

ĐẠI HỌC BÁCH KHOA HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



SOICT

BÁO CÁO BÀI TẬP LỚN
PHÁT TRIỂN ỨNG DỤNG ĐA NỀN TẢNG

MÃ HỌC PHÂN: IT4788

CHƯƠNG 3: TỔNG QUAN VỀ DART VÀ FLUTTER

Nhóm: 09

Mã lớp học: 159462

Giảng viên hướng dẫn: ThS. Nguyễn Tiến Thành

Danh sách sinh viên thực hiện

STT	Họ tên	Mã sinh viên
1	Nguyễn Ngọc Phúc	20220041
2	Nguyễn Duy Khánh	20225019
3	Nguyễn Phú Hưng	20224862

HÀ NỘI – 2025

MỤC LỤC

PHẦN 1. GIỚI THIỆU ĐỀ TÀI.....	1
1.1 Đặt vấn đề.....	1
1.2 Mục tiêu và phạm vi đề tài.....	2
PHẦN 2. GIỚI THIỆU VỀ NGÔN NGỮ LẬP TRÌNH DART.....	3
2.1 Giới thiệu về ngôn ngữ Dart.....	3
2.1.1 Hoàn cảnh ra đời của ngôn ngữ lập trình Dart.....	3
2.1.2 Lịch sử phát triển của ngôn ngữ lập trình Dart	4
2.1.3 Ứng dụng thực tiễn của Dart	8
2.1.4 Cộng đồng và hệ sinh thái	9
2.2 Đặc điểm của ngôn ngữ lập trình Dart	10
2.3 Các nền tảng thực thi của Dart	16
2.3.1 Dart Native.....	17
2.3.2 Dart Web.....	17
2.4 Công cụ phát triển ứng dụng trên Dart	17
2.4.1 Công cụ chính của Dart SDK	17
2.4.2 Công cụ cho phát triển web.....	19
2.5 Chương trình minh họa	20
PHẦN 3. CÚ PHÁP CƠ BẢN CỦA NGÔN NGỮ DART	23
3.1 Các quy tắc chung trong ngôn ngữ lập trình Dart.....	23
3.2 Biến, hằng, kiểu dữ liệu.....	31
3.2.1 Khai báo biến	31
3.2.2 Khai báo hằng.....	33
3.2.3 Các kiểu dữ liệu	34

3.3 Các toán tử trong Dart.....	44
3.3.1 Toán tử số học	44
3.3.2 Toán tử tăng giảm đơn vị	45
3.3.3 Toán tử logic.....	46
3.3.4 Toán tử so sánh	46
3.3.5 Toán tử gán.....	47
3.3.6 Toán tử điều kiện	47
3.3.7 Phép toán với kiểu String	48
3.3.8 Một số toán tử trên lớp, đối tượng.....	50
3.4 Các cấu trúc điều khiển	50
3.4.1 Các cấu trúc rẽ nhánh	50
3.4.2 Cấu trúc lặp	51
3.4.3 Assert	53
3.4.4 Xử lí ngoại lệ với try/catch và throw, on	54
3.5 Xây dựng hàm	55
3.5.1 Tham số tùy chọn.....	57
3.5.2 Tham số mặc định.....	57
3.5.3 Cú pháp hàm rút gọn(arrow function)	58
3.5.4 Hàm ẩn danh - Lambda - Closure	58
3.6 Quản lý các gói trong Dart.....	59
PHẦN 4. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG TRONG DART.....	66
4.1 Các đặc trưng hướng đối tượng trong Dart	66
4.1.1 Đối tượng và lớp	66
4.1.2 Đóng gói:	70
4.1.3 Kê thừa	71
4.1.4 Trừu tượng hóa:	75

4.1.5 Đa hình	76
4.2 Callable class	77
4.3 Giới thiệu về Async Programming.....	78
4.3.1 Lập trình đồng bộ.....	78
4.3.2 Lập trình bất đồng bộ	79
PHẦN 5. GIỚI THIỆU FRAMEWORK FLUTTER	83
5.1 Giới thiệu về Flutter	83
5.2 Kiến trúc của Flutter	88
5.2.1 Tầng Framework	90
5.2.2 Tầng Engine (C++):	91
5.2.3 Tầng Embedder.....	93
5.3 Ứng dụng Flutter đầu tiên	93
5.4 Flutter widgets.....	100
5.4.1 Giới thiệu khái niệm Widget trong Flutter	100
5.4.2 Phân loại widgets trong Flutter	101
5.4.3 Một số widgets phổ biến trong Flutter	103
5.4.4 Cách tiếp cận khai báo trong Flutter.....	114
5.5 Quản lí trạng thái trong Flutter.....	119
5.5.1 Stateless Widget.....	120
5.5.2 Stateful widget.....	122
5.5.3 Các giải pháp quản lý trạng thái ứng dụng	127
5.6 RenderObject và RenderTree	129
5.6.1 Khái niệm	130
5.6.2 Mối quan hệ giữa các Tree	131
TÀI LIỆU THAM KHẢO.....	136

DANH MỤC HÌNH VẼ

Hình 1.1	Framework Flutter	1
Hình 2.1	Ngôn ngữ lập trình Dart	3
Hình 2.2	Các nhà phát triển chính của ngôn ngữ Dart	4
Hình 2.3	Thông kê các ngôn ngữ lập trình phổ biến năm 2023	10
Hình 2.4	Các đặc điểm chính của ngôn ngữ lập trình Dart	11
Hình 2.5	Các nền tảng thực thi của Dart	16
Hình 2.6	Công cụ Dartpad	21
Hình 3.1	Hình ảnh nội dung file pubspec.yaml trong project thực tế . .	60
Hình 4.1	Các nguyên lý OOP trong Dart	66
Hình 5.1	Flutter - Framework phát triển ứng dụng đa nền tảng	83
Hình 5.2	Giao diện chính app MyBMW - sử dụng Flutter	85
Hình 5.3	Danh sách công ty lớn sử dụng Flutter	86
Hình 5.4	Chương trình minh họa in ra chữ " <i>Hello, Flutter!</i> "	88
Hình 5.5	Kiến trúc tổng thể của một ứng dụng Flutter	89
Hình 5.6	Chi tiết về lớp Framework	90
Hình 5.7	Kiến trúc chi tiết lớp Engine	91
Hình 5.8	Kiến trúc chi tiết lớp Embedder	93
Hình 5.9	Công cụ Android Studio	94
Hình 5.10	Cấu trúc của một dự án Flutter	95
Hình 5.11	Kết quả của chương trình minh họa	98
Hình 5.12	Cấu trúc widget của ứng dụng <i>Hello World</i>	103
Hình 5.13	Elevated Button trong Flutter	104
Hình 5.14	Textfield trong Flutter	106
Hình 5.15	Checkbox trong Flutter	106
Hình 5.16	Minh họa về widget Text	107
Hình 5.17	Minh họa về widget Row	108
Hình 5.18	Thuộc tính <i>mainAxisAlignment</i> có giá trị <i>center</i>	108
Hình 5.19	Thuộc tính <i>mainAxisAlignment</i> có giá trị <i>start</i>	108
Hình 5.20	Thuộc tính <i>mainAxisAlignment</i> có giá trị <i>spaceBetween</i>	109
Hình 5.21	Minh họa về widget Column	110
Hình 5.22	Minh họa về widget ListView	111
Hình 5.23	Minh họa về widget Stack	112

Hình 5.24 Minh họa về widget Scaffold	113
Hình 5.25 Mô hình hóa hàm trạng thái	114
Hình 5.26 Sơ đồ cây widget	117
Hình 5.27 Trạng thái của Widget	120
Hình 5.28 Vòng đời của StatelessWidget	121
Hình 5.29 Vòng đời của StatefulWidget	123
Hình 5.30 Counter App Demo	125
Hình 5.31 Các giải pháp quản lý trạng thái ứng dụng	129
Hình 5.32 Các tầng Logic trong hệ thống UI của Flutter	130
Hình 5.33 Mối quan hệ giữa các cây	131

DANH MỤC BẢNG BIỂU

Bảng 2.1	Bảng so sánh các đặc điểm giữa Dart và JavaScript	16
Bảng 3.1	Mô tả các kiểu dữ liệu cơ bản trong Dart	35
Bảng 3.2	Các toán tử truy cập lớp/đối tượng trong Dart.	50
Bảng 5.1	Các thuộc tính của ElevatedButton trong Flutter	105

DANH MỤC THUẬT NGỮ VÀ TỪ VIẾT TẮT

Thuật ngữ	Ý nghĩa
AOT	Ahead-of-time
HTML	Ngôn ngữ đánh dấu siêu văn bản (HyperText Markup Language)
JIT	Just-in-time
URI	Định danh tài nguyên thông nhất (Uniform Resource Identifier)
VM	Máy ảo (Virtual Machine)

PHẦN 1. GIỚI THIỆU ĐỀ TÀI

1.1 Đặt vấn đề

Trong những năm gần đây, xu hướng phát triển ứng dụng đa nền tảng ngày càng phổ biến do nhu cầu triển khai đồng thời trên nhiều hệ điều hành khác nhau như Android, iOS, Windows và Web. Thay vì xây dựng từng ứng dụng riêng biệt cho từng nền tảng, các doanh nghiệp và nhà phát triển mong muốn có thể tiết kiệm thời gian, chi phí và nguồn lực bằng cách xây dựng một ứng dụng duy nhất có khả năng hoạt động tốt trên nhiều thiết bị.



Hình 1.1: Framework Flutter

Flutter – một framework được phát triển bởi Google – cùng với ngôn ngữ Dart, đã trở thành một trong những lựa chọn hàng đầu cho phát triển ứng dụng đa nền tảng. Với khả năng hỗ trợ giao diện người dùng phong phú, hiệu suất cao và công nghệ “hot reload”, Flutter đã nhanh chóng chiếm lĩnh thị trường và được cộng đồng phát triển ứng dụng đón nhận rộng rãi.

Trong bối cảnh đó, việc nghiên cứu và nắm bắt tổng quan về Dart và Flutter không chỉ mang lại lợi ích học thuật cho sinh viên ngành công nghệ thông tin, mà còn mở ra những cơ hội nghề nghiệp tiềm năng trong thị trường lao động hiện nay. Chính vì vậy, đề tài này được thực hiện nhằm mục tiêu tìm hiểu, phân tích và hệ

thống hóa kiến thức cơ bản về Dart và Flutter – từ cú pháp, lập trình hướng đối tượng trong Dart, đến kiến trúc Flutter và cách xây dựng một ứng dụng cơ bản.

1.2 Mục tiêu và phạm vi đề tài

Hiện nay, nhiều công nghệ như React Native, Xamarin hay NativeScript đã được phát triển nhằm hỗ trợ xây dựng ứng dụng đa nền tảng. Tuy nhiên, các framework này vẫn tồn tại nhiều hạn chế về hiệu suất, khả năng tùy biến giao diện, và độ tương thích với hệ điều hành. Flutter nổi lên như một giải pháp khắc phục nhiều nhược điểm trên nhờ kiến trúc widget thống nhất, khả năng kết xuất (rendering) mạnh mẽ, và hiệu năng tiệm cận native.

Mặc dù cộng đồng phát triển Flutter đang phát triển nhanh chóng, sinh viên, đặc biệt là những lập trình viên mới vẫn gặp khó khăn trong việc tiếp cận kiến thức một cách hệ thống do thiếu tài liệu tổng hợp rõ ràng, dễ hiểu. Do đó, đề tài này hướng đến việc biên soạn và trình bày tổng quan một cách logic, trực quan và đầy đủ nhất có thể về Dart và Flutter, phục vụ mục đích học tập, giảng dạy và phát triển ứng dụng thực tế.

Đề tài tập trung nghiên cứu những khía cạnh nền tảng về Dart và Flutter bằng phương pháp nghiên cứu tổng hợp tài liệu kết hợp thực nghiệm cài đặt. Đề tài sẽ mở đầu bằng việc phác họa quá trình hình thành và phát triển của Dart, bối cảnh ra đời cũng như mục tiêu thiết kế của ngôn ngữ. Tiếp theo, nội dung sẽ lần lượt đi sâu vào cú pháp cơ bản, cách Dart hiện thực hóa các nguyên lý lập trình hướng đối tượng, cấu trúc và phong cách định nghĩa hàm, đồng thời giới thiệu hệ sinh thái quản lý gói (pub). Trên nền tảng kiến thức Dart, đề tài chuyển sang Flutter, bắt đầu với kiến trúc và cơ chế render đồ họa. Tiếp đó, đề tài sẽ giới thiệu về hệ thống widget trong ứng dụng Flutter. Cuối cùng, toàn bộ khái niệm sẽ được gắn kết thông qua quy trình xây dựng một ứng dụng mẫu: khởi tạo project, thiết kế UI, xử lý logic, đóng gói và chạy thử trên nhiều nền tảng. Đề tài cung cấp cái nhìn toàn diện, tổng quan về Dart & Flutter, qua đó trang bị lộ trình học tập rõ ràng cho người mới bắt đầu.

PHẦN 2. GIỚI THIỆU VỀ NGÔN NGỮ LẬP TRÌNH DART

Dart, một ngôn ngữ lập trình đa năng do Google phát triển, đã chuyển mình từ một ứng cử viên thay thế JavaScript thành một trụ cột quan trọng trong phát triển ứng dụng đa nền tảng hiện đại. Được công bố rộng rãi vào năm 2011, Dart giờ đây hỗ trợ các ứng dụng trên web, di động, máy tính để bàn và phía máy chủ, nhờ vào sự kết hợp với Flutter cùng bộ tính năng mạnh mẽ. Phần này sẽ giới thiệu nguồn gốc, các mốc phát triển quan trọng và vai trò của Dart trong việc giúp các nhà phát triển tạo ra ứng dụng đa nền tảng hiệu suất cao từ một mã nguồn duy nhất.



Hình 2.1: Ngôn ngữ lập trình Dart

2.1 Giới thiệu về ngôn ngữ Dart

2.1.1 Hoàn cảnh ra đời của ngôn ngữ lập trình Dart

Dart là một ngôn ngữ lập trình được Google giới thiệu nhằm khắc phục những hạn chế của JavaScript trong việc xây dựng các ứng dụng web lớn, đặc biệt về hiệu suất và khả năng bảo trì. Để hiện thực hóa mục tiêu này, Google đã giao cho hai kỹ sư Lars Bak và Kasper Lund phát triển Dart, và ngôn ngữ này chính thức được công bố tại hội nghị GOTO ở Aarhus, Đan Mạch, vào ngày 10 tháng 10 năm 2011. Sự kiện ra mắt này đánh dấu bước tiến quan trọng trong việc cung cấp một công cụ lập trình hiện đại, dễ học và có hiệu suất cao. Đặc biệt, Dart được thiết kế để có thể biên dịch sang JavaScript, giúp đảm bảo tính tương thích với các trình duyệt web phổ biến, đồng thời hỗ trợ phát triển ứng dụng trên nhiều nền tảng khác nhau như web, di động và máy tính.



(a) Lars Bak



(b) Kasper Lund

Hình 2.2: Các nhà phát triển chính của ngôn ngữ Dart

2.1.2 Lịch sử phát triển của ngôn ngữ lập trình Dart

Qua hơn một thập kỷ, Dart đã chuyển mình từ một ý tưởng đầy tham vọng thành một ngôn ngữ đa năng, hỗ trợ phát triển ứng dụng trên nhiều nền tảng như web, di động, máy tính để bàn và server. Hành trình trở thành nền tảng phát triển ứng dụng di động hàng đầu của Flutter không dễ dàng mà phải trải qua nhiều tranh chấp pháp lý cũng như những bước tiến vượt bậc về công nghệ. Lịch sử phát triển của Dart có thể được tóm lược qua các giai đoạn chính sau:

a, Dart 1.0 (2011 - 2013)

Dart được công bố lần đầu vào ngày 10 tháng 10 năm 2011 tại hội nghị GOTO ở Aarhus, Đan Mạch. Mục tiêu của Dart là khắc phục các hạn chế của JavaScript, đặc biệt về hiệu suất, khả năng bảo trì mã nguồn, và trải nghiệm lập trình cho các ứng dụng web lớn.

Tháng 11 năm 2013, Dart 1.0 được phát hành, đánh dấu phiên bản ổn định đầu tiên. Dart 1.0 bao gồm các thành phần chính:

- **Dart Virtual Machine (Dart VM)** cho phép chạy trực tiếp ứng dụng Dart.
- **Trình biên dịch dart2js** để chuyển mã Dart thành JavaScript.
- **Công cụ Dart Editor** hỗ trợ lập trình viên viết mã hiệu quả.

Dù được Google hỗ trợ mạnh mẽ, Dart gặp khó khăn trong việc cạnh tranh với JavaScript do sự phổ biến và hệ sinh thái đã phát triển của ngôn ngữ này. Các trình duyệt web phụ thuộc sâu sắc vào JavaScript, khiến Dart khó thay thế như kỳ vọng

ban đầu.

b, Dart 2.0 (8/2018)

Tháng 8 năm 2018, Google chính thức giới thiệu **Dart 2.0**, đánh dấu một bước ngoặt quan trọng trong lịch sử phát triển của ngôn ngữ lập trình này. Phiên bản này không chỉ cải thiện hiệu suất mà còn mang đến những thay đổi nền tảng, giúp Dart trở nên mạnh mẽ và linh hoạt hơn.

Dart 2.0 đánh dấu bước chuyển quan trọng từ hệ thống kiểu tùy chọn sang hệ thống kiểu tĩnh bắt buộc, cung cấp nền tảng *sound static typing* của ngôn ngữ. Bước chuyển này cho phép trình phân tích tĩnh phát hiện lỗi kiểu ngay trong quá trình biên dịch, thay vì đợi tới lúc chạy chương trình. Việc bắt lỗi sớm giúp lập trình viên viết mã an toàn hơn, đồng thời gia tăng độ tin cậy và khả năng dự đoán của ứng dụng, cải thiện hiệu suất và giảm chi phí gỡ lỗi nhờ loại bỏ nhiều lỗi kiểu xuất hiện ở *runtime*. Hệ thống kiểu tĩnh mạnh mẽ cũng khiến Dart thu hút những lập trình viên ưa thích *type safety*, đặc biệt là những lập trình viên quen thuộc với ngôn ngữ lập trình *Java* hoặc *C#* khi có thể tận dụng các công cụ quen thuộc mà vẫn hưởng lợi từ cú pháp gọn nhẹ của Dart. Song song với tính nghiêm ngặt, Dart vẫn giữ khả năng suy luận kiểu và cho phép dùng từ khóa *dynamic* khi cần sự linh hoạt cao, qua đó cân bằng giữa an toàn và tiện dụng trong phát triển ứng dụng đa nền tảng.

Cải tiến hiệu suất nhờ biên dịch AOT và JIT:

Dart 2.0 tận dụng hai phương thức biên dịch mạnh mẽ:

- **Ahead-of-Time (AOT):** Chuyển mã Dart thành mã máy native trước khi chạy, giúp ứng dụng khởi động nhanh và hoạt động mượt mà, đặc biệt phù hợp với các ứng dụng di động và máy tính để bàn.
- **Just-in-Time (JIT):** Cho phép cập nhật mã nguồn ngay lập tức mà không cần biên dịch lại toàn bộ, hỗ trợ tính năng *hot reload* – một công cụ quan trọng giúp tăng tốc quá trình phát triển ứng dụng.

Định hướng đa nền tảng:

Trước đây, *Dart* chủ yếu tập trung vào phát triển web như một lựa chọn thay thế cho *JavaScript*. Sự tập trung đó đã được mở rộng với sự ra đời của *Dart 2.0*, khi *Google* hướng ngôn ngữ này tới cả nền tảng di động và máy tính để bàn. Tầm nhìn mới này đặt nền móng cho *Flutter*, một *framework* đa nền tảng sử dụng *Dart* làm ngôn ngữ chính. Sự xuất hiện của *Flutter* không chỉ mở rộng phạm vi ứng dụng mà còn tái định vị *Dart 2.0* như một ngôn ngữ lập trình đa năng, sẵn sàng cạnh tranh trong nhiều lĩnh vực.

c, Sự bùng nổ: Flutter và Dart (2018 - nay)

Năm 2018, *Google* ra mắt *Flutter*, một *framework* giao diện người dùng (UI) mã nguồn mở cho phép phát triển ứng dụng *native* trên di động, web và máy tính để bàn từ cùng một mã nguồn. *Framework* này chọn *Dart* làm ngôn ngữ lập trình chính, tận dụng những ưu điểm nổi bật của *Dart* để tạo nên một công cụ phát triển mạnh mẽ và hiệu quả.

- Tính năng *hot reload* giúp tăng tốc phát triển ứng dụng**

Tính năng *hot reload*, được hỗ trợ bởi biên dịch *JIT* của *Dart*, cho phép nhà phát triển thấy ngay kết quả thay đổi mã nguồn mà không cần khởi động lại ứng dụng. *Hot reload* giúp tiết kiệm thời gian, tăng năng suất và đặc biệt hữu ích trong việc thiết kế, tinh chỉnh giao diện người dùng.

- Biên dịch *native* hiệu quả nhờ AOT kết hợp JIT**

Cơ chế biên dịch *AOT* của *Dart* tạo mã *native* cho các nền tảng như *Android* và *iOS*, bảo đảm hiệu suất cao và trải nghiệm mượt mà, sánh ngang ứng dụng *native* truyền thống. Nhờ đó, *Flutter* có thể cạnh tranh trực tiếp với các công nghệ đa nền tảng như *React Native* hay *Xamarin*.

- Hệ thống *widget* linh hoạt giúp thiết kế giao diện dễ dàng**

Hệ thống *widget* phong phú cùng cú pháp rõ ràng của *Flutter* cho phép xây

dựng giao diện người dùng phức tạp, đẹp mắt mà vẫn đơn giản trong triển khai. Việc dùng một ngôn ngữ duy nhất cho cả logic và giao diện giúp giảm thiểu sự phức tạp khi phải chuyển đổi giữa nhiều ngôn ngữ.

Nhờ *Flutter*, *Dart* từ một ngôn ngữ chủ yếu phục vụ lập trình web đã trở thành lựa chọn hàng đầu cho phát triển ứng dụng đa nền tảng. Sự phổ biến của *Flutter* đã thúc đẩy mạnh mẽ việc áp dụng *Dart*, thu hút hàng triệu nhà phát triển trên toàn cầu.

d, Dart 2.12: Null Safety (2021)

Vào tháng 2 năm 2021, **Dart 2.12** được phát hành, mang đến một trong những tính năng quan trọng nhất trong lịch sử của Dart: **Null Safety**. Đây là một bước tiến lớn nhằm nâng cao độ an toàn và tin cậy của mã nguồn.

Null Safety giúp loại bỏ lỗi *null references*. Tính năng này cho phép lập trình viên chỉ định rõ ràng ngay từ lúc thiết kế liệu một biến có thể nhận giá trị *null* hay không. Sự chỉ định rõ ràng đó ngăn chặn lỗi *null pointer exceptions* – một vấn đề phổ biến và khó chịu trong lập trình. Nhờ phát hiện và xử lý các trường hợp *null* tại thời điểm biên dịch, *Null Safety* nâng cao chất lượng mã, giảm thiểu lỗi *runtime* và cải thiện hiệu suất ứng dụng.

