa hai thuộc tính tính toán: right và bottom.
  double get right => left + width; // Getter cho right
  set right(double value) => left = value - width; // Setter cho right
  double get bottom => top + height; // Getter cho bottom
  set bottom(double value) => top = value - height; // Setter cho bottom
}
```

TÍNH KẾ THỪA

Trong Dart, ta có thể sử dụng từ khóa **extends** để tạo một lớp con (**subclass**) kế thừa từ một lớp cha (**superclass**) và dùng từ khoá **super** để tham chiếu đến lớp cha

```dart
class Product {
  double price;
  int quantity;
  String name;
  Product(this.price, {this.quantity = 0});
  void showTotal() {
    print('Total Price is: ${price * quantity}');
  }
}
```

```dart
class Tablet extends Product {
  double width = 0; double height = 0;
  Tablet(double price) : super(price, quantity: 1) {
    this.name = "IPad Pro"; // Truy vấn thuộc tính từ lớp cha
  }
  @override
  void showTotal() {
    print('Name Tablet is: $name');
    super.showTotal(); // Gọi đến phương thức ở lớp cha
  }
}
```

TÍNH TRỪU TƯỢNG – ABSTRACT

- Lớp trừu tượng là lớp không thể được tạo ra đối tượng trực tiếp.
- Nó chỉ được sử dụng để làm cơ sở cho các lớp con kế thừa.
- Một phương thức trong lớp trừu tượng có thể chỉ khai báo tên mà không có nội dung, được gọi là phương thức trừu tượng.
- Các lớp kế thừa bắt buộc phải định nghĩa lại nội dung cho các phương thức trừu tượng này.

```
abstract class A {
  // Khai báo các thuộc tính
  var name = 'My Abstract Class';

  // Khai báo phương thức trừu tượng
  void displayInformation();
}
```

```
class B extends A {
  @override
  void displayInformation() {
    print(this.name);
  }
}
```

```
void main() {
  var i = B();
  i.displayInformation();
  // Kết quả: My Abstract Class
}
```

TÍNH TRỪU TƯỢNG – INTERFACE

- Trong Dart, mọi lớp đều là một **interface** mặc định
- Khi một lớp triển khai một **interface**, lớp đó phải định nghĩa lại tất cả các phương thức và thuộc tính của **interface** đó
- Để triển khai một **interface**, sử dụng từ khóa **implements**.

```dart
class B {
  String name = 'Class B';

  void displayInformation() {
    print('Information from B');
  }
}
```

```dart
class C implements B {
  @override
  String name;

  @override
  void displayInformation() {
    print('Information from C');
  }
}
```

```dart
void main() {
  var c = C();
  c.displayInformation();
 // Kết quả: Information from C
}
```

TÍNH TRỪU TƯỢNG – MIXIN

- **Mixin** là một lớp có chứa các phương thức và thuộc tính dùng để gộp vào một lớp khác
- **Mixin** không thể được sử dụng để tạo đối tượng trực tiếp.
- Để sử dụng **mixin**, sử dụng từ khóa **with**.

```
mixin M {
  void showSomething() {
    print('Print message ...');
  }
}

class B {
  String name = 'Class B';

  void displayInformation() {
    print('Information from B');
  }
}
```

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

  @override
  void displayInformation() {
    showSomething();
  }
}
```

```
void main() {
  var c = C();
  c.displayInformation();
  // Kết quả: Print message ...
}
```

NGÔN NGỮ DART

XỬ LÝ LỖI

XỬ LÝ LỖI

- Trong Dart, ta có thể ném (**throw**) và bắt (**catch**) các ngoại lệ (**exceptions**).
- Dart cung cấp các loại **Exception** và **Error**, cũng như nhiều loại con được định nghĩa sẵn

```
void main() {
  try {
    throw Exception('Lỗi xảy ra!');
  } catch (e) {
    print('Đã bắt lỗi: $e');
  } finally {
    print('Luôn chạy khối này.');
  }
}
```