Phiên bản mới này nhận được sự hoan nghênh nhiệt liệt từ cộng đồng lập trình Dart. *Null Safety* không chỉ giúp mã nguồn trở nên đáng tin cậy hơn mà còn khuyến khích các nhà phát triển viết mã sạch hơn, dễ bảo trì hơn. Đây cũng là minh chứng cho sự trưởng thành của Dart như một ngôn ngữ lập trình hiện đại, đáp ứng các tiêu chuẩn an toàn mã nguồn ngày càng cao.

e, Dart 3.x (2023 - 2025)

Dart tiếp tục phát triển mạnh mẽ với các phiên bản mới, mang đến những tính năng tiên tiến để đáp ứng nhu cầu của các nhà phát triển trong bối cảnh công nghệ không ngừng thay đổi.

- **Dart 3.0 (2023):** Được phát hành vào năm 2023, Dart 3.0 đánh dấu một cột

mốc quan trọng khi yêu cầu tất cả mã nguồn phải tuân thủ Null Safety, đảm bảo tính nhất quán và an toàn trong toàn bộ hệ sinh thái Dart. Ngoài ra, phiên bản này giới thiệu:

- **Records:** Một cách mới để nhóm các giá trị liên quan mà không cần tạo lớp đầy đủ, giúp mã nguồn gọn gàng và dễ đọc hơn.
- **Patterns:** Hỗ trợ phân tích và so khớp cấu trúc dữ liệu, giúp xử lý dữ liệu phức tạp một cách hiệu quả.
- **Class modifiers:** Cung cấp khả năng kiểm soát tốt hơn việc kế thừa và sử dụng các lớp trong mã nguồn.
- **Dart 3.4 (2024):** Phiên bản này bổ sung hỗ trợ biên dịch sang **WebAssembly (Wasm)** – một định dạng mã nhị phân cho phép chạy mã với hiệu suất gần native trên trình duyệt. Điều này mở ra tiềm năng lớn cho các ứng dụng Dart trên web, giúp chúng hoạt động nhanh hơn và hiệu quả hơn, cạnh tranh trực tiếp với JavaScript trong các ứng dụng đòi hỏi hiệu suất cao.
- **Dart 3.7 (2025):** Phiên bản dự kiến ra mắt trong năm 2025. Phiên bản này có những điểm mới:

- **Wildcard variables:** Cho phép bỏ qua các giá trị không cần thiết trong các biểu thức, giúp mã nguồn ngắn gọn và dễ hiểu hơn.
- **Cải tiến định dạng mã nguồn:** Tăng cường khả năng tự động hóa và chuẩn hóa cách trình bày mã, nâng cao tính nhất quán và dễ bảo trì.

2.1.3 Ứng dụng thực tiễn của Dart

Dart ngày càng được ứng dụng rộng rãi, mở rộng ra ngoài phạm vi web và di động, nhờ khả năng thích ứng linh hoạt trên nhiều nền tảng. Khả năng linh hoạt này được thể hiện rõ trong phát triển backend, khi các thư viện như Dart Frog cung cấp giải pháp hiệu quả cho việc xây dựng server-side. Hiệu quả tương tự cũng được khẳng định ở nền tảng web, với Flutter Web và AngularDart hỗ trợ lập trình viên

xây dựng ứng dụng giàu tính tương tác từ một cơ sở mã duy nhất. Lợi ích của cơ sở mã duy nhất còn giúp Dart dễ dàng triển khai ứng dụng máy tính để bàn bằng cách biên dịch native trực tiếp cho Windows, macOS và Linux. Không dừng lại ở đó, tính nhỏ gọn và hiệu quả trong biên dịch native còn giúp Dart trở thành lựa chọn lý tưởng cho phát triển các hệ thống IoT và dự án Fuchsia OS.

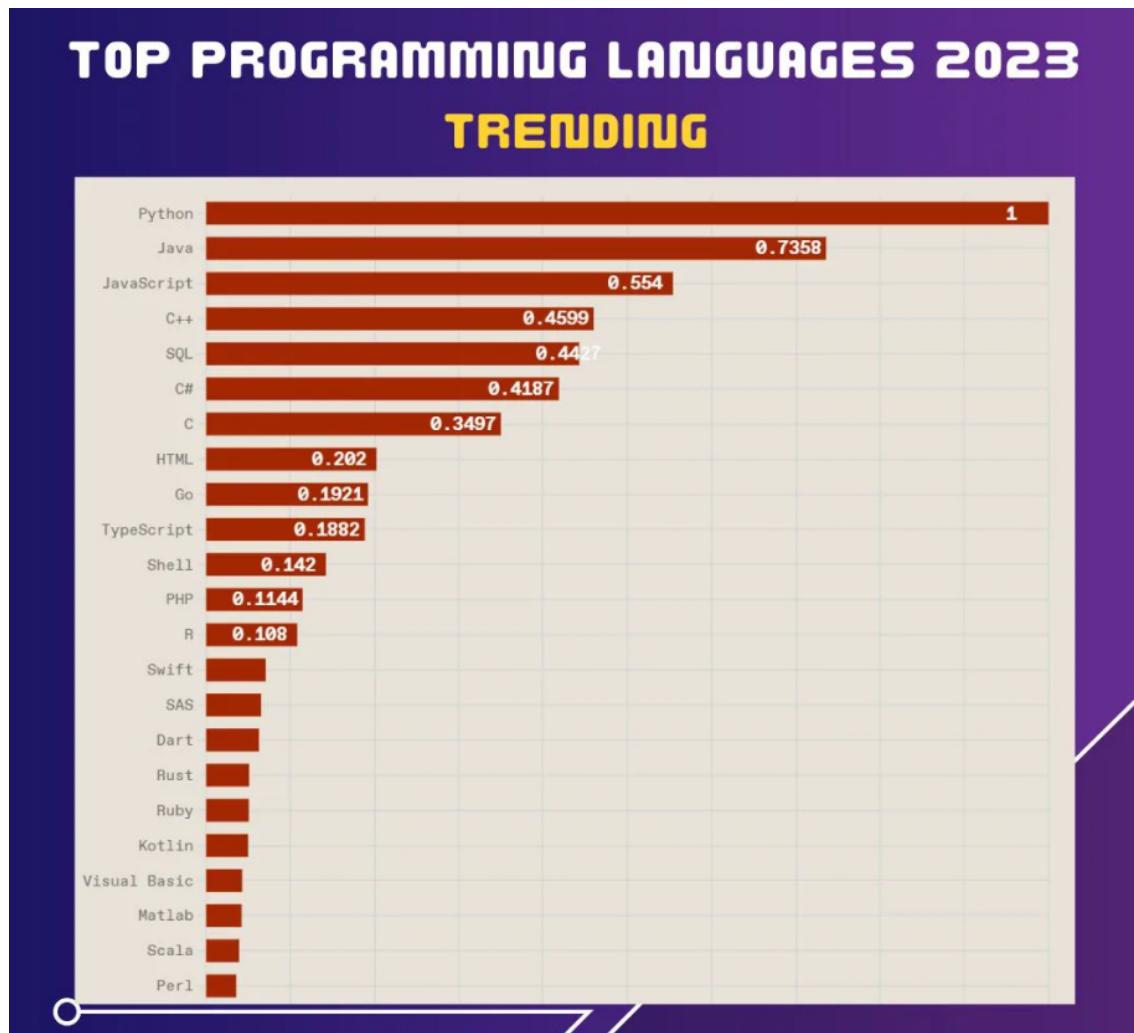
Sự đa dạng về ứng dụng và tính linh hoạt của Dart được tăng cường hơn nữa nhờ hệ sinh thái Flutter, cho phép lập trình viên phát triển ứng dụng đa nền tảng chất lượng cao chỉ từ một mã nguồn duy nhất, đồng thời hưởng lợi từ quy trình phát triển nhanh và trải nghiệm tinh chỉnh trực quan.

2.1.4 Cộng đồng và hệ sinh thái

Dart hiện sở hữu một cộng đồng phát triển năng động và ngày càng mở rộng. Hệ sinh thái Dart bao gồm hàng loạt thư viện, packages và công cụ hỗ trợ đa dạng, từ phát triển ứng dụng di động với Flutter đến lập trình web và server. Sự hỗ trợ mạnh mẽ từ Google cùng với đóng góp từ cộng đồng đã giúp Dart duy trì vị thế là một ngôn ngữ lập trình quan trọng trong thế giới phần mềm hiện đại.

Hệ sinh thái Dart không ngừng mở rộng, thể hiện ở các hội nghị thường niên quy mô lớn như *Flutter Engage* và *Flutter Forward*, ở công cụ DartPad chạy trực tiếp trên trình duyệt cho phép viết và thực thi mã Dart tức thì, cũng như ở kho gói pub.dev với hàng chục nghìn thư viện mã nguồn mở do cộng đồng đóng góp.

Dart 2.0 giới thiệu hệ thống kiểu tĩnh mạnh mẽ; nền tảng kiểu an toàn này đã mở đường cho sự bùng nổ của Flutter. Sự bùng nổ đó, cùng với *Null Safety* được bổ sung trong Dart 2.12, đã dẫn Dart tiến nhanh tới chuỗi phiên bản 3.x (3.0, 3.4, 3.7) với nhiều cải tiến đáng kể. Quá trình nâng cấp liên tục ấy biến Dart thành một ngôn ngữ đa năng và vững mạnh. Nhờ vậy, theo thống kê năm 2023, Dart đã vươn lên vị trí Top 16 ngôn ngữ lập trình được sử dụng nhiều nhất, khẳng định chỗ đứng vững chắc trong cộng đồng phát triển.

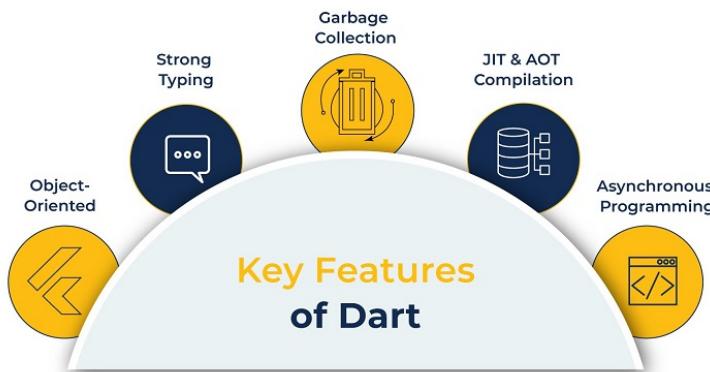


Hình 2.3: Thống kê các ngôn ngữ lập trình phổ biến năm 2023

Với vai trò mở rộng trong nhiều lĩnh vực và sự hỗ trợ từ cộng đồng, Dart hứa hẹn sẽ tiếp tục là một công cụ không thể thiếu cho các nhà phát triển trong tương lai.

2.2 Đặc điểm của ngôn ngữ lập trình Dart

Dart là một ngôn ngữ lập trình hiện đại. Ngôn ngữ này chứng tỏ vị thế nổi bật trong phát triển ứng dụng đa nền tảng nhờ khả năng tích hợp chặt chẽ với *Flutter* – framework giao diện người dùng (UI) do Google phát triển. Sự tích hợp đó cho phép **Dart** tạo ra giao diện mượt mà và hiệu năng cao trên thiết bị di động, trình duyệt web và máy tính để bàn. Ngoài ưu thế tích hợp, cú pháp gần gũi với C, Java và JavaScript giúp lập trình viên đã quen các ngôn ngữ này nhanh chóng nắm bắt cú pháp **Dart**.



Hình 2.4: Các đặc điểm chính của ngôn ngữ lập trình Dart

Khả năng thích ứng đa nền tảng tiếp tục được củng cố bởi hệ thống biên dịch linh hoạt của **Dart**. Cơ chế *ahead-of-time* (AOT) tối ưu hiệu suất thực thi, trong khi *just-in-time* (JIT) hỗ trợ tính năng *hot reload* giúp rút ngắn vòng lặp chỉnh sửa–kiểm thử. Kết hợp với kiểu tĩnh an toàn và bộ thư viện phong phú, tập đặc điểm đó định vị **Dart** thành lựa chọn lý tưởng cho nhà phát triển đang tìm kiếm một ngôn ngữ duy nhất nhưng đủ mạnh để triển khai ứng dụng trên mọi nền tảng. Những đặc điểm nổi bật dưới đây đã giúp Dart trở thành một lựa chọn lý tưởng cho các nhà phát triển ứng dụng:

- **Lập trình bất đồng bộ (Asynchronous Programming):** Dart hỗ trợ lập trình bất đồng bộ thông qua các từ khóa `async` và `await`, cho phép xử lý các tác vụ không đồng bộ như gọi API, tải dữ liệu, hoặc xử lý sự kiện mà không làm gián đoạn giao diện người dùng. Tính năng này rất quan trọng khi phát triển ứng dụng đa nền tảng, vì nó giúp xây dựng các giao diện động và phản hồi nhanh. Tính năng này giúp Dart trở thành lựa chọn lý tưởng cho các ứng dụng đòi hỏi hiệu suất cao và giao diện mượt mà.

Ví dụ, một hàm bất đồng bộ trong Dart có thể được viết như sau:

```
Future<String> fetchData() async {
    await Future.delayed(Duration(seconds: 2));
}
```

```
        return "Loading data successfully!";  
    }  
  
}
```

- **Hot Reload:** Hot Reload là một trong những tính năng nổi bật nhất khi Dart kết hợp với Flutter, mang lại các tiện ích như :
 - **Thay đổi mã nguồn ngay lập tức:** Cho phép lập trình viên thay đổi code và xem ngay kết quả trên giao diện mà không cần khởi động lại toàn bộ ứng dụng.
 - **Giữ nguyên trạng thái ứng dụng:** Không làm mất dữ liệu người dùng hoặc trạng thái hiện tại khi reload.
 - **Giảm thời gian phát triển:** Giúp tinh chỉnh giao diện UI/UX nhanh chóng, thử nghiệm nhiều phương án chỉ trong vài giây.
 - **Ổn định bộ nhớ:** Hạn chế hiện tượng rò rỉ bộ nhớ vốn dễ gặp trong các ngôn ngữ cần build lại toàn bộ như C/C++.
- **Mã nguồn mở (Open Source):** Dart được phát hành dưới giấy phép BSD-style, nhờ đó cộng đồng lập trình viên toàn cầu có thể trực tiếp tham gia phát triển, báo lỗi và đề xuất cải tiến. Sự tham gia rộng rãi ấy đã hình thành kho thư viện phong phú trên pub.dev với hàng chục nghìn package mã nguồn mở, mở rộng khả năng của Dart trong hầu hết các lĩnh vực. Kho thư viện không ngừng lớn mạnh tiếp tục thúc đẩy nhu cầu về tư liệu học tập; tài liệu chính thức tại dart.dev và flutter.dev vì thế luôn được cập nhật thường xuyên, hỗ trợ đồng thời người mới bắt đầu và lập trình viên chuyên nghiệp. Việc tài liệu và thư viện phát triển liên tục góp phần mở rộng hệ sinh thái dựa trên Dart: từ Flutter và AngularDart đến các giải pháp server-side như Shelf hay Dart Frog. Môi trường mã nguồn mở toàn diện như vậy đã thúc đẩy sự phát triển nhanh chóng và bền vững của Dart kể từ khi ra đời.
- **Kiểu tĩnh (Statically Typing):** Dart sử dụng hệ thống kiểu dữ liệu tĩnh với

thiết kế âm thanh (sound type system), giúp:

- **Phát hiện lỗi sớm:** Kiểm tra kiểu dữ liệu ngay tại thời điểm biên dịch, giảm thiểu lỗi runtime.
- **Cải thiện độ an toàn:** Bắt lỗi kiểu dữ liệu hoặc thiếu Null Safety ngay khi lập trình, trước khi ứng dụng được chạy thực tế.
- **Hỗ trợ Null Safety (từ Dart 2.12):**
 - * Các biến phải khai báo rõ ràng nếu có thể nhận giá trị null (ví dụ: `int? value;`).
 - * Loại bỏ hoàn toàn lỗi phỗ biến `NullPointerException`.
- **Giữ cú pháp ngắn gọn:** Với cơ chế suy luận kiểu (type inference), Dart cho phép viết mã ngắn mà vẫn đảm bảo an toàn:

```
var name = 'Dart'; // kiểu String
```

Hệ thống kiểu của Dart mang lại sự cân bằng tuyệt vời giữa hiệu suất, độ an toàn và trải nghiệm lập trình linh hoạt.

- **Tính di động (Portability):** Dart được thiết kế để trở thành ngôn ngữ "**write once, run anywhere**", cho phép biên dịch mã nguồn sang nhiều nền tảng khác nhau mà không cần thay đổi logic chính.
 - **Native code (AOT):** Cơ chế *ahead-of-time compilation* của Dart biên dịch trực tiếp mã nguồn sang mã máy ARM hoặc x86. Cấu trúc biên dịch ấy cho phép tạo các gói cài đặt *APK*, *IPA* hay tệp *EXE* cho ứng dụng Flutter trên di động và máy tính để bàn, đồng thời bảo đảm hiệu suất native và thời gian khởi chạy nhanh.
 - **JavaScript (dart2js):** Trình biên dịch *dart2js* chuyển đổi mã Dart sang JavaScript được tối ưu cho trình duyệt. Quy trình này áp dụng kỹ thuật *tree-shaking* nhằm loại bỏ mã thừa, giảm kích thước tệp sinh ra và đạt

hiệu suất cao hơn khoảng 20 % so với JavaScript viết tay trong các phép đo chuẩn.

- **WebAssembly (từ Dart 3.4):** Kể từ phiên bản 3.4, Dart hỗ trợ biên dịch sang *WebAssembly* thông qua phần mở rộng *WasmGC*. Cách biên dịch này nâng hiệu suất tải và thực thi ứng dụng web thêm 30–50 % so với JavaScript.
- **Server-side và file thực thi độc lập:** Trong giai đoạn phát triển, mã Dart có thể chạy trực tiếp trên *Dart VM*; khi triển khai, cùng một mã nguồn ấy được biên dịch thành tệp thực thi độc lập cho môi trường sản xuất. Ví dụ tiêu biểu là hạ tầng Google Fiber đã dùng Dart ở server-side để xử lý khoảng 10 000 yêu cầu mỗi giây.
- **Casestudy đa nền tảng:** Ứng dụng Google Ads minh họa rõ ràng tính “write once, run anywhere”: một codebase Dart duy nhất phục vụ đồng thời năm nền tảng—Android, iOS, web, Windows và macOS—giúp rút ngắn ước tính 70 % thời gian phát triển so với duy trì năm dự án riêng rẽ.
- **Công cụ hỗ trợ mạnh mẽ (Productive Tooling):** Dart đi kèm với các công cụ phát triển mạnh mẽ, bao gồm DartPad (một trình biên tập trực tuyến cho phép viết và chạy mã Dart mà không cần cài đặt), và tích hợp tốt với các IDE như Visual Studio Code, IntelliJ IDEA, và Android Studio. Các công cụ này cung cấp tính năng gợi ý mã, kiểm tra lỗi thời gian thực, và gỡ lỗi, giúp tăng năng suất lập trình. Ngoài ra, Dart còn có hệ thống quản lý gói (package manager) gọi là `pub`, giúp dễ dàng quản lý và tích hợp các thư viện bên ngoài.
- **Thu gom rác (Garbage Collection):** Dart có hệ thống quản lý bộ nhớ tự động thông qua thu gom rác, giúp giảm thiểu lỗi liên quan đến bộ nhớ như rò rỉ bộ nhớ (memory leak). Hệ thống này tự động giải phóng bộ nhớ không còn sử dụng, đảm bảo hiệu suất ổn định cho ứng dụng, đặc biệt là các ứng dụng lớn và phức tạp. Theo bài kiểm tra của Google, Dart GC giảm 50% thời gian tạm

dùng so với JavaScript V8 trong các ứng dụng phức tạp.

- **Nền tảng cho Flutter:** Dart là ngôn ngữ chính thức của Flutter, một framework của Google để xây dựng ứng dụng giao diện người dùng đa nền tảng. Sự kết hợp giữa Dart và Flutter mang lại hiệu suất cao nhờ biên dịch AOT, cùng với khả năng tạo giao diện đẹp mắt và đồng nhất trên các nền tảng. Tính năng Hot Reload và lập trình bắt đồng bộ của Dart đặc biệt hữu ích trong Flutter, giúp lập trình viên phát triển ứng dụng nhanh chóng và hiệu quả.
- **Hỗ trợ lập trình hướng đối tượng (Object-Oriented Programming):** Dart là một ngôn ngữ hướng đối tượng hoàn chỉnh, hỗ trợ các khái niệm như lớp, giao diện, mixin, và lớp trừu tượng. Điều này cho phép lập trình viên xây dựng các ứng dụng có cấu trúc rõ ràng và dễ bảo trì.

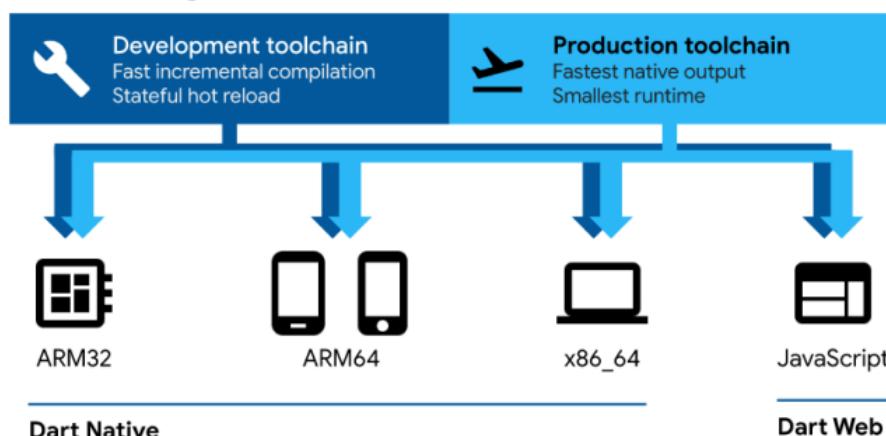
Lựa chọn ngôn ngữ lập trình phù hợp là yếu tố then chốt bảo đảm hiệu suất và khả năng mở rộng của ứng dụng. Sự lựa chọn này thường được cân nhắc giữa hai ngôn ngữ phổ biến là Dart và JavaScript, mỗi ngôn ngữ sở hữu những đặc điểm riêng. Các đặc điểm ấy sẽ được trình bày trong Bảng so sánh các tiêu chí giữa Dart và JavaScript như sau:

Tiêu chí	Dart	JavaScript
Kiểu dữ liệu	Kiểu tĩnh (Statically Typed), hỗ trợ Null Safety	Kiểu động (Dynamically Typed)
Thu gom rác	Bộ thu gom rác thế hệ (Generational GC)	Bộ thu gom rác incremental truyền thống
Biên dịch	AOT và JIT compilation, hỗ trợ native	Chạy trực tiếp trên trình duyệt (interpreted)
Công cụ hỗ trợ	DartPad, Flutter DevTools, Analyzer	Chrome DevTools, VSCode, Node.js Tools
Tính di động	Web, mobile, desktop, server, IoT, Fuchsia OS	Web, server (Node.js)
Hiệu suất	Native tốc độ cao với AOT, nhanh hơn trong mobile apps	Tối ưu tốt trên trình duyệt, nhưng mobile native chậm hơn Dart+Flutter

Bảng 2.1: Bảng so sánh các đặc điểm giữa Dart và JavaScript

2.3 Các nền tảng thực thi của Dart

Dart hỗ trợ hai nền tảng chính cho việc phát triển ứng dụng: **Dart Native** và **Dart Web**. Như được minh họa trong *Hình 2.5*, mỗi nền tảng có các đặc điểm và mục đích sử dụng riêng biệt, cho phép các nhà phát triển xây dựng ứng dụng đa nền tảng một cách hiệu quả.



Hình 2.5: Các nền tảng thực thi của Dart

2.3.1 Dart Native

Dart Native được thiết kế để phát triển ứng dụng di động và máy tính để bàn. Nó cho phép biên dịch mã Dart thành mã máy bản xứ, đảm bảo hiệu suất cao trên các nền tảng như Android, iOS, Windows, macOS, và Linux. Dart Native hỗ trợ các kiến trúc phần cứng như ARM32, ARM64 (cho thiết bị di động) và x86_64 (cho máy tính để bàn).

Trong quá trình phát triển, Dart Native cung cấp:

- **Biên dịch tăng cường nhanh chóng:** Giúp lập trình viên thấy ngay kết quả thay đổi mã mà không cần khởi động lại ứng dụng.
- **Hot reload:** Cho phép cập nhật mã nguồn trong thời gian thực, giữ nguyên trạng thái ứng dụng, rất hữu ích cho thiết kế giao diện.

2.3.2 Dart Web

Dart Web cho phép phát triển ứng dụng web bằng cách biên dịch mã Dart thành JavaScript, cho phép chạy trên bất kỳ trình duyệt web hiện đại nào. Điều này giúp xây dựng ứng dụng web phức tạp với hiệu suất cao, tận dụng các tính năng như hệ thống kiểu dữ liệu tĩnh và lập trình bắt đồng bộ.

Từ phiên bản 3.4, Dart Web còn hỗ trợ biên dịch sang WebAssembly, tăng hiệu suất thêm 30-50% so với JavaScript, đặc biệt cho ứng dụng web không dùng Flutter.

2.4 Công cụ phát triển ứng dụng trên Dart

Dart SDK cung cấp một bộ công cụ mạnh mẽ để hỗ trợ phát triển ứng dụng trên nhiều nền tảng. Dưới đây là các công cụ chính và cách sử dụng của chúng:

2.4.1 Công cụ chính của Dart SDK

- **dart:** Giao diện dòng lệnh để tạo, chạy, phân tích, kiểm tra, biên dịch và đóng gói Dart.
 - **dart run:** Chạy các chương trình Dart.

- **dart analyze:** Phân tích mã nguồn để tìm lỗi và cảnh báo.
- **dart test:** Chạy các bài kiểm tra đơn vị.
- **dart compile:** Biên dịch Dart thành các định dạng khác nhau:
 - * **exe:** Tạo ra một tệp thực thi độc lập cho Windows, macOS, hoặc Linux, bao gồm mã đã biên dịch và một runtime Dart nhỏ.
 - * **aot-snapshot:** Tạo ra một module AOT đã biên dịch, phụ thuộc vào kiến trúc phần cứng, không bao gồm runtime Dart.
 - * **jit-snapshot:** Tạo ra một module JIT với các lớp đã phân tích và mã đã biên dịch từ một lần chạy huấn luyện, giúp khởi động nhanh hơn.
 - * **kernel:** Đóng gói ứng dụng vào một tệp portable chứa Kernel AST, có thể chạy trên tất cả các hệ điều hành và kiến trúc CPU.
 - * **js:** Biên dịch Dart thành JavaScript có thể triển khai, với các mức tối ưu hóa từ -O0 đến -O4.
 - * **wasm:** Biên dịch thành WebAssembly (đang trong giai đoạn phát triển).
- **dartdoc:** Trình tạo tài liệu, giúp tạo ra tài liệu tham khảo HTML từ mã nguồn Dart.
 - Sử dụng: Chạy lệnh `dart doc .` trong thư mục gốc của gói sau khi chạy `dart pub get`.
 - Kết quả: Tài liệu được đặt trong thư mục `doc/api` (có thể cấu hình).
- **pub:** Trình quản lý gói, cho phép quản lý các gói phụ thuộc cho dự án Dart.
 - **pub get:** Lấy các gói phụ thuộc được chỉ định trong tệp `pubspec.yaml`.
 - **pub upgrade:** Nâng cấp các gói phụ thuộc lên phiên bản mới nhất phù hợp với ràng buộc trong `pubspec.yaml`.
 - **pub publish:** Đăng gói lên `pub.dev` để người khác có thể sử dụng.

- **pub global activate:** Làm cho các công cụ dòng lệnh của gói có sẵn toàn cục.

2.4.2 Công cụ cho phát triển web

Để phát triển các ứng dụng trên nền web, Dart cung cấp các công cụ sau:

- **dart2js:** Biên dịch Dart thành JavaScript, cho phép chạy ứng dụng Dart trên trình duyệt web.
 - Sử dụng: Chạy lệnh `dart compile js` (thay thế cho `dart2js` trong các phiên bản cũ).
 - Tùy chọn: Có thể tối ưu hóa với các cờ như `-O0` đến `-O4`.
- **webdev:** Công cụ dòng lệnh để biên dịch và kiểm tra ứng dụng web Dart.
 - **serve:** Khởi động máy chủ phát triển để gõ lỗi với các cập nhật tăng cường (hot reload).
 - **build:** Tạo ra mã JavaScript đã tối ưu hóa cho sản xuất.
- **dartdevc:** Biên dịch phát triển Dart thành JavaScript (cho môi trường phát triển), hỗ trợ biên dịch modular nhanh chóng cho các trình duyệt hiện đại.
 - Sử dụng: Tự động được sử dụng khi chạy `webdev serve` trong chế độ phát triển.

Các công cụ trên không chỉ hỗ trợ phát triển ứng dụng Dart mà còn giúp tối ưu hóa quy trình làm việc, từ việc quản lý phụ thuộc đến biên dịch và triển khai. Đặc biệt, với sự kết hợp của Flutter, Dart đã trở thành một ngôn ngữ mạnh mẽ cho phát triển ứng dụng đa nền tảng. Các công cụ như `webdev` và `dartdevc` giúp phát triển web nhanh chóng và hiệu quả, trong khi `dart compile` và `pub` đảm bảo tính linh hoạt và khả năng mở rộng của các dự án.

2.5 Chương trình minh họa

Dưới đây là một ví dụ mẫu về một chương trình minh họa sử dụng ngôn ngữ lập trình Dart.

Ví dụ:

```
// First Programming
void main() {
    var a = 'World';
    print('Hello $a');
}
```

- **Hàm main ()**: Đây là điểm bắt đầu của mọi chương trình Dart. Khi chương trình được thực thi, Dart runtime sẽ tìm và chạy hàm main () đầu tiên. Nếu không có hàm main (), chương trình sẽ không thể chạy được và sẽ báo lỗi.
- **Từ khóa void**: Từ khóa này chỉ ra rằng hàm main không trả về giá trị nào. Dart yêu cầu rõ ràng về kiểu dữ liệu của hàm để đảm bảo tính an toàn kiểu (type safety).
- **Câu lệnh var a = 'World';**
 - Từ khóa var cho phép khai báo biến mà không cần chỉ định kiểu dữ liệu tường minh; Dart sẽ tự động suy luận kiểu dựa trên giá trị được gán.
 - Trong ví dụ trên, a sẽ được xác định là kiểu String vì nó được gán giá trị chuỗi 'World'.
- **Câu lệnh print ('Hello \$a');**
 - Hàm print () được sử dụng để in dữ liệu ra màn hình console.
 - Cú pháp \$a trong chuỗi là cách nội suy biến (string interpolation) trong Dart, cho phép chèn giá trị biến a trực tiếp vào chuỗi mà không cần phép nối thủ công.

Thực thi chương trình:

Để chạy chương trình Dart đơn giản này, ta thực hiện các bước sau:

1. Lưu đoạn mã vào một file với phần mở rộng .dart, ví dụ:
hello_world.dart.
2. Mở terminal hoặc command prompt, chuyển đến thư mục chứa file đó.
3. Gõ lệnh:

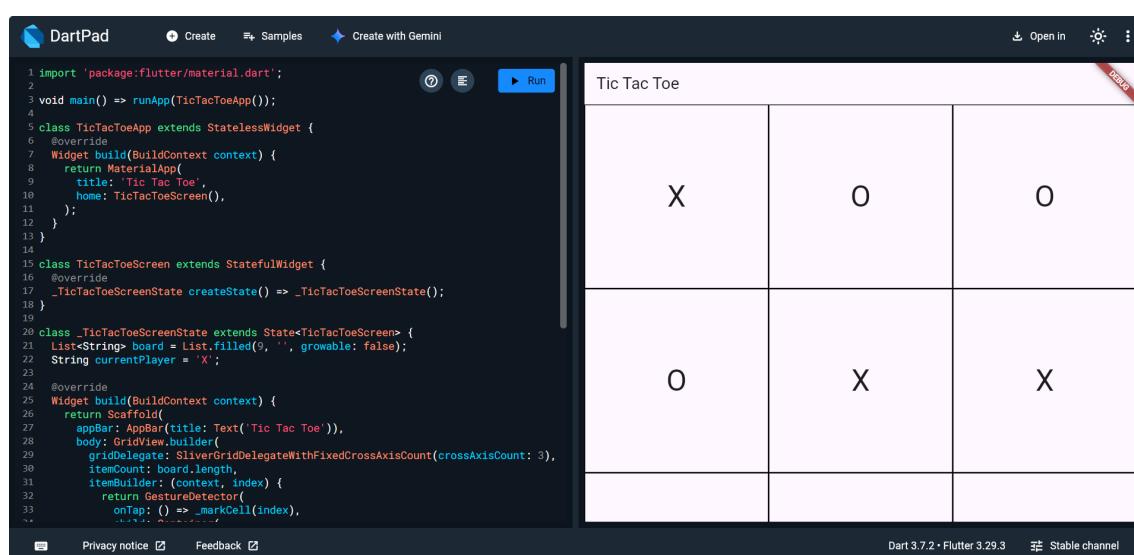
```
dart hello_world.dart
```

4. Nếu mọi thứ đúng, ở terminal sẽ hiển thị kết quả:

```
Hello World
```

Giới thiệu công cụ DartPad:

Dưới đây là hình ảnh của công cụ DartPad trong thực tế với 1 Project minh họa nhỏ về game TicTacToe:



Hình 2.6: Công cụ Dartpad

DartPad là một công cụ cực kỳ hữu ích do Google cung cấp, cho phép người học và lập trình viên thử nghiệm và thực thi mã Dart ngay trên trình duyệt mà không cần cài đặt bất kỳ phần mềm nào.

PHẦN 2. GIỚI THIỆU VỀ NGÔN NGỮ LẬP TRÌNH DART

- Truy cập DartPad tại địa chỉ: <https://dartpad.dev/>
- DartPad hỗ trợ đầy đủ cú pháp Dart, tính năng kiểm tra lỗi thời gian thực, và thậm chí còn hỗ trợ lập trình Flutter để phát triển giao diện người dùng ngay trong trình duyệt.
- Người dùng có thể:
 - Viết mã Dart nhanh chóng và dễ dàng.
 - Thực thi chương trình và xem kết quả ngay lập tức.
 - Chia sẻ đoạn mã của mình qua URL ngắn gọn.
- DartPad rất hữu ích cho:
 - Sinh viên mới học lập trình.
 - Người dạy học hoặc thuyết trình giới thiệu về Dart/Flutter.
 - Lập trình viên muốn kiểm tra nhanh một đoạn mã nhỏ.

PHẦN 3. CÚ PHÁP CƠ BẢN CỦA NGÔN NGỮ DART

3.1 Các quy tắc chung trong ngôn ngữ lập trình Dart

Dart hỗ trợ hai loại biên dịch đó là: Just in time (JIT) và Ahead of time (AoT). Cú pháp của nó về cơ bản là sự pha trộn của CPP, Python, Java và JavaScript. Cú pháp là một chương trình Dart cơ bản bao gồm nhiều yếu tố khác nhau như từ khóa, mã định danh, hằng số, ký tự chuỗi, kiểu dữ liệu và ký hiệu.

Hàm `main` – điểm bắt đầu của chương trình:

Hàm `main` trong Dart có thể được định nghĩa **không có tham số** hoặc **có tham số** truyền vào. Cụ thể, nếu chương trình không cần nhận tham số từ bên ngoài, ta định nghĩa hàm `main` đơn giản như sau:

```
void main() {  
    // Code here ...  
}
```

Ngược lại, nếu muốn truyền tham số (ví dụ các đối số dòng lệnh) cho chương trình, ta có thể khai báo `main` với một danh sách tham số kiểu chuỗi:

```
void main(List<String> args) {  
    // ...  
}
```

Ở ví dụ trên, `List<String> args` là tham số tùy chọn, chứa danh sách các chuỗi nhập vào chương trình (ví dụ khi chạy ứng dụng từ dòng lệnh). Kiểu trả về của `main` luôn là `void` (hàm không trả về giá trị).

Như vậy, Dart cho phép linh hoạt trong khai báo hàm `main`: có thể có hoặc không có tham số. Tuy nhiên, trong cả hai trường hợp, hàm `main` chỉ được gọi một lần khi chương trình khởi động và kết thúc khi toàn bộ chương trình kết thúc. Người lập trình cần phải đảm bảo mọi logic khởi chạy ứng dụng đều được gọi trực

tiếp hoặc gián tiếp từ hàm main này,

Phân biệt chữ hoa và chữ thường trong tên:

Dart là ngôn ngữ phân biệt chữ hoa và chữ thường (case-sensitive).

Các định danh (tên biến, hàm, lớp,...) được viết khác nhau về chữ hoa/chữ thường sẽ được hiểu là những định danh hoàn toàn độc lập.

Ví dụ, MyVariable và myVariable được coi là hai tên khác nhau trong Dart.

Do đó, người lập trình cần phải thống nhất trong cách đặt tên biến, tránh gây nhầm lẫn giữa các biến, hàm,...

Kết thúc lệnh bằng dấu chấm phẩy:

Tương tự nhiều ngôn ngữ C-like khác, mỗi câu lệnh trong Dart phải kết thúc bằng dấu chấm phẩy (;). Dấu chấm phẩy đóng vai trò phân tách các lệnh liên tiếp trong mã nguồn.

Khi trình biên dịch Dart đọc mã, dấu ; cho biết đây là điểm kết thúc của một lệnh. Nếu thiếu dấu chấm phẩy ở cuối một lệnh, mã nguồn sẽ không hợp lệ và gây lỗi cú pháp.

Ví dụ, giả sử ta viết hai lệnh nối tiếp mà quên dấu ; ở giữa:

```
print('Hello')  
print('World')
```

Ở tình huống này, trình biên dịch sẽ không thể hiểu được ranh giới giữa hai lệnh print và sẽ báo lỗi. Vì vậy cần phải sửa bằng cách thêm ; để ngăn cách giữa các lệnh:

```
print('Hello');  
print('World');
```

Khối lệnh và cặp dấu ngoặc nhọn:

Một nhóm các lệnh liên tiếp được nhóm lại với nhau tạo thành một khối lệnh (block). Trong Dart cũng như trong nhiều ngôn ngữ lập trình khác, khối lệnh được bao bọc bởi cặp dấu ngoặc nhọn mở và đóng. Mục đích của khối lệnh là để gộp nhiều câu lệnh thành một đơn vị logic, chẳng hạn nội dung của một hàm, hoặc thân của một câu lệnh điều kiện `if/else`, hoặc thân vòng lặp `for/while`.

Mỗi khi bắt đầu một khối lệnh mới, ta dùng dấu `{` và khi kết thúc khối đó, ta dùng dấu `}`. Các khối lệnh có thể chồng nhau: bên trong một khối lệnh có thể chứa thêm các khối lệnh con (ví dụ: bên trong hàm `main` có khối lệnh của một vòng lặp, trong vòng lặp lại có khối lệnh của một cấu trúc điều kiện, v.v.).

Việc thụt lề (indentation) thường được sử dụng khi viết mã để cho thấy trực quan các khối lệnh nằm trong nhau, giúp người đọc dễ theo dõi cấu trúc chương trình.

```
if (a > 1) {  
    // Nested code blocks are allowed  
    if (...) {  
        // Block 1  
    } else {  
        // Block 2  
    }  
} else {  
    // Alternative block  
}
```

Trong thực tế, nếu khối lệnh chỉ có duy nhất một lệnh bên trong, Dart cho phép người lập trình bỏ cặp ngoặc nhọn và viết trực tiếp lệnh đó sau cấu trúc điều khiển.

```
if (x > 0) print('x is positive');
```

Dart khuyến khích để mã nguồn rõ ràng và tránh các lỗi logic do nhầm lẫn phạm vi của khối lệnh.

Chú thích trong mã nguồn:

Chú thích là phần nội dung trong mã nguồn mà trình biên dịch sẽ **bỏ qua không**

thực thi, thường được dùng để giải thích code hoặc tạm thời vô hiệu hóa một đoạn mã. Dart hỗ trợ cả chú thích một dòng và chú thích nhiều dòng, tương tự như ngôn ngữ C/C++.

- **Chú thích trên một dòng:**

Bắt đầu bằng ký tự `//`. Tất cả nội dung sau `//` trên cùng một dòng sẽ được coi là chú thích và bị bỏ qua khi chạy chương trình.

Ví dụ:

```
// This line is ignored (a comment)
print('Hello'); // Prints 'Hello'
```

Ở ví dụ trên, dòng đầu tiên hoàn toàn là chú thích, còn ở dòng thứ hai, phần từ `//` trở đi ("`// Prints 'Hello'`") cũng là chú thích (giải thích cho lệnh `print`)

- **Chú thích trên nhiều dòng:**

Bắt đầu bằng `/*` và kết thúc bằng `*/`. Mọi nội dung nằm giữa cặp `/*` và `*/` sẽ được coi là chú thích. Kiểu chú thích này cho phép ghi chú trải trên nhiều dòng.

Ví dụ:

```
/*
Calculate the area of a circle.
Radius r = 5.
*/
double area = 3.14 * 5 * 5;
```

Tất cả những dòng văn bản giữa `/*` và `*/` ở trên đều là chú thích và sẽ bị bỏ qua khi chương trình chạy. Ngoài ra, Dart cho phép lồng các chú thích nhiều dòng bên trong nhau.

Ngoài hai loại chú thích trên, Dart còn hỗ trợ **chú thích tài liệu** (documentation comments). Đây là các chú thích đặc biệt dùng để tự động sinh tài liệu API cho

code, bắt đầu bằng `///` (ba dấu gạch chéo) cho mỗi dòng chú thích đơn, hoặc cặp `/** ... */` cho chú thích nhiều dòng dạng tài liệu.

Ví dụ:

```
/// Calculate factorial of positive integer n.  
/// return n! (1 * 2 * ... * n).  
int factorial(int n) { ... }
```

Các chú thích bắt đầu bằng `///` như trên sẽ được các công cụ tạo tài liệu (vd. dartdoc) hiểu và trích xuất thành phần mô tả cho hàm `factorial`. Mặc dù chú thích tài liệu được đặt trong mã nguồn, chúng không ảnh hưởng đến việc biên dịch chương trình, tương tự như các chú thích thông thường.

Quy tắc định danh:

Mọi định danh (tên biến, tên hàm, tên lớp, v.v.) trong Dart phải tuân thủ quy tắc đặt tên giống như nhiều ngôn ngữ lập trình khác. Cụ thể, định danh phải bắt đầu bằng một ký tự chữ cái (a–z hoặc A–Z) hoặc dấu gạch dưới `_`, và các ký tự tiếp theo có thể là chữ cái hoặc chữ số. Định danh không được phép bắt đầu bằng chữ số, không được chứa khoảng trắng hay các ký tự đặc biệt (ngoại trừ ký tự gạch dưới, và cả ký hiệu `$` cũng được cho phép sử dụng trong định danh). Ngoài ra, không được dùng các từ khóa dành riêng của Dart làm tên định danh.

Ví dụ:

- `MyClass, _result123` là các **định danh hợp lệ**.
- `123number` (bắt đầu bằng chữ số) và `username` (chứa ký tự không hợp lệ là dấu `(-)`) là những **định danh không hợp lệ**.

Về quy ước đặt tên nhằm viết mã nhất quán và dễ đọc, Dart tuân thủ phong cách *camelCase/PascalCase*: Tên biến, hàm, và thuộc tính thường được viết dưới dạng *camelCase* (chữ cái đầu tiên viết thường, các chữ cái đầu của những từ tiếp theo viết hoa), còn tên lớp, tên enum thì viết dưới dạng *PascalCase* (viết hoa chữ cái

dầu của mỗi từ) theo khuyến nghị của hướng dẫn Effective Dart.

Ví dụ:

- **DataRepository** phù hợp khi đặt tên lớp vì nó tuân theo PascalCase.
- **isValid, calculateSum** phù hợp khi đặt tên biến vì tuân theo camelCase.

Việc tuân theo các quy tắc và quy ước định danh này giúp mã Dart rõ ràng, dễ hiểu và tránh được lỗi khi biên dịch.

Mọi thứ trong Dart là đối tượng:

Dart là ngôn ngữ hướng đối tượng thuần túy, nghĩa là **mọi thứ** có thể gán cho biến đều là đối tượng. Số nguyên, số thực, chuỗi, giá trị boolean, thậm chí cả hàm cũng đều được coi là đối tượng (thể hiện bởi các lớp như int, double, String, bool, Function, ...). Đặc biệt, ngay cả giá trị **null** cũng được biểu diễn như một đối tượng thuộc kiểu Null trong Dart (trước khi có null-safety, null là một đối tượng hợp lệ).

Mỗi đối tượng trong Dart là một thể hiện của một lớp nào đó, và tất cả các lớp (kể cả các lớp do người dùng định nghĩa) đều ngầm định kế thừa từ một lớp cơ sở chung có tên là Object. Object là gốc của mọi lớp trong hệ thống loại của Dart. Mọi đối tượng Dart đều chia sẻ một số phương thức cơ bản từ Object:

- **hashCode** – Thuộc tính trả về mã băm (hash code) của đối tượng

Thuộc tính hashCode trả về một số nguyên đại diện cho trạng thái của đối tượng, được sử dụng trong các cấu trúc dữ liệu dựa trên băm như Set và Map.

Mã băm mặc định được triển khai trong lớp Object chỉ phản ánh danh tính (identity) của đối tượng.

Khi ghi đè toán tử == để so sánh các đối tượng dựa trên trạng thái (tức là các thuộc tính), cần ghi đè thuộc tính hashCode để đảm bảo rằng hai đối tượng được coi là bằng nhau sẽ có cùng mã băm.