NGÔN NGỮ DART

XỬ LÝ ĐỒNG THỜI

GIỚI THIỆU TỔNG QUAN {1}

- Trong lập trình, có 2 khái niệm là đồng bộ (**synchronous**) và bất đồng bộ (**asynchronous**).
- **Lập trình Đồng bộ**: Các tác vụ được thực hiện tuần tự, tác vụ tiếp theo chỉ được thực hiện khi tác vụ hiện tại hoàn thành. Điều này có thể gây ra tình trạng "chờ đợi" khi một tác vụ mất nhiều thời gian để hoàn thành

```
import 'dart:io';

void main() {
  print('Đang đọc file 1...');
  String content1 = File('file1.txt').readAsStringSync();
  print('Nội dung file 1: $content1');

  print('Đang đọc file 2...');
  String content2 = File('file2.txt').readAsStringSync();
  print('Nội dung file 2: $content2');
}
```

GIỚI THIỆU TỔNG QUAN {2}

- **Lập trình Bất đồng bộ:** Cho phép thực hiện nhiều tác vụ cùng một lúc mà không phải chờ đợi tác vụ trước hoàn thành. Điều này làm tăng hiệu suất và khả năng phản hồi của chương trình.

```dart
import 'dart:io';

void main() async {
  print('Đang đọc file 1...');
  Future<String> content1 = File('file1.txt').readAsString();

  print('Đang đọc file 2...');
  Future<String> content2 = File('file2.txt').readAsString();

  print('Nội dung file 1: ${await content1}');
  print('Nội dung file 2: ${await content2}');
}
```

FUTURE & STREAM {1}

- **Future** đại diện cho một giá trị hoặc lỗi sẽ có sẵn trong tương lai.
- Thường được sử dụng cho các hoạt động không đồng bộ như gọi API, đọc/ghi tệp hoặc bất kỳ hoạt động nào cần thời gian để hoàn thành.

```dart
import 'dart:async';
void main() {
  print('Bắt đầu tải dữ liệu người dùng...');
  getUserData().then((data) {
    print('Dữ liệu người dùng: $data');
  }).catchError((error) {
    print('Lỗi khi tải dữ liệu: $error');
  });
}
```

```dart
Future<String> getUserData() async {
  // Giả sử quá trình tải dữ liệu mất 2 giây
  await Future.delayed(Duration(seconds: 2));
  return 'Tên: Phú Khuyên, Tuổi: 26';
}
```

FUTURE & STREAM {2}

- **Stream** đại diện cho một chuỗi các giá trị không đồng bộ hoặc một luồng dữ liệu liên tục.
- Ví dụ điển hình là xử lý các sự kiện người dùng (như click chuột, nhập liệu), đọc dữ liệu liên tục từ một kết nối mạng, hoặc các cập nhật từ cảm biến.

```dart
import 'dart:async';

void main() {
  print('Bắt đầu nhận tin nhắn mới...');
  Stream<String> messageStream = getMessageStream();

  messageStream.listen((message) {
    print('Tin nhắn mới: $message');
  });
}
```

```dart
Stream<String> getMessageStream() async* {
  // Giả sử có một tin nhắn mới đến mỗi 1 giây
  List<String> messages = ['Xin chào!', 'Bạn có khỏe không?',
'Hẹn gặp lại!'];
  for (String message in messages) {
    await Future.delayed(Duration(seconds: 1));
    yield message; // Gửi tin nhắn vào stream
  }
}
```

ASYNC & AWAIT

- **async** và **await** giúp ta viết mã không đồng bộ trông giống như mã đồng bộ, làm cho nó dễ đọc và dễ bảo trì hơn.
- Sử dụng từ khóa **async** để khai báo một hàm không đồng bộ và từ khóa **await** để chờ đợi một **Future** hoàn thành.

```dart
Future<String> fetchUserOrder() async {
  return Future.delayed(Duration(seconds: 2), () => 'Cà phê');
}

void main() async {
  print('Đang chờ đơn hàng...');
  String order = await fetchUserOrder();
  print('Đơn hàng của bạn: $order');
}

// Kết quả:
// Đang chờ đơn hàng...
// (sau 2 giây)
// Đơn hàng của bạn: Cà phê
```

Cảm ơn

Các bạn đã chú ý lắng nghe

HỎI ĐÁP

Q&A