- **toString()** – Phương thức toString()

Phương thức `toString()` trả về một chuỗi mô tả đối tượng, thường được sử dụng để hiển thị thông tin trong quá trình gõ lỗi hoặc ghi log. Mặc định, `toString()` trả về chuỗi dạng Instance of 'ClassName', điều này không cung cấp nhiều thông tin hữu ích. Do đó, người ta thường ghi đè phương thức này để trả về chuỗi mô tả chi tiết hơn về đối tượng.

Ví dụ:

```
class Person {  
    final String name;  
    final int age;  
  
    Person(this.name, this.age);  
  
    @override String toString()  
    => 'Person(name: $name, age: $age)';  
}
```

Với cách ghi đè này, khi in một đối tượng Person ta sẽ nhận được kết quả rõ ràng, tường minh: Person(name: Alice, age: 30)

- **Toán tử == – So sánh đối tượng**

Toán tử `==` được sử dụng để so sánh hai đối tượng. Mặc định, Dart so sánh hai đối tượng dựa trên danh tính, tức là chúng phải là cùng một đối tượng trong bộ nhớ để được coi là bằng nhau. Hai đối tượng nếu có giá trị giống nhau nhưng không cùng danh tính thì khi sử dụng toán tử `==` trả về kết quả `False`.

Để so sánh các đối tượng dựa trên giá trị, cần ghi đè toán tử `==` và thuộc tính `hashCode`

Từ khóa public/private/protected:

Dart không sử dụng các từ khóa như `public`, `private` hay `protected` để quy định phạm vi truy cập như trong một số ngôn ngữ hướng đối tượng khác (ví dụ Java, C++). Mặc định, mọi lớp, thuộc tính và phương thức được khai báo trong

Dart đều có phạm vi truy cập “công khai” đối với các đoạn mã khác, trừ khi được định nghĩa đặc biệt để giới hạn phạm vi. Nếu không làm gì thêm, các thành phần định nghĩa có thể được truy cập từ bất kỳ đâu.

Thiết kế của Dart đơn giản hóa việc đóng gói khi cho rằng thành phần nào không được đánh dấu riêng tư thì sẽ được hiểu là công khai. Điều này giúp loại bỏ sự rườm rà khi phải chỉ định `public` cho từng thành phần, đồng thời khuyến khích lập trình viên suy nghĩ về việc tổ chức mã theo thư viện (library) để kiểm soát truy cập thay vì dựa vào từ khóa.

Cách khai báo private bằng dấu _:

Thay vì dùng từ khóa, Dart quy định phạm vi riêng tư (`private`) ở cấp thư viện thông qua quy ước đặt tên: Nếu tên định danh bắt đầu bằng dấu gạch dưới `_` thì định danh đó được xem là `private` trong library. Các thành viên (thuộc tính, phương thức) hoặc các khai báo (biến, hàm, lớp) có tên bắt đầu bằng `_` chỉ có thể được truy cập ở trong cùng một thư viện (thông thường là cùng một file `.dart`). Ngược lại, nếu tên không bắt đầu bằng `_` thì thành phần đó là công khai và có thể được truy cập từ bất cứ đâu sau khi import.

Ví dụ: Giả sử có lớp Person được định nghĩa như sau:

```
class Person {  
    String _name; // private property  
    int age;      // public property  
  
    Person(this._name, this.age);  
}
```

Trong ví dụ trên:

- Thuộc tính `_name` là một thuộc tính `private` của lớp Person. Điều này đồng nghĩa `_name` chỉ có thể được truy cập hoặc sửa đổi bởi mã nằm trong cùng library định nghĩa lớp Person (ví dụ bên trong chính file khai báo Person hoặc

các file khác dùng part of chung thư viện), và không thể truy cập trực tiếp từ một library khác.

- Ngược lại, thuộc tính `age` không có dấu gạch dưới đầu tên nên được coi là công khai – code từ module khác (sau khi import thư viện chứa Person) có thể đọc/ghi `age` tùy ý.

Quy tắc dùng dấu gạch dưới để đánh dấu private giúp mã Dart ngắn gọn hơn (vì không cần từ khóa) nhưng vẫn đảm bảo đóng gói thông tin: mọi chi tiết triển khai nội bộ mà lập trình viên muốn giấu đều chỉ cần đặt tên bắt đầu bằng `_` là đủ.

3.2 Biến, hằng, kiểu dữ liệu

Trong ngôn ngữ lập trình Dart, mỗi giá trị đều có một **kiểu dữ liệu** xác định, và tất cả những giá trị có thể gán vào biến đều là **đối tượng** (kể cả các giá trị cơ bản như số, chuỗi, hay `null`). Các kiểu dữ liệu cơ bản có sẵn trong Dart bao gồm: **int** (số nguyên, 64-bit), **double** (số thực dấu phẩy động kép, 64-bit), **String** (chuỗi ký tự Unicode), **bool** (kiểu luận lý, với hai giá trị `true/false`), v.v. Ngoài ra còn có các kiểu tổng quát như **Object** (kiểu gốc của mọi đối tượng) và **dynamic** (kiểu động).

3.2.1 Khai báo biến

Dart là ngôn ngữ kiểm tra kiểu tĩnh (*statically-typed*), tức là mỗi biến trong chương trình đều mang một kiểu dữ liệu cố định ngay từ khi khai báo. Tuy nhiên, Dart hỗ trợ *suy luận kiểu* nên ta không nhất thiết phải chỉ rõ kiểu dữ liệu khi khai báo biến. Cụ thể, để khai báo một biến, ta có thể dùng từ khóa **var** theo cú pháp:

```
var <variable_name> = <value_expression>;
```

Khi sử dụng `var`, trình biên dịch Dart sẽ tự động suy ra kiểu dữ liệu của biến dựa trên giá trị được gán ban đầu. Ví dụ:

```
var name = 'Alice'; // Dart infers this as String  
var age = 25; // Dart infers this as int
```

Trong ví dụ trên, biến `a` được gán một chuỗi ký tự, nên Dart hiểu `a` thuộc kiểu `String`. Sau khi khai báo, `a` chỉ có thể nhận các giá trị kiểu `String`; nếu cố gán một kiểu khác (ví dụ số nguyên) cho `a` thì chương trình sẽ báo lỗi kiểu (*type error*) khi biên dịch.

Dart cũng cho phép khai báo biến với kiểu dữ liệu tường minh thay vì sử dụng `var`. Người lập trình có thể chỉ định trực tiếp kiểu dữ liệu trước tên biến. Ví dụ:

```
String s = "String"; // Variable s of type String
int i = 5; // Variable i of type integer
double d = 1.1234; // Variable d of type double
bool b = true; // Variable b of type boolean
```

Việc khai báo rõ kiểu dữ liệu như trên giúp mã nguồn rõ ràng về mục đích sử dụng biến, và trình biên dịch sẽ kiểm tra để đảm bảo biến chỉ được gán các giá trị thuộc đúng kiểu đó. Ngược lại, sử dụng `var` đem lại sự tiện lợi và ngắn gọn, đặc biệt khi kiểu dữ liệu có thể dễ dàng suy ra từ biểu thức gán. Trong thực tế, ta có thể chọn cách khai báo tùy tình huống, nhưng cần nhất quán và dễ đọc.

Trong trường hợp cần một biến có thể chấp nhận nhiều loại giá trị khác nhau, Dart cung cấp kiểu đặc biệt **dynamic**. Khi một biến được khai báo là `dynamic`, nghĩa là quá trình kiểm tra kiểu tại biên dịch sẽ được nới lỏng: biến đó có thể nhận bất kỳ kiểu dữ liệu nào trong lúc chạy, và mọi kiểm tra về tính hợp lệ kiểu sẽ dời đến thời gian chạy (*runtime*).

Ví dụ:

```
dynamic dyn = 123; // Initialize dyn as an integer
dyn = "Dynamic"; // dyn is reassigned to a string
dyn = 1.12345; // dyn is reassigned to a double
```

Ở ví dụ trên, cùng một biến `dyn` lần lượt được gán số nguyên, chuỗi và số thực mà không gây lỗi biên dịch. Tuy nhiên, việc sử dụng `dynamic` cũng tiềm ẩn rủi ro: vì trình biên dịch không kiểm tra chặt chẽ kiểu, nên các lỗi do gán sai kiểu chỉ được

phát hiện khi chạy chương trình, có thể dẫn đến ngoại lệ nếu ta gọi một phương thức không phù hợp với kiểu thực sự của giá trị. Do đó, **chỉ nên dùng dynamic khi thật sự cần thiết**, trong các tình huống linh hoạt đặc biệt; còn nhìn chung, với các biến có kiểu xác định, nên dùng `var` (kết hợp suy luận kiểu) hoặc khai báo kiểu tường minh để tận dụng kiểm tra lỗi ngay từ lúc biên dịch.

3.2.2 Khai báo hằng

Bên cạnh biến thay đổi được, Dart cho phép khai báo *hằng* – tức biến chỉ đọc, không thể thay đổi giá trị sau khi đã được khởi tạo. Để khai báo *hằng*, ta sử dụng từ khóa **final** hoặc **const** (có thể dùng thay cho `var`, và cũng có thể kèm kiểu dữ liệu hoặc không).

Ví dụ:

```
const constantName = value_expression;  
final name1 = value_expression;  
// You can also specify the type explicitly  
final String name2 = value_expression;
```

Cả `final` và `const` đều tạo ra một biến chỉ được gán một lần duy nhất. Sự khác biệt chính giữa chúng là thời điểm và cách thức giá trị của biến được xác định:

- **final** – Biến được khai báo với `final` là một *hằng chỉ đọc* tại *thời gian chạy*. Điều này có nghĩa là ta có thể gán giá trị cho nó một lần duy nhất, nhưng giá trị đó có thể được tính toán hoặc lấy từ một biểu thức khi chương trình chạy. Khi đã gán rồi, biến `final` không thể thay đổi (mọi cố gắng gán lần thứ hai sẽ gây lỗi). Ví dụ, ta có thể viết `final x = DateTime.now();` để lấy thời điểm hiện tại làm giá trị, và `x` sẽ giữ nguyên giá trị đó suốt thời gian chạy chương trình.
- **const** – Biến khai báo với `const` là một *hằng số tại thời điểm biên dịch* (compile-time constant). Điều này đòi hỏi giá trị gán cho `const` phải được biết chắc chắn và cố định ngay trong mã nguồn (ví dụ: các literal hoặc biểu

thức hằng). Nói cách khác, **một biến const bắt buộc phải nhận một biểu thức có thể đánh giá tại thời điểm biên dịch**. Các biến const cũng mang tính chất *bất biến* giống final (không thể thay đổi sau khi gán), và trên thực tế biến const cũng được xem là final ngầm định. Tuy nhiên, vì yêu cầu giá trị cố định sẵn, const thường được dùng cho các hằng số đơn giản hoặc để tạo ra các đối tượng bất biến tại compile-time.

Dưới đây là ví dụ minh họa cách sử dụng const và final:

```
// Constant, value = 120 (determined at compile time)
const int minutes = 2 * 60;

// Get a random number (0..499)
var randomNumber = Random().nextInt(500);

// Final constant, value determined at runtime
final result = randomNumber * 2;

// Cannot use const because the expression is not a
// compile-time constant
const wrongResult = randomNumber * 2;
```

Giải thích: Trong đoạn mã trên minutes là hằng số tính sẵn ($2 * 60$ được tính toán ngay khi biên dịch, cho ra 120); ngược lại, result được khai báo final và nhận giá trị là kết quả phép tính với một số ngẫu nhiên sinh ra lúc chạy chương trình. Ta **không thể** dùng const trong trường hợp giá trị gán không phải hằng số xác định sẵn(dòng lệnh cuối).

3.2.3 Các kiểu dữ liệu

Dart hỗ trợ các kiểu dữ liệu cơ bản như **bool, int, double, String, List, Set** và **Map**. Mỗi kiểu dữ liệu biểu diễn một loại thông tin khác nhau, có miền giá trị và đặc điểm riêng.

Kiểu dữ liệu	Mô tả
bool	Kiểu logic (Boolean), chỉ có hai giá trị: <code>true</code> (đúng) hoặc <code>false</code> (sai).
int	Kiểu số nguyên (integer), biểu diễn các số nguyên (có dấu) trong khoảng 64-bit.
double	Kiểu số thực dấu phẩy động (double precision), độ chính xác kép 64-bit, biểu diễn số thực (số có phần thập phân).
String	Kiểu chuỗi ký tự (string), biểu diễn một dãy các ký tự Unicode (UTF-16). Khai báo bằng cặp dấu nháy đơn hoặc nháy kép.
List (Danh sách)	Cấu trúc danh sách (list) các phần tử, tương tự mảng một chiều. Các phần tử có thứ tự tuyến tính và được truy cập bằng chỉ số (index).
Set (Tập hợp)	Cấu trúc tập hợp (set) các phần tử không trùng lặp. Không duy trì thứ tự cố định của phần tử, dùng để lưu các giá trị duy nhất.
Map (Ánh xạ)	Cấu trúc ánh xạ (map) gồm các cặp khóa - giá trị (key - value). Mỗi khóa ánh xạ tối đa một giá trị, cho phép tra cứu giá trị thông qua khóa.

Bảng 3.1: Mô tả các kiểu dữ liệu cơ bản trong Dart

a, Bool

Dart cung cấp hỗ trợ sẵn có cho kiểu dữ liệu Boolean. Kiểu dữ liệu Boolean trong DART chỉ hỗ trợ hai giá trị `true` và `false`. Từ khóa `bool` được sử dụng để biểu diễn một ký tự Boolean trong DART.

Ví dụ:

```
void main() {
  bool test;
  test = 12 > 5;
  print(test);
}
```

Trong ví dụ trên, `test` là biến `bool`. Biểu thức so sánh `12 > 5` sẽ quyết định giá trị của biến `test`. Câu lệnh `print (test)` sẽ trả ra kết quả `true`.

b, Int

int (viết tắt của `integer`) là kiểu dữ liệu số nguyên trong Dart, dùng để biểu diễn các số nguyên (không có phần thập phân). Miền giá trị của `int` trong Dart rất lớn:

PHẦN 3. CÚ PHÁP CƠ BẢN CỦA NGÔN NGỮ DART

trên các nền tảng thông thường, `int` là số nguyên 64-bit có dấu, giá trị nhỏ nhất khoảng -2^{63} và lớn nhất khoảng $2^{63}-1$ (tức khoảng từ -9.22×10^{18} đến 9.22×10^{18}).

Cú pháp khai báo một số nguyên kiểu `int`:

```
int var_name // Declare an integer
```

Ví dụ:

```
void main() {  
    // declare an integer  
    int num1 = 10;  
    // print value  
    print(num1);  
}
```

Khi chạy chương trình, kết quả in ra là **10**

c, Double

double là kiểu dữ liệu số thực dấu phẩy động trong Dart, dùng để biểu diễn các số có phần thập phân (số thực). Kiểu `double` chiếm 64-bit và tuân theo tiêu chuẩn IEEE 754 (độ chính xác kép). Cú pháp khai báo một số thực kiểu `double`:

```
double var_name // Declare a double value
```

Ví dụ:

```
void main() {  
    // Declare a double value  
    double num1 = 12.34;  
    // print value:  
    print(num1);  
}
```

Khi chạy chương trình, kết quả in ra là **12.34**.

d, String

String là kiểu dữ liệu chuỗi ký tự trong Dart, dùng để biểu diễn văn bản (text). Mỗi chuỗi là một dãy các ký tự thuộc bảng mã Unicode (mỗi ký tự được mã hóa dưới dạng một hoặc hai byte UTF-16). Kiểu **String** trong Dart là một lớp đối tượng (thuộc `dart:core`), và giống như nhiều ngôn ngữ khác, chuỗi trong Dart là **bất biến** (immutable), tức là giá trị chuỗi không thể thay đổi sau khi được tạo ra. Mọi thao tác "thay đổi" trên chuỗi thực chất sẽ tạo ra một chuỗi mới.

Để tạo một chuỗi ký tự, ta đặt nội dung chuỗi trong cặp dấu nháy đơn ('...') hoặc nháy kép ("..."). Dart cho phép sử dụng linh hoạt cả hai loại dấu nháy để thuận tiện khi chuỗi bản thân có chứa dấu nháy.

Một số thuộc tính và phương thức thông dụng của **String** bao gồm:

- `.length`: Trả về độ dài chuỗi
- `.toUpperCase()` / `toLowerCase()`: Dùng để chuyển chuỗi thành chữ hoa / chữ thường
- `.contains()`: Dùng để kiểm tra chuỗi có chứa một chuỗi con
- `.substring(start, [end])`: Dùng để trích xuất chuỗi con từ chuỗi cho trước, bắt đầu từ vị trí `start` đến hết vị trí `end - 1`. Nếu không có `[end]` thì mặc định cắt đến hết xâu.
- `.compareTo(str)`: So sánh chuỗi theo thứ tự từ điển, trả về -1/0/1 tương ứng với nhỏ hơn, bằng, lớn hơn.

Ví dụ:

```
var s1 = "abcdef";
print (s1.length);
// Output: 6

var s2 = "phuc";
print(s2.toUpperCase());
```

```
// Output: PHUC

var s3 = "abcdefghijklm";
print(s3.substring(1, 4));
// Output: abc

var s4 = "helloiamkhanh";
print(s4.contains("khanh"));
// Output: true

var s5 = "abcde";
print(s5.compareTo("bcdef"));
// Output: -1
```

e, List

List là cấu trúc dữ liệu dạng danh sách tuyến tính các phần tử trong Dart, tương tự như mảng (array) một chiều trong các ngôn ngữ khác. Mỗi phần tử trong List đều có một vị trí chỉ số (index) xác định, bắt đầu từ 0. Phần tử đầu tiên của list có chỉ số 0, phần tử thứ hai chỉ số 1, v.v., và phần tử cuối cùng có chỉ số `length - 1` (trong đó `length` là độ dài danh sách). List duy trì thứ tự sắp xếp phần tử theo trình tự thêm vào.

- Nếu khai báo **List** với kiểu dữ liệu cụ thể, thì tất cả các phần tử trong List phải có kiểu dữ liệu đó.

Ví dụ:

```
// Valid
List<int> numbers = [1, 2, 3];

// Error: 'a' is not of type int
List<int> invalidNumbers = [1, 'a', 3];
```

- Nếu khai báo **List<dynamic>** hoặc **không chỉ định kiểu**, Dart cho phép các phần tử có kiểu khác nhau.

Ví dụ:

```
// Allowed because the list is declared as dynamic
List<dynamic> mixedList = [1, 'hello', true, 3.14];
var anotherList = [1, 'abc', 2.5];
```

Có hai loại list: **List cố định độ dài** và **List có thể thay đổi độ dài (growable)**.

Danh sách có chiều dài cố định **không thể** thay đổi số lượng phần tử sau khi được khởi tạo.

Ví dụ List có độ dài cố định:

```
List<String> myList = List<String>(3);
myList[0] = 'one';
myList[1] = 'two';
myList[2] = 'three';
// myList.add('four'); -> Error
```

Danh sách có thể mở rộng **cho phép** thay đổi số lượng phần tử.

Ví dụ List có thể mở rộng:

```
List<int> myList = List<int>();
myList.add(42);
myList.add(2023);
print(myList); // [42, 2023]
print(myList.length); // 2

myList.add(2024);
print(myList); // [42, 2023, 2024]
print(myList.length); // 3
```

Một số thuộc tính và phương thức phổ biến của List:

- `.length`: Trả về số phần tử có trong danh sách
- `.index`: Dùng để truy cập vào phần tử có vị trí `index` trong list
- `.add(value)`: Dùng để thêm giá trị `value` vào cuối danh sách.

- `.removeAt (int index)`: Dùng để xóa phần tử ở vị trí index trong danh sách.
- `.removeLast ()`: Xóa phần tử ở vị trí cuối cùng trong danh sách.
- `.clear ()`: Xóa tất cả các phần tử trong danh sách.

Ví dụ:

```
var arr = [1, 12, 4, 26, 9, 2, 'u', 'i', 'a'];
print(arr.length); // Output: 9
print(arr[0]); // Output: 1

// Add an element to the List
arr.add(10);
print(arr);
// Output: [1, 12, 4, 26, 9, 2, 'u', 'i', 'a', 10]

// Remove an element by value
arr.remove('a');
print(arr);
// Output: [1, 12, 4, 26, 9, 2, 'u', 'i', 10]

// Remove an element by index
arr.removeAt(0);
print(arr);
// Output: [12, 4, 26, 9, 2, 'u', 'i', 10]

// Remove the last element in the List
arr.removeLast();
print(arr);
// Output: [12, 4, 26, 9, 2, 'u', 'i']

// Remove all elements from the List
arr.clear();
print(arr);
// Output: []
```

f, Set

Set (Tập hợp) là một cấu trúc dữ liệu lưu trữ các phần tử không trùng lặp. Khác với List, Set không duy trì thứ tự các phần tử dựa trên vị trí chỉ số; thay vào đó, Set được tổ chức như một tập hợp toán học, mỗi giá trị xuất hiện tối đa một lần và thứ tự duyệt có thể khác với thứ tự thêm.

Trong Set, vì không có chỉ số, ta không truy cập phần tử theo vị trí như list. Thay vào đó, có thể dùng vòng lặp hoặc các phương thức như contains (element) để kiểm tra sự có mặt của phần tử.

Set được khai báo bằng cách sử dụng từ khóa set. Có 2 cách để khai báo:

```
// Method 1: Using var keyword  
var variable_name = <variable_type>{};
```

```
// Method 2: Using explicit Set declaration  
Set<variable_type> variable_name = {};
```

Một số phương thức phổ biến của Set:

- .length: Trả về số lượng phần tử trong set.
- .contains (element_name): Kiểm tra sự tồn tại của element_name trong tập hợp, trả về true nếu tồn tại, trả về false nếu không tồn tại.
- .remove (element_name): Xóa phần tử có giá trị bằng với element_name ra khỏi tập hợp.
- .forEach (...): Duyệt qua tất cả các phần tử có trong set.
- .add (value): Thêm phần tử có giá trị bằng với value vào set, nếu nó chưa tồn tại trong đó.
- .clear (): Xóa hết tất cả các phần tử trong tập hợp.

Ví dụ:

```
var se = {5, 10, 15, 20, 25};
```

```

// Count the number of elements in the set:
print(se.length); // Result: 5

// Check if an element exists in the set:
print(se.contains(15)); // Result: true

// Remove an element with a specific value from the set:
se.remove(15);
print(se); // Result: {5, 10, 20, 25}

// Iterate through all elements in the set:
se.forEach((element) {
    print(element);
});
// Result:
// 5
// 10
// 20
// 25

// Add an element to the set if it doesn't already exist
se.add(30);
print(se); // Result: {5, 10, 20, 25, 30}

// Remove all elements from the set:
se.clear();
print(se); // Result: {}

```

g, Map

Map (Ánh xạ khóa - giá trị) là cấu trúc dữ liệu dạng bảng băm (hash table) trong Dart, cho phép lưu trữ các cặp *key-value* (khóa và giá trị tương ứng). Map tương tự như *dictionary* trong Python: mỗi khóa (key) duy nhất ánh xạ tới một giá trị (value). Dart cung cấp Map<K, V> trong đó K là kiểu khóa, V là kiểu giá trị. Cú pháp để khai báo một Map sử dụng cặp dấu ngoặc nhọn {}, nhưng khác với Set ở chỗ mỗi phần tử là một cặp K: V.

Ví dụ về khai báo một **Map** trong Dart:

```
var gifts = {  
    // Key:      Value  
    'first': 'partridge',  
    'second': 'turtledoves',  
    'fifth': 'golden rings',  
};
```

Một số phương thức phổ biến của Map:

- `.length`: Trả về số lượng phần tử trong Map.
- `.containsKey(key) / containsValue(value)`: Kiểm tra sự tồn tại của key/ value trong Map, trả về true nếu tồn tại, trả về false nếu không tồn tại.
- `.remove(key)`: Xóa cặp key, value theo key.
- `.forEach(fn)`: có thể duyệt qua Map bằng `forEach((k, v) { })`, hoặc duyệt qua `map.keys` (tập hợp các khóa) hay `map.values` (tập hợp các giá trị).
- `[] =`: Thêm một cặp key-value vào Map
- `.clear()`: Xóa hết tất cả các phần tử trong Map.

Ví dụ:

```
var mp = {'a': 1, 'b': 2, 'c': 3};  
  
// Return the number of elements in the Map:  
print(mp.length); // Output: 3  
  
// Check if a key exists in the Map:  
print(mp.containsKey('b')); // Output: true  
  
// Check if a value exists in the Map:  
print(mp.containsValue(2)); // Output: true
```

```
// Remove a key-value pair by key:  
mp.remove('b');  
print(mp); // Output: {'a': 1, 'c': 3}  
  
// Iterate over all key-value pairs in the Map:  
mp.forEach((k, v) {  
    print('Key: $k, Value: $v');  
});  
// Output:  
// Key: a, Value: 1  
// Key: c, Value: 3  
  
// Add a new key-value pair to the Map:  
mp['d'] = 4;  
print(mp); // Output: {'a': 1, 'c': 3, 'd': 4}  
  
// Remove all elements from the Map:  
mp.clear();  
print(mp); // Output: {}
```

3.3 Các toán tử trong Dart

Dart cung cấp một hệ thống toán tử phong phú để hỗ trợ các phép toán cơ bản và xử lý logic. Các toán tử trong Dart có thể được chia thành nhiều nhóm, bao gồm: toán tử số học, toán tử tăng giảm đơn vị, toán tử logic, toán tử so sánh và toán tử gán.

3.3.1 Toán tử số học

Các toán tử số học trong Dart được sử dụng để thực hiện các phép tính toán trên các số nguyên (`int`) hoặc số thực (`double`).

- `+`: Phép cộng.
- `-`: Phép trừ.
- `*`: Phép nhân.

- /: Phép chia (trả về double ngay cả khi hai toán hạng là int).
- %: Phép chia lấy phần dư (modulo).
- ~/: Phép chia lấy phần nguyên (integer division).

Ví dụ:

```
int a = 7;  
int b = 3;  
  
print(a + b);    // 10  
print(a - b);    // 4  
print(a * b);    // 21  
print(a / b);    // 2.333333333333333 (double)  
print(a % b);    // 1 (remainder part)  
print(a ~/ b);   // 2 (integer part)
```

3.3.2 Toán tử tăng giảm đơn vị

Dart hỗ trợ các toán tử tăng và giảm giá trị của biến số nguyên lên 1 đơn vị:

- ++var: Tăng biến lên 1, sau đó trả về giá trị mới.
- var++: Trả về giá trị hiện tại, sau đó tăng biến lên 1.
- --var: Giảm biến xuống 1, sau đó trả về giá trị mới.
- var--: Trả về giá trị hiện tại, sau đó giảm biến xuống 1.

Ví dụ:

```
int x = 5;  
  
print(++x); // 6 (tăng x trước, rồi in ra)  
print(x++); // 6 (in ra x trước, rồi mới tăng thành 7)  
print(x);   // 7  
print(--x); // 6 (giảm x trước, rồi in ra)  
print(x--); // 6 (in ra x trước, rồi giảm thành 5)  
print(x);   // 5
```

3.3.3 Toán tử logic

Dart hỗ trợ các toán tử logic để thực hiện các phép toán luận lý.

- || (hoặc): Kết quả true nếu ít nhất một toán hạng là true.
- && (và): Kết quả true nếu cả hai toán hạng đều true.
- ! (phủ định): Đảo ngược giá trị logic của toán hạng.

Ví dụ:

```
bool a = true;  
bool b = false;  
  
print(a || b); // true  
print(a && b); // false  
print(!a); // false  
print(!b); // true
```

3.3.4 Toán tử so sánh

Toán tử so sánh được dùng để so sánh hai giá trị. Kết quả của các phép so sánh là một giá trị bool (true hoặc false):

- ==: So sánh bằng.
- !=: So sánh khác.
- >: So sánh lớn hơn.
- <: So sánh nhỏ hơn.
- >=: So sánh lớn hơn hoặc bằng.
- <=: So sánh nhỏ hơn hoặc bằng.

Ví dụ:

```
int a = 5;  
int b = 7;
```

```
print(a == b); // false
print(a != b); // true
print(a > b); // false
print(a < b); // true
print(a >= b); // false
print(a <= b); // true
```

3.3.5 Toán tử gán

Gồm = và các toán tử gán rút gọn như : +=, -=, *=, =, ??=.

```
int? a;
a ??= 10; // a is currently null, so assign 10 to a
print(a); // Output: 10

int? b = 5;
b ??= 20; // b already has a value (5), nothing change
print(b); // Output: 5
```

Điểm mới của toán tử ??= là đây là toán tử gán rút gọn, chỉ gán nếu biến hiện tại đang có giá trị null.

3.3.6 Toán tử điều kiện

Dart hỗ trợ hai toán tử điều kiện: toán tử ba ngôi (?:) và toán tử hợp nhất null (??).

Toán tử ba ngôi condition ? expression1 : expression2 sẽ trả về expression1 nếu condition là đúng (true), ngược lại trả về expression2

Ví dụ:

```
var a = 10;
var b = 20;
var maxVal = (a > b) ? a : b;
print(maxVal); // 20
```

Toán tử hợp nhất **null** expression1 ?? expression2 sẽ trả về expression1 nếu nó không null, ngược lại trả về expression2.

Ví dụ:

```
String name;  
var display = name ?? "Unknown";  
print(display); // "Unknown"
```

3.3.7 Phép toán với kiểu String

- Dart cho phép nối chuỗi bằng toán tử +

Ví dụ:

```
String str1 = "abc";  
String str2 = "def";  
String str = str1 + str2;  
print(str); // Output: "abcdef"
```

- So sánh hai chuỗi kí tự bằng toán tử ==

Ví dụ:

```
String str1 = "abc";  
String str2 = "aBc";  
print(str1 == str2); // Output: false  
  
String str3 = "abc";  
print(str1 == str3); // Output: true
```

- Đối với chuỗi kí tự gồm nhiều dòng: Dùng ba dấu nháy đơn hoặc kép (ví dụ '''...''' hoặc """...""") để viết chuỗi gồm nhiều dòng.

Ví dụ:

```
String txt= '''  
Content Line 1  
Content Line 2  
Content Line 3
```

```
''';  
print(txt);  
/* Output:  
Content Line 1  
Content Line 2  
Content Line 3  
*/  
  
String txt2 = """  
    Content Line 2.1  
    Content Line 2.2  
    Content Line 2.3  
""";  
print(txt2);  
/* Output:  
Content Line 2.1  
Content Line 2.2  
Content Line 2.3  
*/
```

- Chèn một biến hoặc một biểu thức vào chuỗi bằng cách ký hiệu \$ten_bien,

\$bieu_thuc_gia_tri

Ví dụ:

```
var a = 10;  
var b = 20;  
String kq = "Sum of $a, $b is ${a + b}$";  
print(kq);  
// Output: Sum of 10, 20 is 30
```

3.3.8 Một số toán tử trên lớp, đối tượng

Toán tử	Ý nghĩa
[]	Truy cập phần tử trong danh sách hoặc map (theo chỉ số hoặc khóa)
.	Truy cập thuộc tính hoặc phương thức của đối tượng
? .	Truy cập thành viên nếu đối tượng không null; nếu đối tượng null thì trả về null
as	Ép kiểu đối tượng (cú pháp: obj as MyClass)
is	Kiểm tra kiểu (đúng nếu đối tượng là kiểu chỉ định)
is !	Kiểm tra phủ định kiểu (đúng nếu đối tượng không phải kiểu chỉ định)

Bảng 3.2: Các toán tử truy cập lớp/đối tượng trong Dart.

3.4 Các cấu trúc điều khiển

Dart cung cấp các câu lệnh điều khiển luồng chương trình tương tự các ngôn ngữ khác.

3.4.1 Các cấu trúc rẽ nhánh

- **Cấu trúc if - else:** Lệnh `if` kiểm tra điều kiện và thực thi khối lệnh tương ứng

Ví dụ:

```
var number = 10;
if (number % 2 == 0) {
    print("Even Number");
} else {
    print("Odd Number");
}
// Output: Even Number
```

- **Cấu trúc switch - case:** Cho phép kiểm tra một biểu thức trên nhiều trường hợp (`case`), mỗi nhánh thường kết thúc bằng `break` để thoát khỏi switch.

Ví dụ:

```
var day = 3;  
switch (day) {  
  case 1:  
    print('Monday');  
    break;  
  case 2:  
    print('Tuesday');  
    break;  
  case 3:  
    print('Wednesday');  
    break;  
  default:  
    print('No Information');  
}
```

3.4.2 Cấu trúc lặp

Dart hỗ trợ các vòng lặp **for**, **while** và **do-while**

- **Vòng lặp for:**

Ví dụ:

```
for (int i = 1; i <= 5; i++) {  
  print('$i');  
}
```

- **Vòng lặp while:**

Ví dụ:

```
int i = 1;  
while (i <= 5) {  
  print('$i');  
  i++;  
}
```

- **Vòng lặp do - while:**

Ví dụ:

```
int i = 1;  
do {  
    print('$i');  
    i++;  
} while (i <= 5);
```

Ba cách trên đều cho ra cùng một kết quả:

```
1  
2  
3  
4  
5
```

Các lệnh break và continue

- Từ khóa `break` dùng để kết thúc vòng lặp ngay lập tức (bỏ qua các vòng lặp còn lại)

Ví dụ:

```
for (int i = 1; i <= 6; i++) {  
    if (i == 5) break;  
    print(i);  
}  
/* Output:  
1  
2  
3  
4
```

- Từ khóa `continue` bỏ qua phần còn lại của lần lặp hiện tại và tiếp tục với lần lặp kế tiếp

Ví dụ:

```
for (int i = 1; i <= 6; i++) {  
    if (i == 5) continue;
```

```
    print(i);  
}  
/* Output:  
1  
2  
3  
4  
6  
*/
```

3.4.3 Assert

Dart cũng hỗ trợ câu lệnh `assert` dùng trong giai đoạn debug để kiểm tra tính đúng đắn của điều kiện logic. Cú pháp `assert (!condition, 'message')` sẽ ngắt chương trình nếu điều kiện sai.

`assert` là một công cụ cực kỳ hữu ích để kiểm tra điều kiện logic khi đang phát triển ứng dụng (debug). Nó đảm bảo một điều kiện nào đó phải đúng tại thời điểm thực thi ngay lập tức.

Ví dụ:

```
void main() {  
    double accountBalance = 0;  
    assert(accountBalance > 0, "Not have enough money!")  
;  
    print("Transaction successful");  
}
```

Khi chạy chương trình, kết quả thu được ở Output là:

```
Uncaught Error, error: Error: Assertion failed:  
file:///tmp/dartpadHATVQD/lib/main.dart:5:5  
accountBalance > 0  
"Not have enough money!"
```

3.4.4 Xử lý ngoại lệ với try/catch và throw, on

Dart cung cấp cơ chế xử lý ngoại lệ giống nhiều ngôn ngữ khác: `try`, `catch`, `on` và `finally`. Khối `try-catch` dùng để bắt và xử lý lỗi bất ngờ. Có thể dùng `on Exception catch (e)` để bắt các loại ngoại lệ cụ thể. Khối `finally` luôn được thực thi sau khi `try` và `catch` (dù có lỗi hay không).

Ví dụ:

```
void main() {
    try {
        int a = 10 ~/ 0; // Division by zero causes an
        exception
    } catch (e) {
        print('An error occurred: $e');
        // Output: An error occurred:
        IntegerDivisionByZeroException
    }
}
```

Để có thể tự ném ra ngoại lệ, sử dụng `throw`

Ví dụ:

```
void checkAge(int age) {
    if (age < 18) {
        throw Exception('Not old enough');
    }
    print('Old enough!');
}

void main() {
    try {
        checkAge(15);
    } catch (e) {
        print('Error: $e');
    }
}
// Output: Error: Exception: Not old enough
```

Có thể sử dụng `on` để bắt ngoại lệ cụ thể

```
void main() {
    try {
        int result = int.parse('abc');
        print('Result: $result');
    } on FormatException catch (e) {
        print('Format error: $e');
    }
}

// Output: Format error: FormatException: abc
```

Khối lệnh nằm trong `finally` sẽ luôn được thực hiện dù mã có xảy ra ngoại lệ hay không.

```
void main() {
    try {
        int result = 10 ~/ 0;
        print('Result: $result');
    } catch (e) {
        print('An error occurred: $e');
    } finally {
        print('The finally block is always executed.');
    }
}
```

Kết quả trả ra của đoạn chương trình trên là:

```
An error occurred: Unsupported operation: Infinity
The finally block is always executed.
```

3.5 Xây dựng hàm

Hàm là tập hợp các câu lệnh lấy dữ liệu đầu vào, thực hiện một số phép tính cụ thể và tạo ra đầu ra. Các hàm có chức năng tái sử dụng mã nguồn của chương trình, chia chương trình phức tạp thành các nhóm nhỏ, tăng sự kết dính của các module.

Trong Dart, hàm (function) được định nghĩa theo cú pháp:

```
<return_type> functionName(<list_arguments>) {  
    // body...  
}
```

Trong đó, `<kieu_tra_ve>` có thể là một kiểu dữ liệu cụ thể(ví dụ như `int`, `String`), hoặc `void` nếu hàm không trả về giá trị. Nếu không ghi rõ kiểu trả về, Dart mặc định dùng `dynamic`.

Ví dụ minh họa về hàm trả về giá trị:

```
int add(int a, int b) {  
    // A simple operation: return the sum of a and b  
    // The add function returns an int value  
    return a + b;  
}  
  
void main() {  
    var result = add(3, 7);  
    print(result); // Output: 10  
}
```

Ví dụ về hàm không trả về giá trị `void`:

```
void GFG() {  
    // A function that prints a greeting message  
    print("Welcome to GeeksForGeeks");  
}  
  
void main() {  
    // Call the function  
    GFG();  
}  
  
// Output: Welcome to GeeksForGeeks
```

Trong Dart, hàm (function) là một kiểu dữ liệu, vì vậy nó có thể được gán cho biến hoặc truyền làm tham số cho hàm khác. Ví dụ:

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

3.5.1 Tham số tùy chọn

Dart hỗ trợ *tham số tùy chọn*, giúp người dùng có thể gọi hàm mà không cần truyền đủ tất cả tham số. Khi gọi hàm có thể sử dụng hoặc không, các tham số tùy chọn gom lại trong [], không truyền thì giá trị mặc định là null.

Ví dụ:

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

3.5.2 Tham số mặc định

Nếu muốn tham số có giá trị mặc định, nghĩa là khi gọi hàm mà thiếu giá trị cho tham số đó, thì nó sẽ nhận mặc định, các giá trị tham số mặc định sẽ được đặt trong cặp {}

Ví dụ:

```

double tinh tong (var a, {double b = 1, double c = 2}) {
    return a + b + c;
}

void main() {
    print(tinh tong(1)); // 4.0
    print(tinh tong(1, b: 3)); // 6.0
    print(tinh tong(1, c: 5)); // 7.0
}

```

3.5.3 Cú pháp hàm rút gọn(arro function)

Nếu hàm chỉ có một biểu thức, có thể viết rút gọn bằng toán tử =>:

<kiểu trả về> tenHàm(<tham_so>) => <biểu_thức>;

Ví dụ:

```

double tinh tong (var a, var b) {
    return a + b;
}

// Can be written as:
double tinh tong (var a, var b) => a + b;

```

3.5.4 Hàm ẩn danh - Lambda - Closure

Hầu hết khai báo hàm là có tên hàm, tuy nhiên trong nhiều ngữ cảnh khai báo hàm và không dùng đến tên, hàm đó gọi là **hàm ẩn danh**. Để tạo ra hàm ẩn danh, làm như hàm có tên bình thường, chỉ có điều phần kiểu trả về và tên bị thiếu:

Ví dụ:

```

// Anonymous function using regular syntax
(var a, var b) {
    return a + b;
};

// Using arrow syntax () => {}
(var a, var b) => {
    return a + b;
};

```

```
// If the body only contains a return expression, you can
// simplify it
(var a, var b) => a + b;
```

Khai báo trên là hàm ẩn danh, nhưng để sử dụng nó thì ra lệnh cho nó chạy luôn hoặc gán nó vào một biến rồi dùng biến gọi hàm.

Ví dụ:

```
// Declare and invoke immediately
var x = (var a, var b) {
    return a + b;
} (5, 6);

print(x); // Output: 11

// Assign anonymous function to a variable
var add = (var a, var b) {
    return a + b;
};

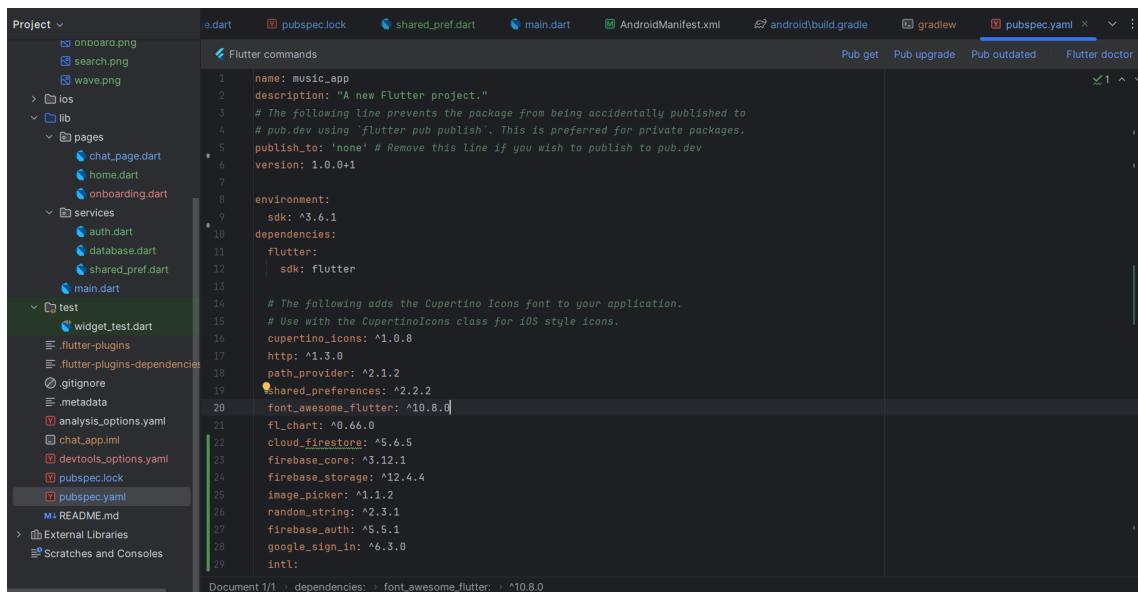
print(add(10, 11)); // Output: 21
```

3.6 Quản lý các gói trong Dart

Dart sử dụng công cụ pub làm trình quản lý gói (package manager) để tải và cài đặt các thư viện (package) từ kho lưu trữ **pub.dev**. Mỗi gói Dart cần một file **pubspec.yaml** chứa metadata về gói đó, bao gồm tên gói, mô tả, phiên bản và danh sách các phụ thuộc (dependencies) mà gói cần.

File **pubspec.yaml** đóng vai trò tập trung thông tin để công cụ pub quản lý và phân phối gói, đồng thời hỗ trợ công cụ của Dart và Flutter trong quá trình biên dịch và xuất bản gói.

PHẦN 3. CÚ PHÁP CƠ BẢN CỦA NGÔN NGỮ DART



```
name: music_app
description: "A new Flutter project."
# The following line prevents the package from being accidentally published to
# pub.dev using 'flutter pub publish'. This is preferred for private packages.
publish_to: 'none' # Remove this line if you wish to publish to pub.dev
version: 1.0.0+1

environment:
  sdk: '3.6.1'

dependencies:
  flutter:
    sdk: flutter

# The following adds the Cupertino Icons font to your application.
# Use with the CupertinoIcons class for iOS style icons.
cupertino_icons: ^1.0.8
http: ^1.3.0
path_provider: ^2.1.2
shared_preferences: ^2.2.2
font_awesome_flutter: ^10.8.0
fl_chart: ^0.66.0
cloud_firestore: ^3.6.5
firebase_core: ^3.12.1
firebase_storage: ^12.4.4
image_picker: ^1.1.2
random_string: ^2.3.1
firebase_auth: ^5.6.1
google_sign_in: ^6.3.0
intl:
```

Hình 3.1: Hình ảnh nội dung file pubspec.yaml trong project thực tế

Trong **pubspec.yaml**, các trường metadata chính thường gặp bao gồm:

- **name**: Tên của package (bắt buộc phải có cho mọi gói Dart).

Ví dụ: name: music_app

- **version**: Phiên bản của gói

Ví dụ: version: 1.0.0+1

- **description**: Mô tả ngắn gọn về gói hoặc dự án

Ví dụ: description: "A new flutter project"

- **dependencies**: Danh sách các gói khác mà gói này phụ thuộc. Nếu dự án không dùng gói ngoài nào thì phần này có thể để trống hoặc không khai báo.

Trong dependencies, mỗi mục là tên gói và phiên bản tương thích.

Ví dụ:

dependencies:

cupertino_icons: ^1.0.8

http: ^1.3.0

Ngoài ra, còn có các trường thông tin khác như homepage, repository, environment, ... tùy nhu cầu dự án.

Cách sử dụng các lệnh import trong Dart:

Để sử dụng các thư viện trong mã Dart, cú pháp chung là `import 'uri'`. Có 3 dạng uri chính:

- **Thư viện chuẩn của Dart:** Ví dụ `dart:core`, `dart:math`, `dart:convert`, `dart:io`, Những thư viện này cung cấp các chức năng cơ bản và thường được nhập tự động. Ví dụ, thư viện toán học được import bằng `import 'dart:math'`; để sử dụng các hàm như `sqrt` hoặc hằng số π .

```
import 'dart:math';
void main() {
    int a = 4;
    double b = 2.0;
    print('${sqrt(a)}');
    print('${max(a, b)}');
}
/*
2
4
*/
```

- **Tệp nguồn nội bộ (file dự án):** Sử dụng đường dẫn tương đối hoặc tuyệt đối đến file Dart trong dự án.

Ví dụ, `import 'lib/myfile.dart'` sẽ nạp file `myfile.dart` nằm trong thư mục `lib/`

- **Thư viện từ gói tải về (packages):** Dùng định danh theo dạng package:

`package:<ten_goi>/<thu_vien_goi>`

Ví dụ, gói `googleapis_auth` có thành phần `auth_browser` cung cấp chức năng xác thực Auth với tài khoản Google thì nạp thư viện đó vào bằng:
`import "package:googleapis_auth/auth_browser.dart"`

Một số thư viện chuẩn phổ biến trong Dart:

Dart cung cấp một loạt thư viện mặc định (bắt đầu bằng dart:) để xử lý nhiều tác vụ thông thường. Các thư viện quan trọng bao gồm:

- **dart:core**: Thư viện lõi cung cấp các kiểu dữ liệu cơ bản và hàm tiện ích.

Ví dụ:

```
void main() {  
    String name = 'Khanh';  
    int age = 20;  
    print('Name: $name, Age: $age');  
  
    var now = DateTime.now();  
    var later = now.add(Duration(days: 2));  
    print('Two days later is: $later');  
}  
/*  
Name: Khanh, Age: 20  
Two days later is: 2025-05-02 07:34:20.215  
*/
```

- **dart:collection**: Cung cấp các cấu trúc dữ liệu nâng cao như **HashSet**, **HashMap**, **Queue**... phục vụ lưu trữ và truy vấn dữ liệu động.

Ví dụ về **Queue**:

```
import 'dart:collection';  
void main() {  
    Queue<String> queue = Queue();  
    queue.addAll(['one', 'two', 'three']);  
    print(queue.removeFirst()); // one  
    print(queue); // (two, three)  
}
```

Ví dụ về **SplayTreeSet**:

```
// Import the collection library for SplayTreeSet  
import 'dart:collection';
```

```

void main() {
    // Create a SplayTreeSet of integers
    var evenNumbers = SplayTreeSet<int>();
    evenNumbers.addAll([10, 2, 6, 4, 8]);

    print('Set after adding elements:');
    for (var x in evenNumbers) {
        print(x);
    }

    // Remove the value 6
    evenNumbers.remove(6);
    print('\nAfter removing number 6:');
    print(evenNumbers);

    // Display the smallest and largest values
    print('\nSmallest number: ${evenNumbers.first}');
    print('Largest number: ${evenNumbers.last}');
}

```

Kết quả của chương trình trên in ra là:

Set after adding elements:

2
4
6
8
10

After removing number 6:

{2, 4, 8, 10}

Smallest number: 2

Largest number: 10

- `dart:math`: Thư viện toán học chứa các hàm và các hằng số cơ bản.

Ví dụ:

```
import 'dart:math';

void main() {
    print(pi);           // 3.141592...
    print(sqrt(49));    // 7.0
    print(pow(3, 3));   // 27

    // Print a random integer from 0 to 9
    print(Random().nextInt(10));
}
```

- **dart:convert:** Cung cấp bộ mã hóa và giải mã dữ liệu, hỗ trợ JSON, UTF-8, và các định dạng dữ liệu khác.

Ví dụ về chuyển đổi từ **Map** thành **JSON**:

```
import 'dart:convert';
void main() {
    var user = {'name': 'Hai', 'age': 22};
    var jsonStr = jsonEncode(user);
    print(jsonStr); // {"name": "Hai", "age": 22}
}
```

Ví dụ giải mã từ **JSON**:

```
import 'dart:convert';
void main() {
    var jsonStr = '{"name": "Hai", "age": 22}';
    var user = jsonDecode(jsonStr);
    print(user['name']); // Hai
}
```

Ví dụ về mã hóa và giải mã **Base64**:

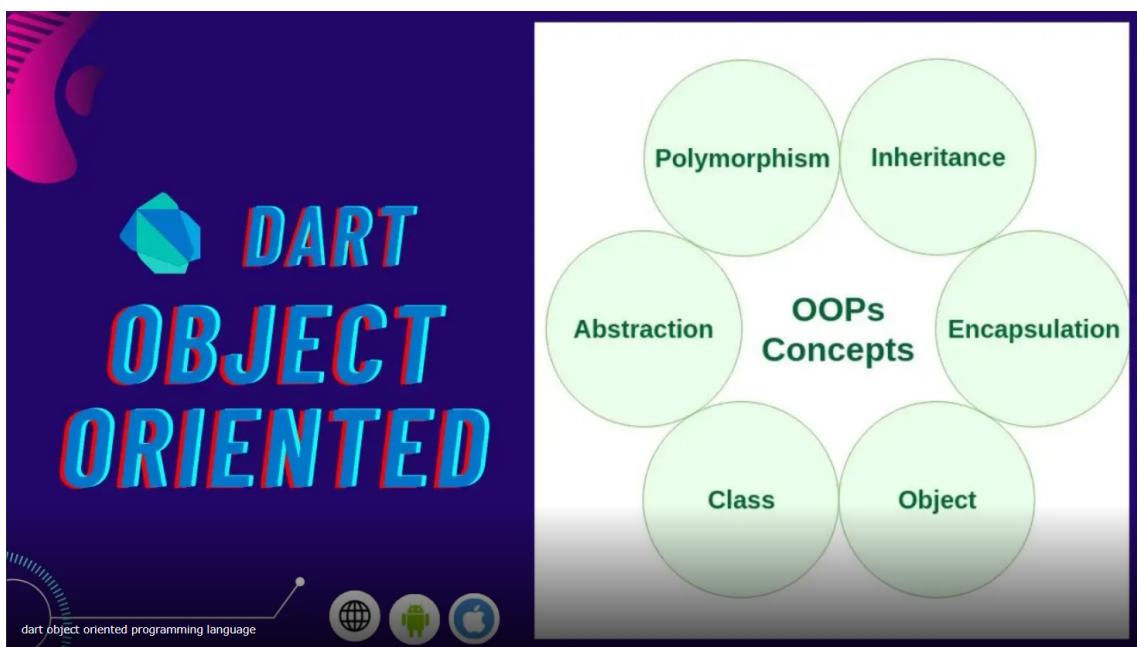
```
import 'dart:convert';
void main() {
    var message = 'Dart';
    var bytes = utf8.encode(message);
```

```
var base64Str = base64Encode(bytes);  
print(base64Str); // RGFyda==  
  
var x = utf8.decode(base64Decode(base64Str));  
print(x); // Dart  
}
```

- `dart:io`: Thư viện nhập xuất cho chương trình chạy trên máy (VM), hỗ trợ thao tác file, socket, HTTP,...
- `dart:js`, `dart:html`: Các thư viện phục vụ lập trình Web. Trong đó `dart:html` cho phép thao tác DOM và API HTML5 trên trình duyệt, còn `dart:js` hỗ trợ tương tác với mã JavaScript.

PHẦN 4. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG TRONG DART

Lập trình hướng đối tượng (Object-Oriented Programming – OOP) là một mô hình lập trình dựa trên khái niệm đối tượng. Mỗi đối tượng trong OOP đóng gói dữ liệu (thuộc tính) và các hành vi (phương thức), và chương trình được thiết kế bằng cách tạo ra các đối tượng tương tác với nhau. Tương tự như nhiều ngôn ngữ lập trình hiện đại khác như Java, C++, hay C#, Dart cũng cung cấp đầy đủ các đặc trưng của lập trình hướng đối tượng như: **đối tượng, lớp, tính đóng gói, tính kế thừa, tính đa hình và tính trừu tượng**. Nó có vai trò quan trọng trong việc xây dựng các ứng dụng lớn với cấu trúc rõ ràng, dễ bảo trì và mở rộng.



Hình 4.1: Các nguyên lý OOP trong Dart

4.1 Các đặc trưng hướng đối tượng trong Dart

4.1.1 Đối tượng và lớp

- **Lớp (Class)** là một khuôn mẫu dùng để định nghĩa các thuộc tính và phương thức, các thành phần tĩnh của một kiểu đối tượng. Nó mô tả cấu trúc và hành vi chung của các đối tượng cùng loại.
- **Object (Đối tượng)** là thể hiện cụ thể của một lớp, được tạo ra qua constructor

của lớp, có trạng thái riêng biệt và có thể thực hiện được các hành vi được định nghĩa trong lớp.

Việc sử dụng lớp và đối tượng trong Dart giúp tổ chức dữ liệu và hành vi thành các đơn vị rõ ràng, đóng góp vào tính module hóa của chương trình.

Trong Dart, lớp được khai báo bằng từ khóa `class`.

```
class class_name {  
    // Body of class...  
}
```

Các đối tượng được khai báo bằng cách sử dụng từ khóa `new` và sau là tên lớp.

```
var object_name = new class_name([ arguments ]);
```

Để minh họa cách sử dụng lớp và đối tượng trong Dart, ta xét ví dụ về lớp `Product` mô phỏng một sản phẩm:

```
class Product {  
    // Declare attributes  
    String manufacture = '';  
    String name = '';  
    var price;  
    int quantity;  
  
    // Declare constructor  
    Product(var price, {int quantity = 0}) {  
        this.price = price;  
        this.quantity = quantity;  
    }  
  
    // Declare methods  
    calculateTotal() {  
        return this.price * this.quantity;  
    }  
}
```

```
showTotal() {  
    var total = this.calculateTotal();  
    print("The total amount is: $total");  
}  
}
```

Tạo và sử dụng đối tượng:

```
var product = new Product(600, quantity: 1);  
product.showTotal();  
product.quantity = 2;  
product.showTotal();
```

Kết quả chạy chương trình:

```
The total amount is: 600  
The total amount is: 1200
```

Ở ví dụ trên, lớp **Product** có các thuộc tính `manufacture`, `name`, `price`, `quantity`, hàm khởi tạo với 2 tham số: tham số bắt buộc `price` và tham số tùy chọn `quantity` với giá trị mặc định là 0. Lớp có hai phương thức `calculateTotal()` tính tổng giá trị đơn hàng và `showTotal()` để in ra tổng số tiền.

Trong Dart, từ khóa **this** dùng để tham chiếu đến chính đối tượng của lớp dùng từ khóa. Sau đó, khởi tạo đối tượng `product` từ lớp **Product** với `price` là 600 và `quantity` là 1. Phương thức `product.showTotal()` tính tổng số tiền dựa trên `price` và `quantity`. Tiếp theo, gán lại giá trị cho `quantity` thành 2 và gọi lại `showTotal()` để in lại tổng số tiền.

Trong Dart, một lớp có thể định nghĩa nhiều constructor bằng cách sử dụng **named constructors**. Điều này cho phép tạo ra các cách khởi tạo đối tượng khác nhau, phục vụ cho các mục đích cụ thể.

Ví dụ:

```

class Student {
    String name;
    int age;

    // Primary constructor
    Student(this.name, this.age);

    // Named constructor for unknown students
    Student.unknown() : name = 'Unknown', age = 0;

    void info() {
        print('Name: $name - Age: $age');
    }
}

void main() {
    var s1 = Student('Ky', 20);
    s1.info(); // Name: Ky - Age: 20

    var s2 = Student.unknown();
    s2.info(); // Name: Unknown - Age: 0
}

```

Từ khóa **static** giúp khai báo các thuộc tính hoặc phương thức không gắn liền với một thể hiện cụ thể, mà thuộc về lớp.

```

class TempConverter {
    static double zeroCelsiusToF = 32;

    static double toCelsius(double fahrenheit) {
        return (fahrenheit - 32) * 5 / 9;
    }
}

void main() {
    print(TempConverter.zeroCelsiusToF); // 32
    print(TempConverter.toCelsius(100)); // 37.777...
}

```

Phương thức `toCelsius()` và thuộc tính `zeroCelsiusToF` là static, cho phép gọi trực tiếp qua tên lớp **TempConverter** mà không cần khởi tạo đối tượng.

4.1.2 Đóng gói:

Đóng gói là việc gom nhóm dữ liệu và phương thức vào cùng một lớp, hạn chế truy cập bên ngoài nhằm bảo vệ tính toàn vẹn của dữ liệu. Không giống như đa phần các ngôn ngữ khác có các chỉ định truy cập **public**, **private**, **protect**, Dart chỉ thể hiện tính đóng gói thông qua tiền tố `_` mà không có giới hạn truy cập một cách tường minh. Đồng thời, Dart hỗ trợ các phương thức getter và setter để kiểm soát việc truy xuất và cập nhật thuộc tính.

Ví dụ:

```
class BankAccount {  
    // _balance is a private variable, accessible only  
    // within this class  
    double _balance;  
  
    BankAccount(this._balance);  
  
    // Getter to access the _balance  
    double get balance => _balance;  
  
    // Method to deposit money  
    void deposit(double amount) {  
        if (amount > 0) {  
            _balance += amount;  
        }  
    }  
}
```

Ở hàm `main()`, ta thực hiện thay đổi số dư tài khoản:

```
void main() {  
    var account = BankAccount(1000);  
    account.deposit(500);  
    print(account.balance); // Output: 1500
```

}

Trong ví dụ trên, thuộc tính `_balance` được khai báo với tiền tố gạch dưới (`_`) cho biết nó là thuộc tính private của lớp **BankAccount**. Đoạn mã trong hàm `main()` không thể truy cập trực tiếp `_balance`; thay vào đó, ta sử dụng phương thức công khai `deposit` để thay đổi số dư và **getter** `balance` để đọc giá trị. Nhờ vậy, lớp **BankAccount** kiểm soát được cách thức cập nhật dữ liệu, đảm bảo giá trị `_balance` không bị gán giá trị không hợp lệ từ bên ngoài. Đó chính là tính chất đóng gói: nhóm dữ liệu và phương thức xử lý, đồng thời bảo vệ dữ liệu riêng tư.

4.1.3 Kế thừa

Kế thừa cho phép một lớp con tái sử dụng và mở rộng các thuộc tính và phương thức của lớp cha. Trong Dart, từ khóa **extends** được sử dụng để kế thừa một lớp khác, và từ khóa **super** cho phép lớp con truy cập thành phần của lớp cha. Lớp con có thể **ghi đè (override)** các phương thức của lớp cha bằng chú thích **@override** để định nghĩa hành vi riêng. Dart chỉ hỗ trợ kế thừa đơn (một lớp con chỉ có một lớp cha) nhưng cho phép sử dụng **mixins** để mô phỏng đa kế thừa.

Cú pháp để một lớp con kế thừa một lớp cha:

```
class ChildClass extends ParentClass {
    // Childclass members
}
```

Ví dụ về sử dụng từ khóa **extends**:

```
// Create a parent class
class Animal {
    String name;
    Animal(this.name);

    void makeSound() {
        print("The animal makes a sound");
    }
}
```

```

        }
    }

    // Create a child class that inherits using the 'extends'
    // keyword

    class Dog extends Animal {
        String breed;

        Dog(String name, this.breed) : super(name);

        @override
        void makeSound() {
            print("The dog barks");
        }
    }

    void main() {
        // Create an instance of the child class
        var myDog = Dog("Buddy", "Golden Retriever");

        print(myDog.name);           // Output: Buddy
        print(myDog.breed);         // Output: Golden Retriever
        myDog.makeSound();          // Output: The dog barks
    }
}

```

Lớp **Animal** có thuộc tính `breed`, phương thức `makeSound()`. Lớp **Dog** kế thừa từ lớp **Animal** chỉ ra rằng lớp này kế thừa mọi thành viên từ **Animal**. Trong lớp con có sử dụng chú thích `@override` để ghi đè phương thức `makeSound()` của lớp cha. Ở `main()`, ta khởi tạo đối tượng `myDog` thuộc lớp **Dog** với hai thuộc tính `name`, `breed`. Khi thực hiện `myDog.makeSound()` thì chương trình sẽ thực hiện phương thức `makeSound()` của lớp con.

Toán tử **cascade** `(..)` của Dart cho phép thực hiện liên tiếp nhiều thao tác trên cùng một đối tượng mà không cần lặp lại tên đối tượng. Ví dụ, nếu không dùng **cascade**, ta phải viết:

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

Còn khi sử dụng toán tử **cascade**, ta chỉ cần:

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

Nếu đối tượng có thể là null, ta nên dùng toán tử chuỗi null (`?...`) để tránh lỗi khi gọi phương thức trên null.

Giao diện (Interface):

Khi một lớp được coi là giao diện thì lớp triển khai của nó phải định nghĩa lại mọi phương thức, thuộc tính có trong giao diện. Để một lớp thực thi giao diện này mà không kế thừa trực tiếp, ta dùng từ khóa **implement**

Ví dụ:

```
class PrinterInterface {
    void display(String text);
}

class ConsolePrinter implements PrinterInterface {
    @override
    void display(String text) {
        print('Printed: $text');
    }
}

void main() {
    PrinterInterface p = ConsolePrinter();
    p.display('Hello Dart'); // Printed: Hello Dart
}
```

Lớp **PrinterInterface** có định nghĩa phương thức `display()`. Lớp **ConsolePrinter** được khai báo implements **PrinterInterface** và cung

cấp phần triển khai của `display()`. Khi chạy, ta thấy in ra Printed: Hello Dart. Sử dụng `implements` buộc lớp con phải thực thi (override) tất cả các phương thức của interface đã khai báo. Một lớp có thể thực thi nhiều giao diện cùng một lúc, ví dụ:

```
class Multi implements InterfaceA, InterfaceB {  
    ...  
}
```

Mixin:

Mixin là một cơ chế tái sử dụng mã cho nhiều lớp khác nhau thông qua từ khóa `with`. Mixin không được sử dụng trực tiếp để tạo ra một đối tượng, mà nó chứa các phương thức, thuộc tính dùng để gộp vào một lớp khác.

Ví dụ:

```
 mixin Flyer {  
     void fly() {  
         print('Flying');  
     }  
 }  
  
 class Bird with Flyer {  
     void sing() {  
         print('Singing');  
     }  
 }  
  
 void main() {  
     var b = Bird();  
     b.fly(); // Output: Flying  
     b.sing(); // Output: Singing  
 }
```

Flyer là một mixin định nghĩa phương thức `fly()`. Lớp **Bird** sử dụng `with Flyer` nên tự động có cả phương thức `fly()`. Khi chạy, chương trình in Flying và Singing. Như vậy, với mixin ta có thể bổ sung hành vi cho lớp mà không cần thiết

lập quan hệ kế thừa.

4.1.4 Trừu tượng hóa:

Trùu tượng (abstraction) trong OOP cho phép định nghĩa các lớp hoặc phương thức chưa triển khai chi tiết để tạo nên một khái niệm chung. Trong Dart, từ khóa **abstract** được dùng khai báo một lớp hoặc phương thức là trừu tượng. Một **lớp trừu tượng** không thể được tạo đối tượng trực tiếp và có thể chứa các phương thức trừu tượng (chỉ khai báo chữ ký, không có phần thân). Các lớp con phải ghi đè và triển khai các phương thức trừu tượng đó.

Ví dụ:

```
abstract class Shape {  
    // Abstract method without implementation  
    double area();  
}  
  
class Square implements Shape {  
    double side;  
    Square(this.side);  
  
    @override  
    double area() {  
        return side * side;  
    }  
}  
  
void main() {  
    Shape s = Square(5);  
    print('Area of the square: ${s.area()}');  
    // Output: Area of the square: 25  
}
```

Trong ví dụ trên, **Shape** là lớp trừu tượng khai báo phương thức `area()` mà không cung cấp định nghĩa. Lớp `Square` dùng từ khóa `implements` để triển khai giao diện của `Shape`, do đó nó phải cung cấp hiện thực cho phương thức

area(). Chú thích @override khẳng định rằng Square.area() ghi đè và định nghĩa chức năng cụ thể. Khi trong main(), ta có thể khai báo biến s kiểu Shape nhưng khởi tạo dưới dạng Square(5) để thể hiện rằng Square là một hiện thực của khái niệm trừu tượng Shape. Việc sử dụng lớp trừu tượng cho phép tách biệt phần khai báo chung của các đối tượng cùng loại với phần triển khai cụ thể.

4.1.5 Đa hình

Đa hình (polymorphism) là một nguyên lý cốt lõi trong lập trình hướng đối tượng, cho phép các đối tượng thuộc các lớp khác nhau được xử lý thông qua một giao diện chung. Nói cách khác, nhiều đối tượng khác loại có thể thực hiện cùng một lời gọi phương thức theo cách riêng của chúng. Trong Dart, đa hình được thực hiện chủ yếu thông qua cơ chế **kế thừa** và **ghi đè** phương thức (method overriding). Dart sẽ xác định và thực thi phương thức phù hợp với lớp thực tế của đối tượng tại thời điểm chạy chương trình, giúp mã nguồn linh hoạt và dễ mở rộng hơn.

Ví dụ:

```
class Animal{
    void speak() {
        print('...');
    }
}

class Dog extends Animal{
    @override void speak() {
        print('Grhh grhh!');
    }
}

class Cat extends Animal{
    @override void speak() {
        print("Meo meo!");
    }
}

void main(){
    List<Animal> zoo = [Dog(), Cat()];
}
```

```
for (var a in zoo) a.speak();  
}
```

Kết quả chương trình trên in ra là:

```
Grhh grhh!  
Meo meo!
```

Đoạn mã trên thể hiện tính đa hình khi ta tạo một danh sách `zoo` chứa các đối tượng thuộc lớp `Dog` và `Cat`, cả hai đều là các lớp con của `Animal`. Mặc dù danh sách khai báo kiểu chung là `Animal`, khi lặp qua từng phần tử và gọi `a.speak()`, đối tượng thực thi phương thức tương ứng với lớp của nó: `Dog` in ra "Grhh grhh!" còn `Cat` in ra "Meo meo!". Như vậy, cùng một lời gọi `speak()` nhưng kết quả lại khác nhau tùy vào lớp thực thể, đúng với khái niệm đa hình.

Đa hình giúp viết mã tổng quát hơn: Người dùng có thể xử lý tập hợp các đối tượng khác nhau cùng họ (kế thừa từ một lớp cha hoặc cùng triển khai một giao diện) mà không cần biết chính xác loại con của chúng, tạo điều kiện thuận lợi cho mở rộng và bảo trì.

4.2 Callable class

Trong Dart, phương thức đặc biệt `call()` cho phép đối tượng của một lớp hoạt động như một hàm. Khi một lớp định nghĩa một phương thức có tên `call`, ta có thể gọi một instance của lớp đó giống như gọi một hàm bình thường, có thể nhận đối số và trả về kết quả. Cú pháp này giúp mở rộng khả năng của lớp, cho phép sử dụng đối tượng như một hàm có thể gọi được (callable).

Ví dụ:

```
class Employee{  
    String name;  
    Employee(this.name);  
    void call() {  
        print('Hello, I am $name');
```

```
    }
}

void main() {
    var emp = Employee('Alice');
    emp(); // method call()
}
```

Trong ví dụ trên, khi gọi `emp()`, Dart thực chất sẽ gọi phương thức `call()` của đối tượng `emp`. Vì vậy chương trình sẽ in ra kết quả:

```
Hello, I am Alice
```

4.3 Giới thiệu về Async Programming

Lập trình đồng bộ (synchronous) và **lập trình bất đồng bộ (asynchronous)** là hai mô hình thực thi khác nhau trong Dart.

4.3.1 Lập trình đồng bộ

Code chạy trong Dart là chạy trên một luồng (thread), dòng code được thi hành hết câu lệnh này sang câu lệnh khác. Nếu một khối lệnh nào đó gây block thread (làm tắc thread) thì toàn bộ ứng dụng sẽ bị treo.

Ví dụ:

```
const info = "#4fs358w";
getInformation() {
    return info;
}
showInfomation() {
    var data = getInformation();
    print('Data - ' + DateTime.now().toString());
    print(data);
}
secondFunction() {
    print(Time - ' + DateTime.now().toString());
}
void main() {
    showInfomation();
```

```

        secondFunction();
    }
}

```

Giả sử code ở trên nếu showInformation() hoặc getInformation() mất nhiều thời gian để hoàn thành thì khôi lệnh khác như secondFunction() phải chờ nó hoàn thành thì mới được thi hành.

4.3.2 Lập trình bất đồng bộ

Lập trình bất đồng bộ - Asynchronous: Khi một phương thức đang thực hiện công việc của mình thì khôi lệnh khác - hàm khác vẫn được thi hành. Cơ chế bất đồng bộ là chương trình cho phép phân nhánh quá trình code hoạt động, làm cho có cảm giác như đa luồng (có thể vẫn là 1 thread) - có lúc thì chạy code ở nhánh này, có lúc thì chạy code ở nhánh khác - cảm giác thi hành nhiều việc đồng thời.

Dart sử dụng lớp **Future** với các từ khóa `async`, `await`.

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

```

Future<String> fetchData() async {
    await Future.delayed(Duration(seconds: 2));
    return "Get data successfully";
}

```

Trong ví dụ trên, `fetchData()` trả về một `Future<String>`, nghĩa là nó sẽ trả về một chuỗi sau khi tác vụ (giả lập bằng `Future.delayed`) hoàn thành.

Hàm bất đồng bộ async: Khai báo có từ khóa `async` phía sau, trả về đối tượng là **Future<T>**.

Cú pháp: Đặt `async` trước khôi mã của hàm.

Ví dụ:

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

Nếu hàm đó đã khai báo là bất đồng bộ `async` thì trong hàm có thể sử dụng cú pháp `await` biến _thức. Khi gặp `await`, chương trình sẽ tạm dừng thực thi hàm hiện tại và chuyển sang thực thi các tác vụ khác trong hàng đợi sự kiện (event loop). Khi Future hoàn thành, chương trình sẽ tiếp tục thực thi từ dòng mã ngay sau `await`.

Ví dụ:

```
Future<String> getInfomation() async {
    return "Employee information (from Future)";
}

Future<void> showInfomation() async {
    print(await getInfomation());
}

void main() {
    showInfomation();
}
```

Hàm `getInfomation` được đánh dấu `async` và trả về `Future<String>`. Điều này nghĩa là hàm này khởi tạo một tác vụ bất đồng bộ và sẽ hoàn thành trong tương lai. Từ khóa `await` trong hàm `showInfomation` được sử dụng để đợi giá trị kết quả từ `getInfomation()`. Cụ thể, khi gọi `await getInfomation()`, chương trình sẽ tạm dừng tại vị trí này trong hàm `showInfomation` cho đến khi `getInfomation` trả về giá trị. Sau đó, kết quả nhận được sẽ được in ra.

Xử lý lỗi trong lập trình bất đồng bộ: Trong các hàm bất đồng bộ, có thể sử dụng các khối `try-catch` để xử lý lỗi từ các Future. Đây là một phần quan trọng để đảm bảo chương trình không bị dừng đột ngột khi gặp lỗi.

Ví dụ:

```
Future<String> fetchUserOrder() async {
    throw Exception("Cannot locate user order");
}

Future<void> main() async {
    try {
        var order = await fetchUserOrder();
        print(order);
    } catch (err) {
        print('Caught error: $err');
    }
}
```

Kết quả in ra của chương trình trên là:

```
Caught error: Exception: Cannot locate user order
```

Streams: Ngoài Future, Dart còn hỗ trợ **Stream** để xử lý các chuỗi sự kiện bất đồng bộ, như dữ liệu từ mạng hoặc sự kiện người dùng. **Stream** đại diện cho một chuỗi các giá trị được phát ra theo thời gian.

Ví dụ:

```
Stream<int> getNumbers() async* {
    for (int i = 0; i < 5; i++) {
        await Future.delayed(Duration(seconds: 1));
        yield i;
    }
}

Future<void> main() async {
    await for (var number in getNumbers()) {
        print(number);
    }
}
```

Kết quả in ra của chương trình trên là các số 0, 1, 2, 3, 4, số sau xuất hiện sau số

trước 1 giây:

0
1
2
3
4

PHẦN 5. GIỚI THIỆU FRAMEWORK FLUTTER

5.1 Giới thiệu về Flutter



Hình 5.1: Flutter - Framework phát triển ứng dụng đa nền tảng

Flutter là một framework phát triển ứng dụng đa nền tảng (Cross-Platform SDK) dành cho thiết bị di động. Nó cung cấp một hệ sinh thái đầy đủ cho việc phát triển ứng dụng di động hiện đại, bao gồm:

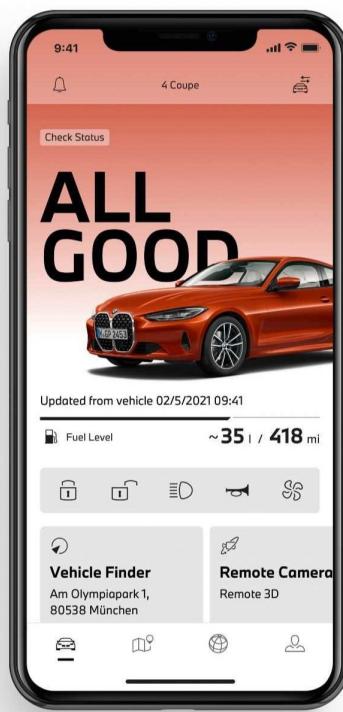
- **Khung giao diện người dùng (UI Framework):** Flutter cung cấp một mô hình giao diện dựa hoàn toàn trên các widget, trong đó mỗi widget đại diện cho một phần tử giao diện từ nhỏ nhất như nút bấm, đoạn văn bản đến toàn bộ màn hình ứng dụng.
- **Danh mục widget phong phú:** Bao gồm cả widget được thiết kế theo chuẩn Material Design (Android) và Cupertino (iOS), giúp ứng dụng hoạt động và hiển thị phù hợp với nền tảng đích.
- **Công cụ phát triển mạnh mẽ:** Hỗ trợ gỡ lỗi, phân tích hiệu năng (performance profiling), tích hợp chặt với các IDE phổ biến (VS Code, Android Studio), và hỗ trợ tính năng nổi bật **hot reload**.

- **Khả năng đa nền tảng:** Flutter có thể build ứng dụng không chỉ trên thiết bị di động mà còn hỗ trợ máy tính để bàn (desktop), trình duyệt (web), và các nền tảng thử nghiệm như Google Fuchsia.

Mục đích của Flutter hướng đến việc dễ dàng triển khai ứng dụng trên bất kỳ nền tảng nào với mã nguồn duy nhất. Nó cho phép phát triển hiệu quả nhờ cấu trúc UI dựa trên các widget khai báo, khả năng tương thích đa nền tảng, máy ảo hỗ trợ **hot reload**, và đặc biệt là cách tiếp cận **lập trình phản ứng** (reactive programming). Trong Flutter, giao diện người dùng được xây dựng như một cây widget phản ánh trực tiếp trạng thái của ứng dụng. Khi trạng thái thay đổi, UI sẽ tự động được cập nhật theo cách tối ưu nhất, giúp việc phát triển trở nên nhanh chóng, chính xác và dễ bảo trì. Việc này không chỉ giúp rút ngắn thời gian phát triển mà còn nâng cao trải nghiệm người dùng.

Lịch sử phát triển:

- **2015:** Flutter được phát triển bởi Google và lần đầu tiên được giới thiệu vào năm 2015 với tên gọi "Sky". Khi đó, nó chỉ có một tính năng chính là "Hot Reload" trên thiết bị Android, giúp giảm thời gian phát triển ứng dụng từ 7 phút xuống còn 400ms.
- **5/2017:** Flutter chính thức được phát hành với phiên bản ổn định. Đây là một bước ngoặt quan trọng, khi Flutter trở thành một framework mã nguồn mở, hỗ trợ phát triển ứng dụng cho cả Android và iOS từ một codebase duy nhất. Đồng thời, Flutter cũng bắt đầu được sử dụng trong các ứng dụng thương mại, như Google Pay và Google Earth.
- **2018:** Flutter tiếp tục phát triển với sự hỗ trợ mạnh mẽ từ Google. Framework này bắt đầu thu hút sự chú ý của các nhà phát triển trên toàn thế giới nhờ khả năng phát triển nhanh chóng và hiệu suất cao. Các công ty lớn như Alibaba (với ứng dụng Xianyu) và BMW (với ứng dụng MyBMW) bắt đầu áp dụng Flutter cho các dự án của họ.



Hình 5.2: Giao diện chính app MyBMW - sử dụng Flutter

- **2020:** Flutter mở rộng hỗ trợ sang các nền tảng khác, bao gồm web, desktop (Windows, macOS, Linux), cung cấp vị thế của Flutter như một framework đa nền tảng thực sự.
- **2023:** Theo báo cáo từ Flutter team, hơn một triệu ứng dụng đã được xuất bản sử dụng Flutter, vượt qua tổng số ứng dụng được phát triển trên tất cả các framework đa nền tảng khác cộng lại.
- **Hiện nay (2025):** Flutter tiếp tục được Google hỗ trợ và phát triển, có một hệ sinh thái phong phú với khoảng 37.500 gói thư viện (package) và plugin được chia sẻ trên kho lưu trữ chính thức pub.dev

Cộng đồng Flutter vẫn đang phát triển mạnh mẽ cùng với sự đóng góp tích cực từ các lập trình viên trên toàn thế giới ngày càng giúp tăng cường hiệu quả và tính linh hoạt của framework.

Đối tượng sử dụng:

Flutter phù hợp với nhiều nhóm đối tượng, mang lại nhiều lợi ích:

- **Lập trình viên mới, chưa có nhiều kinh nghiệm:** Dễ học, dễ cài đặt, không cần kinh nghiệm phát triển di động, với tài liệu chi tiết và cộng đồng hỗ trợ lớn. Theo khảo sát Stack Overflow 2024, 12,4% lập trình viên đã sử dụng Flutter, và 60,6% trong số đó muốn tiếp tục sử dụng, cho thấy mức độ phổ biến và sự tin tưởng.
- **Lập trình viên Web chuyển sang di động:** Những người quen JavaScript hoặc TypeScript có thể dễ dàng chuyển sang Flutter, tận dụng codebase chung cho Android và iOS, giảm thời gian phát triển. Theo Nomtek Blog, Flutter cho phép tái sử dụng 80-95% mã nguồn, tiết kiệm chi phí và công sức.

Ví dụ: Flutter hỗ trợ phát triển web với PWA (Progressive Web Apps), phù hợp cho cửa hàng trực tuyến, trang tin tức, hoặc trang đăng ký, với thiết kế nhất quán và hiệu suất cao

- **Doanh nghiệp và startup:** Flutter lý tưởng cho dự án mới để kiểm chứng ý tưởng nhanh chóng nhờ chu kỳ phát triển nhanh và widget sẵn có. Các startup như Nubank, Invoice Ninja, Reflectly đã thành công với Flutter nhờ tính chi phí hiệu quả và tính năng phong phú. Các công ty như Toyota, BMW, eBay, Alibaba, Google (Google Pay, Google Earth, Google Analytics) sử dụng Flutter cho ứng dụng thương hiệu, với lợi ích như giảm thời gian phát triển và giảm kích thước codebase (giảm kích thước 45%, phát hành nhanh hơn 44%).



Hình 5.3: Danh sách công ty lớn sử dụng Flutter

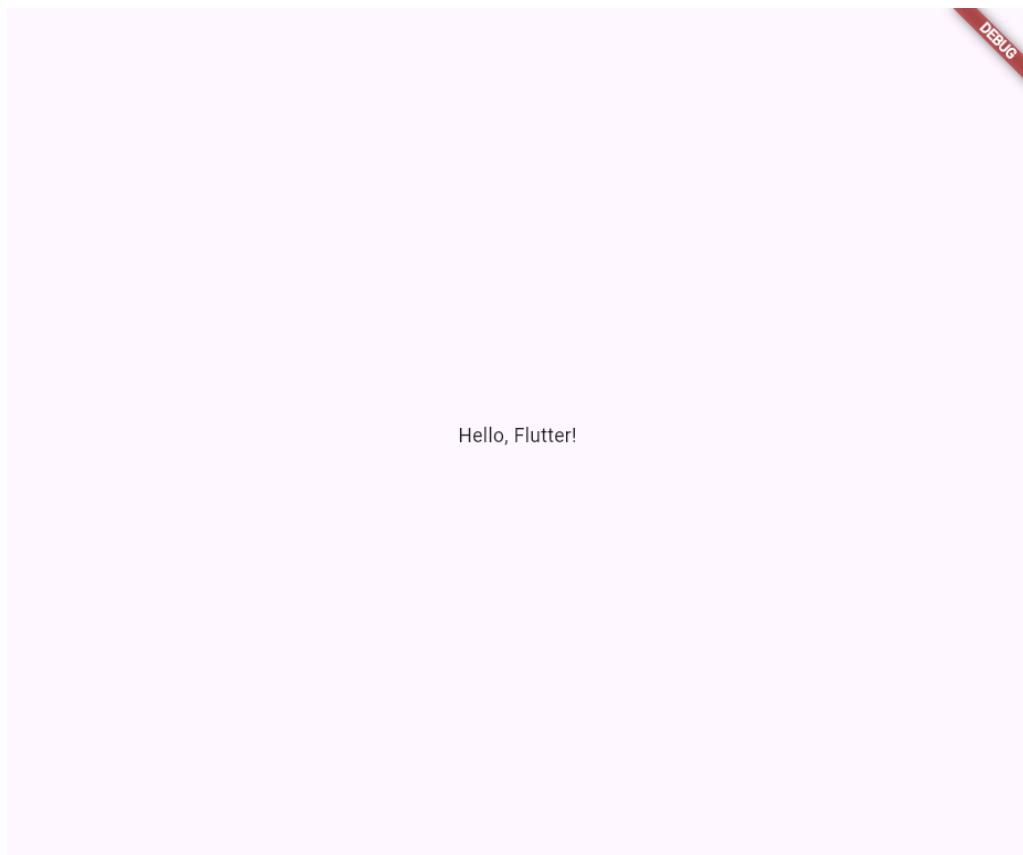
Flutter sử dụng ngôn ngữ lập trình **Dart** làm nền tảng. **Dart** cung cấp cú pháp hướng đối tượng gọn gàng cùng hai chế độ biên dịch hiện đại—*ahead-of-time* (AOT) và *just-in-time* (JIT). Các chế độ biên dịch này vừa tối ưu hiệu năng trên thiết bị thật, vừa hỗ trợ *hot reload* để phản hồi tức thời trong giai đoạn phát triển. Bên cạnh đó, cơ chế *Null Safety* của Dart loại trừ lỗi giá trị `null` ngay khi biên dịch, qua đó nâng độ tin cậy của ứng dụng Flutter.

Nhờ các đặc điểm trên mà Flutter có thể xây dựng được một ứng dụng chất lượng cao, hiệu năng tốt và giao diện đẹp. Flutter vượt trội so với các nền tảng khác trong việc kiểm soát giao diện, nhất quán giữa các nền tảng.

Ví dụ, một đoạn mã Flutter cơ bản trong Dart có thể trông như sau:

```
import 'package:flutter/material.dart';
void main() {
    runApp(const MyApp());
}
class MyApp extends StatelessWidget {
    const MyApp({Key? key}): super(key: key);
    @override
    Widget build(BuildContext context) {
        return const MaterialApp(
            home: Scaffold(
                body: Center(child: Text('Hello, Flutter!')),
            ),
        );
    }
}
```

Đoạn mã này hiển thị dòng chữ "Hello, Flutter!" ở giữa màn hình:



Hình 5.4: Chương trình minh họa in ra chữ "*Hello, Flutter!*"

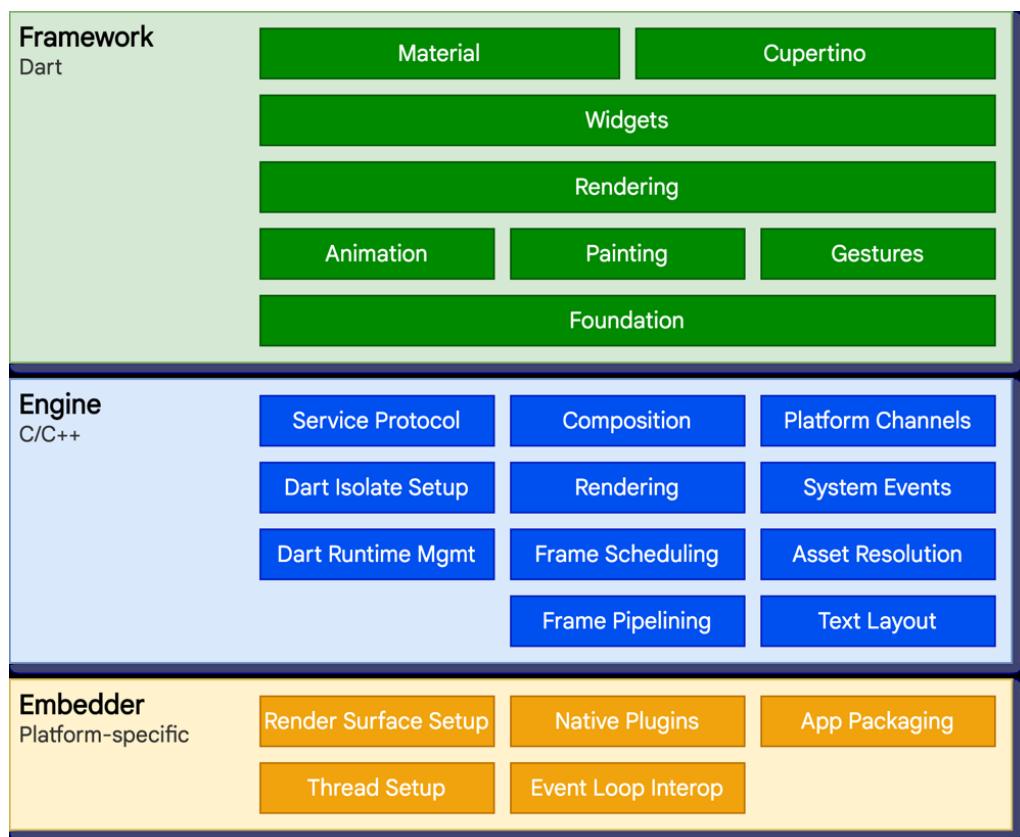
5.2 Kiến trúc của Flutter

Flutter áp dụng mô hình **kiến trúc phân lớp** nhằm tối ưu hóa khả năng đa nền tảng, hiệu năng và tính mở rộng. Framework này được xây dựng như một hệ thống các thư viện độc lập, trong đó mỗi thư viện chỉ phụ thuộc vào lớp bên dưới. Không có lớp nào có đặc quyền truy cập vào các lớp khác, và mỗi phần của framework được thiết kế để có thể tùy chọn và thay thế. Cấu trúc phân lớp này cho phép các nhà phát triển dễ dàng thêm các tính năng mới hoặc tùy chỉnh các thành phần hiện có để đáp ứng nhu cầu cụ thể của ứng dụng. Mỗi lớp trong kiến trúc Flutter được thiết kế độc lập, giúp giảm thiểu sự phụ thuộc giữa các thành phần, từ đó nâng cao khả năng bảo trì và mở rộng của ứng dụng.

Một khái niệm quan trọng của **Flutter framework** đó là các thành phần sẽ được nhóm lại theo độ phức tạp và được sắp xếp rõ ràng trong các tầng có độ phức tạp giảm dần. Một **layer** (tầng, lớp) được tạo thành bằng việc sử dụng các class tiếp

theo ngay cạnh nó. Top của tất cả các layer là các widget đặc biệt cho Android và iOS. Layer tiếp theo là widget gốc của flutter. Tiếp nữa là *Rendering layer*, đây là level thấp nhất trong việc sinh các thành phần của flutter app. Layer tiếp theo là nền tảng gốc hệ điều hành.

Sơ đồ kiến trúc tổng thể của Flutter được chia thành ba tầng chính: tầng **Framework** (viết bằng Dart), tầng **Engine** (viết bằng C/C++) và tầng **Embedder** (giao tiếp với hệ điều hành)



Hình 5.5: Kiến trúc tổng thể của một ứng dụng Flutter

- **Tầng Framework** cung cấp các thư viện Dart cấp cao để xây dựng giao diện người dùng, bao gồm các thư viện cơ sở như foundation, các dịch vụ nền tảng như animation, painting, gestures và các lớp rendering, widgets, cũng như thư viện giao diện Material và Cupertino.
- **Tầng Engine** là lõi của Flutter (hầu hết viết bằng C++), chịu trách nhiệm các tác vụ cấp thấp như raster hóa cảnh thông qua Skia, bộ cục văn bản, xử lý I/O và kiến trúc plugin, đồng thời quản lý môi trường thực thi Dart.

- **Tầng Embedder** chứa mã gốc riêng theo nền tảng (như Java/Kotlin cho Android, Swift/Objective-C cho iOS) để khởi tạo ứng dụng Flutter, thiết lập bề mặt vẽ đồ họa, vòng lặp sự kiện và tích hợp với các dịch vụ hệ điều hành như nhập liệu và truy cập phần cứng

5.2.1 Tầng Framework

Viết hoàn toàn bằng Dart, tầng Framework nằm phía trên cùng, cung cấp API cao cấp để xây dựng và quản lý giao diện theo mô hình phản ứng.



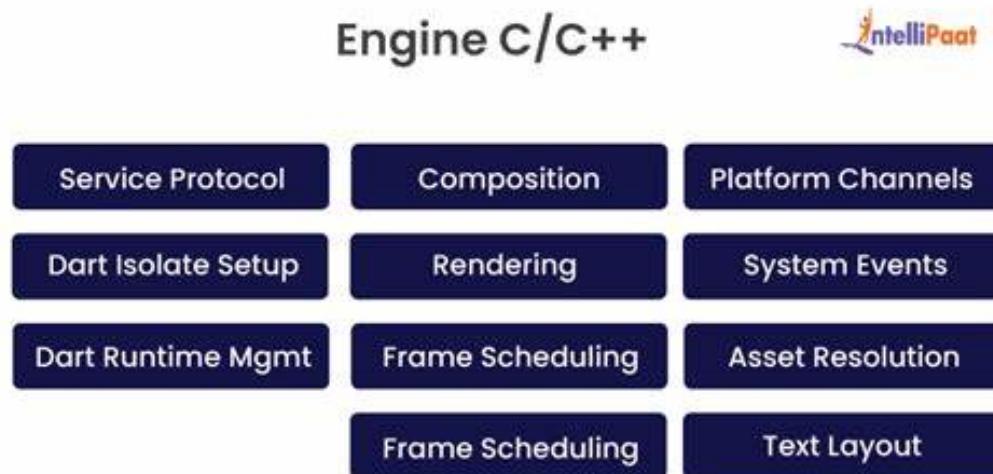
Hình 5.6: Chi tiết về lớp Framework

- **Foundation:** Cung cấp các lớp cơ bản và công cụ như error handling, assertions, sử dụng ngôn ngữ thiết kế Material (Android) hoặc Cupertino (iOS).
- **Animation/ Painting/ Gesture:** Thực hiện hoá các chuyển động trong Flutter (Animation), hỗ trợ xây dựng giao diện (Painting), nhận dạng tương tác cử chỉ của người dùng.
- **Rendering Layer:** Quản lý layout và painting, sử dụng **RenderObject** để xác định cách widgets được hiển thị. Lớp này cung cấp abstraction để xử lý layout, cho phép xây dựng cây các đối tượng renderable, và tự động cập nhật khi có thay đổi.
- **Widgets:** Là thành phần cơ bản để xây dựng giao diện trong Flutter, sử dụng kiến trúc cây (widget tree). Nó được xây dựng theo dạng cây với những object là lớp kế thừa của **RenderObject**. Widgets nhận và lưu trữ trạng thái (state)

của UI, và sẽ rebuild nếu state thay đổi.

- **Material/Cupertino:** Được xây dựng từ Widgets library để triển khai ngôn ngữ thiết kế Material (Android) và iOS (Cupertino). Điều này giúp ứng dụng có giao diện native, phù hợp với từng nền tảng

5.2.2 Tầng Engine (C++):



Hình 5.7: Kiến trúc chi tiết lớp Engine

Tầng Engine là lõi của Flutter, chủ yếu được viết bằng C++ để đảm bảo hiệu suất cao, đặc biệt phù hợp cho các tác vụ cần truy cập trực tiếp vào phần cứng. Nó chịu trách nhiệm xử lý các tác vụ cấp thấp bao gồm:

- **Rasterization:** Quá trình chuyển đổi đồ họa vector thành pixel để hiển thị trên màn hình, sử dụng Impeller trên iOS, Android, desktop (hiện đang thử nghiệm, sau cờ), và Skia trên các nền tảng khác. Skia là thư viện đồ họa 2D nổi tiếng, được Google tích hợp trong Chrome và Android, đảm bảo hiệu suất render tối ưu.
- **Core APIs:** Cung cấp triển khai cho các API cốt lõi:
 - **Graphics:** Không chỉ bao gồm rasterization mà còn xử lý các lệnh vẽ, quản lý texture, và các tác vụ đồ họa khác.
 - **Text layout:** Tính toán cách hiển thị văn bản, bao gồm lựa chọn font, kích

thuộc, và định dạng văn bản trên giao diện.

- **File and network I/O:** Hỗ trợ ứng dụng đọc ghi file và thực hiện các yêu cầu mạng, rất cần thiết cho những ứng dụng đòi hỏi kết nối internet.
 - **Trợ năng (Accessibility):** Đảm bảo ứng dụng phù hợp cho mọi người dùng, bao gồm cả người khuyết tật, thông qua tích hợp với các công cụ như screen reader, đáp ứng tiêu chuẩn accessibility.
 - **Plugin architecture:** Cho phép sử dụng mã native thông qua plugin, mở rộng chức năng ứng dụng như tích hợp camera, GPS, hoặc trình phát video.
 - **Dart runtime:** Cung cấp máy ảo Dart để thực thi mã nguồn, hỗ trợ hiệu suất runtime với JIT (Just-In-Time) trong quá trình debug và AOT trong bản release.
 - **Compilation toolchain:** Hỗ trợ công cụ biên dịch (AOT compilation), chuyển mã Dart thành mã native nhằm cải thiện tốc độ thực thi, đặc biệt trên các thiết bị di động.
- Đóng vai trò trung gian giữa mã nguồn viết bằng Dart và thiết bị phần cứng (hoặc phần mềm bên ngoài ứng dụng).
 - Thực thi các đoạn mã đã được thông dịch hoặc biên dịch, đồng thời cung cấp các hệ thống runtime như garbage collector và các thư viện cần thiết của ngôn ngữ.

5.2.3 Tầng *Embedder*



Hình 5.8: Kiến trúc chi tiết lớp Embedder

Tầng *Embedder* tạo cầu nối trực tiếp giữa *Flutter Engine* và hệ điều hành mục tiêu. Cầu nối này được hiện thực khác nhau trên từng nền tảng: ở Android, mã Java hoặc Kotlin nhúng nội dung Flutter vào *Activity*; ở iOS, mã Objective-C hoặc Swift kết xuất giao diện lên *UIViewController*; còn trên Web, mã JavaScript gắn kết engine với *HTML Canvas*.

Nhờ cấu trúc chuyên biệt ấy, tầng *Embedder* đồng thời khởi tạo engine và ngữ cảnh đồ họa, điều phối vòng lặp sự kiện cùng các luồng nền tảng, cũng như trung gian trao đổi dữ liệu giữa Flutter và hệ thống nhập liệu, cảm biến, camera, microphone, GPS cùng mọi dịch vụ bản địa khác.

5.3 Ứng dụng Flutter đầu tiên

Với khả năng cross-platform, Flutter cho phép các nhà phát triển tạo ra các ứng dụng hiệu suất cao, sử dụng ngôn ngữ Dart, một ngôn ngữ hiện đại, dễ học và hỗ trợ lập trình hướng đối tượng cũng như lập trình chức năng. Để phát triển ứng dụng Flutter, Android Studio là một trong những công cụ lý tưởng nhất.



Hình 5.9: Công cụ Android Studio

Android Studio vốn là môi trường phát triển tích hợp (IDE) chính thức cho ứng dụng Android, nhưng nó cũng được tích hợp chặt chẽ để hỗ trợ Flutter. Nhờ nền tảng mạnh mẽ từ IntelliJ IDEA, Android Studio cung cấp các tính năng vượt trội như:

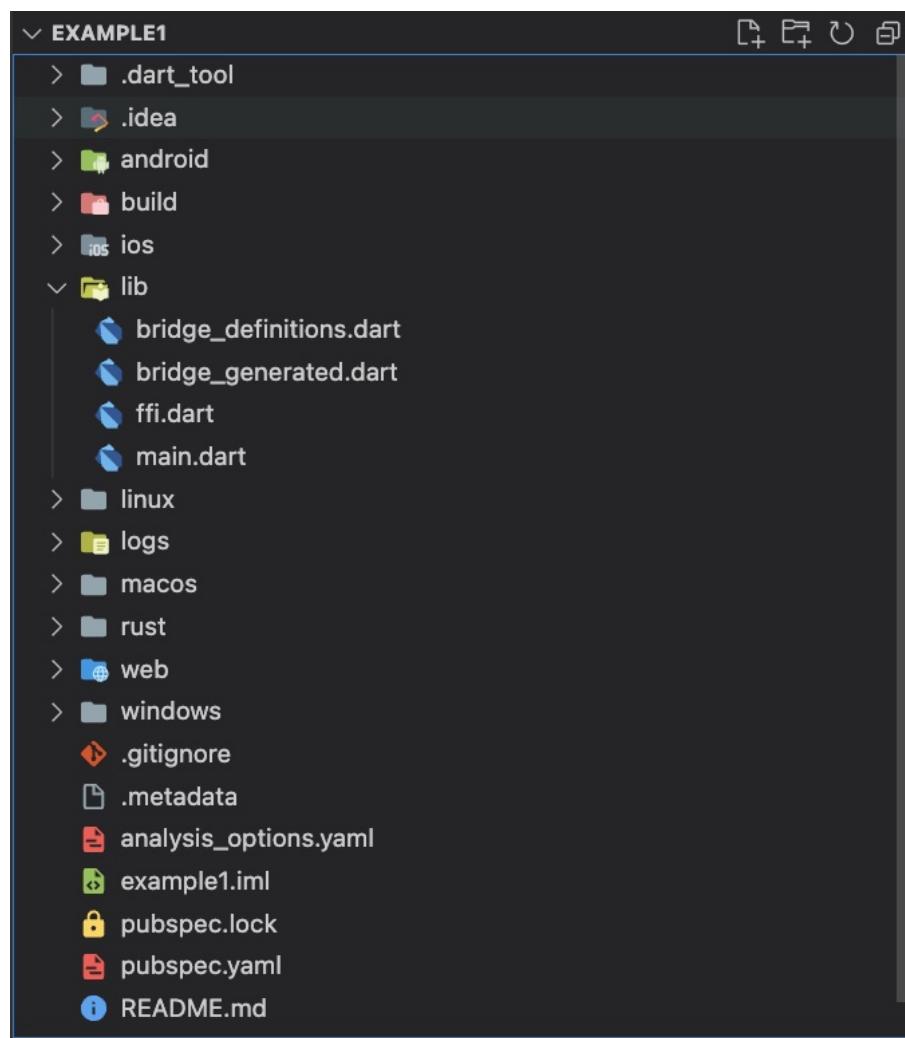
- **Flutter Inspector:** Giúp kiểm tra và phân tích cấu trúc giao diện người dùng (widget) của ứng dụng.
- **Hot Reload:** Cho phép xem thay đổi mã nguồn ngay lập tức mà không cần khởi động lại ứng dụng, tăng tốc quá trình phát triển.
- **Công cụ debug:** Hỗ trợ tìm và sửa lỗi một cách hiệu quả.

Những tính năng này giúp nâng cao năng suất, đặc biệt khi bắt đầu xây dựng ứng dụng Flutter đầu tiên.

Cấu trúc của một dự án Flutter: Để tạo một dự án Flutter mới trong Android Studio, ta có thể chọn menu *File > New > New Flutter Project* hoặc sử dụng lệnh CLI trong terminal.

```
flutter create <project_name>
```

Lệnh trên sẽ sinh ra một cấu trúc dự án cơ bản với một tệp pubspec.yaml và các thư mục cần thiết. Khi build lần đầu, Flutter cũng tạo ra tệp pubspec.lock để khóa phiên bản các gói phụ thuộc, đảm bảo tính tái lập của dự án. Cấu trúc dự án không chỉ giúp quản lý mã nguồn và tài nguyên một cách khoa học mà còn là nền tảng để ứng dụng hoạt động trơn tru trên các nền tảng khác nhau. Một dự án Flutter điển hình có cấu trúc thư mục như bên dưới:



Hình 5.10: Cấu trúc của một dự án Flutter

Các thư mục và tệp chính bao gồm:

- **android:** Thư mục được sinh tự động, chứa mã và cấu hình dành riêng cho nền tảng Android, bao gồm các tệp **AndroidManifest**, các script **Gradle**, file cấu hình ứng dụng. Đây là nơi Flutter tích hợp với hệ sinh thái Android.
- **ios:** Thư mục được sinh tự động, chứa mã và cấu hình dành riêng cho nền tảng

iOS, bao gồm dự án **Xcode**, các tệp **plist** (Info.plist), và tài nguyên liên quan đến iOS.

- **lib:** Thư mục chứa mã Dart của ứng dụng Flutter. Tại đây thường có tệp `main.dart` làm điểm vào (entry point) của ứng dụng, và có thể chia nhỏ thành các thư mục con để quản lý mã nguồn (widgets, models, services, v.v.), đây là thư mục mà lập trình viên làm việc nhiều nhất.
- **test:** Folder chứa Dart code, gồm các bài kiểm thử (unit tests, widget tests, integration tests) cho ứng dụng. Mặc định có một số tệp ví dụ và bài kiểm thử mẫu. Thư mục này giúp tổ chức mã kiểm thử riêng biệt với mã ứng dụng.
- **assets (Có thể có):** Chứa các tài nguyên tĩnh của ứng dụng như hình ảnh, font chữ, hoặc file dữ liệu. Flutter cung cấp hệ thống quản lý tài nguyên qua `pubspec.yaml`, cho phép truy cập các tài nguyên này trong mã Dart.
- **.gitignore:** Git version control file - Đây là file chứa cấu hình cho project git.
- **.metadata:** Sinh tự động bởi flutter tools.
- **.packages:** Được sinh tự động để theo dõi flutter packages.
- **pubspec.yaml:** Là tệp cấu hình chính (định dạng YAML) của dự án Flutter. Đóng vai trò như "trái tim", tại đây khai báo metadata của dự án (tên, phiên bản, mô tả), tài nguyên(hình ảnh, font,...) và danh sách các phụ thuộc (packages) cần thiết. Ví dụ, dự án mặc định sẽ có Flutter SDK và gói `cupertino_icons` được liệt kê trong `dependencies`, cùng các phụ thuộc dev.

Ngoài ra, tùy vào cấu hình và mục tiêu phát triển, dự án có thể có thêm các thư mục như **web/** (nếu bật hỗ trợ Web), **linux/**, **windows/**, **macos/** (cho desktop), hoặc thư mục **build/** chứa các đầu ra biên dịch.

Phân tích mã nguồn lib/main.dart của ứng dụng đầu tiên:

Dưới đây là ví dụ mã nguồn trong file `lib/main.dart` cho ứng dụng Flutter

cơ bản kiểu “Hello World”:

```
import 'package:flutter/material.dart';
void main() {
    runApp(const MyApp());
}

class MyApp extends StatelessWidget {
    const MyApp({super.key});

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Welcome to Flutter',
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Welcome to Flutter'),
                ),
                body: const Center(
                    child: Text('Hello World'),
                ),
            ),
        );
    }
}
```

Kết quả minh họa của đoạn mã nguồn trên:



Hình 5.11: Kết quả của chương trình minh họa

Phân tích chi tiết từng thành phần trong đoạn mã trên:

- `import 'package:flutter/material.dart';` : Dòng này import thư viện Material Design của Flutter, cho phép sử dụng các widget chuẩn phong cách Material như MaterialApp, Scaffold, AppBar, Center, Text... mà ứng dụng cần. Thư viện này cung cấp các thành phần UI sẵn có để xây dựng giao diện theo phong cách Material Design (của Google).
- `void main() . . . :` Điểm khởi đầu của ứng dụng Flutter là hàm main(). Phương thức runApp() được gọi và truyền vào đối tượng của lớp **MyApp**. Mục đích của phương thức runApp() là để đưa giao diện Widget vào hiển thị trên màn hình.
- `class MyApp extends StatelessWidget :` Định nghĩa lớp MyApp kế thừa từ StatelessWidget. Widget được sử dụng để tạo UI(giao diện

người dùng). StatelessWidget là loại widget không có trạng thái thay đổi (immutable), nghĩa là nó xây dựng giao diện dựa vào thông tin cố định và sẽ không tự tái tạo khi có sự kiện (nếu cần trạng thái thay đổi ta dùng StatefulWidget thay thế). Do đó, MyApp là thành phần gốc đơn giản của ứng dụng, chịu trách nhiệm trả về widget con trong phương thức build().

- Widget build (BuildContext context): **MyApp** ghi đè phương thức build(). Mục đích của phương thức build() là tạo một phần UI cho ứng dụng. Đây là nơi định nghĩa cây widget mà MyApp xây dựng. Phương thức trả về một widget, trong trường hợp này là một MaterialApp.
- return MaterialApp(...): MaterialApp là widget cấp cao dùng cho ứng dụng theo chuẩn Material Design. Nó bao gồm nhiều cấu hình và widget con cần thiết (như điều hướng, theme, locale), và thường là widget gốc trong các ứng dụng Material. Việc sử dụng MaterialApp giúp dễ dàng tạo giao diện phong cách Material. Trong ví dụ trên, MaterialApp được cấu hình với thuộc tính title (tiêu đề ứng dụng) và home (widget trang chính).
 - title: 'Welcome to Flutter': Dùng để thiết lập tiêu đề ứng dụng. Trên Android, tiêu đề này xuất hiện ở màn hình đa nhiệm (Recent Apps), và trên web nó hiển thị trong tab trình duyệt, còn ở iOS thì giá trị này không được sử dụng để hiển thị vì iOS lấy tên ứng dụng từ file cấu hình Info.plist.
 - home: Scaffold(...): Thuộc tính home của MaterialApp quy định trang đầu tiên (route mặc định '/') của ứng dụng.
- Scaffold: Widget Scaffold cung cấp khung bối cảnh cơ bản cho ứng dụng theo Material Design. Nó tự động thiết lập vùng hiển thị, bao gồm thanh AppBar ở trên, nội dung trong thân, ... Trong ví dụ, Scaffold chứa hai phần chính:
 - appBar: AppBar(...): Thiết lập thanh ứng dụng (App Bar) nằm ở

dầu màn hình. Đây là một widget dạng toolbar tiêu chuẩn, có thể chứa tiêu đề, biểu tượng, các nút hành động, v.v. Ở đây, AppBar được tạo với tiêu đề là một widget Text hiển thị chuỗi “Welcome to Flutter”.

- body: Center(child: Text('Hello World')): Đây là phần thân của Scaffold. Ở ví dụ này, body là một widget Center, nhằm căn giữa nội dung con của nó. Bên trong Center, có một widget Text để hiển thị văn bản “Hello World”.
- Các thành phần con khác:
 - const MyApp(super.key;) : Định nghĩa constructor hằng (const constructor) cho lớp MyApp. Sử dụng const giúp Flutter tối ưu hóa xây dựng widget (có thể tạo các instance hằng) khi tham số không đổi. Tham số key được truyền lên lớp cha để quản lý trong cây widget.
 - @override: Ghi chú override phương thức build. Từ khoá này giúp kiểm tra rằng lớp con đang ghi đè đúng phương thức của lớp cha.
 - ThemeData(primarySwatch: Colors.blue): Thiết lập chủ đề (theme) cho ứng dụng, ví dụ màu chính của Material. (Ở ví dụ trên, ta không đặt theme riêng, nên màu mặc định sẽ là màu trắng)

5.4 Flutter widgets

5.4.1 Giới thiệu khái niệm Widget trong Flutter

Trong Flutter, giao diện người dùng được xây dựng hoàn toàn từ các *widget*. Widget là thành phần cơ bản nhất tạo nên toàn bộ giao diện người dùng của ứng dụng Flutter. Mỗi widget là một lớp định nghĩa một phần cấu hình giao diện. Một widget biểu diễn cách mà màn hình nên hiển thị trong một trạng thái nhất định. Hầu hết mọi thứ nhìn thấy như hình ảnh, biểu tượng, văn bản đều là widget; thậm chí các thành phần dùng để bố cục như Row, Column, Container cũng là widget.

Bằng cách kết hợp (compose) các widget đơn giản, Flutter cho phép tạo ra các

giao diện phức tạp một cách linh hoạt. Ứng dụng là một top-level widget bao gồm một hoặc nhiều widget con. Trong bản thân mỗi widget lại có thể chứa một hoặc nhiều widget con khác.

Các widget do người dùng định nghĩa sẽ là lớp con của StatelessWidget hoặc StatefulWidget tùy vào việc widget đó có cần lưu trữ trạng thái hay không

5.4.2 Phân loại widgets trong Flutter

Flutter cung cấp nhiều loại widget khác nhau tùy mục đích sử dụng. Widgets có thể là:

- **Một phần tử cấu trúc:** Đây là những widgets được sử dụng để xây dựng các thành phần giao diện cơ bản (nút, menu, appBar,...). Các widget này giúp tạo ra các yếu tố tương tác trong giao diện người dùng.
- **Một phần tử kiểu dáng (styling elements):** Flutter hỗ trợ các widget và thuộc tính giúp kiểm soát kiểu dáng toàn cục hoặc cục bộ, bao gồm:
 - Theme và ThemeData: Dùng để định nghĩa giao diện tổng thể của toàn bộ ứng dụng như màu sắc, font, kích thước chữ, v.v.
 - TextStyle: Thiết lập kiểu chữ mặc định cho tất cả các widget con.
 - TextStyle: Dùng để định nghĩa cụ thể font chữ, kích thước, màu sắc cho các thành phần văn bản.
 - Padding, Margin, BoxDecoration: Dùng để thiết lập khoảng cách và hiệu ứng đồ họa.

Việc sử dụng đúng các widget kiểu dáng giúp đảm bảo tính thống nhất, dễ bảo trì và tái sử dụng trong toàn bộ ứng dụng.

- **Một đặc trưng của Layout:** Ví dụ như Padding, Align, Center, Margin, Theme. Các widget này không trực tiếp hiển thị dữ liệu nhưng có

tác dụng điều chỉnh vị trí, khoảng cách, căn chỉnh hoặc kiểu dáng của các widget con.

Ví dụ: Đoạn mã dưới đây minh họa việc tạo một widget tùy chỉnh có áp dụng kiểu dáng bằng cách sử dụng Padding và TextStyle:

```
class PaddedText extends StatelessWidget {
    final String _data;

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

    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: const EdgeInsets.all(12.0),
            child: Text(
                _data,
                style: const TextStyle(
                    fontSize: 18,
                    color: Colors.blue,
                    fontWeight: FontWeight.bold,
                ),
            ),
        );
    }
}
```

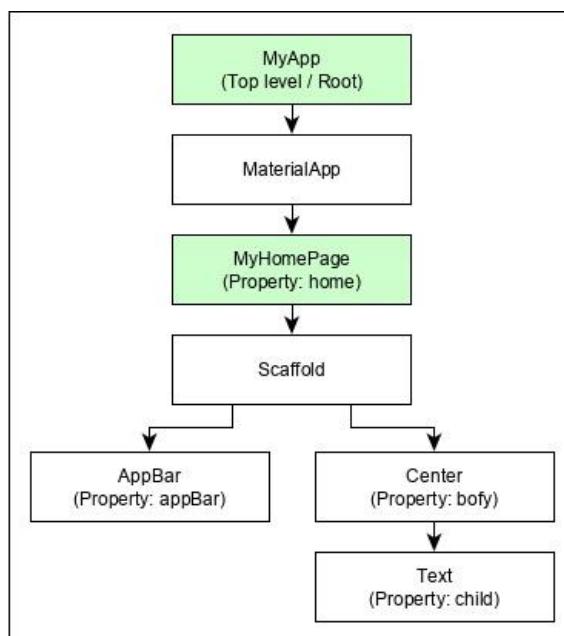
Ở chương trình ví dụ trên:

- Padding: Là phần tử layout, dùng để tạo khoảng cách xung quanh widget con của nó, trong ví dụ này, Padding tạo ra khoảng cách 12.0 pixel ở tất cả các hướng xung quanh widget Text
- Text: Là phần tử cấu trúc, dùng để hiển thị văn bản là biến `_data`
- TextStyle: Là một phần tử kiểu dáng, dùng để áp dụng kiểu dáng cho văn bản: màu xanh dương, in đậm, cỡ chữ 18.

Sơ đồ cấu trúc Widget:

Cấu trúc widget của ứng dụng Hello World thể hiện mối quan hệ cha con chặt chẽ. Mỗi widget cha cung cấp ngữ cảnh hoặc thuộc tính cho widget con của nó. **MaterialApp** cung cấp `home` cho **MyHomePage**, **Scaffold** cung cấp `appBar` và `body` cho **AppBar** và **Center**, **Center** cung cấp `child` cho **Text**.

Flutter render UI bằng cách duyệt qua cây widget từ gốc (**MyApp**) đến lá (**Text**), tạo ra một cây render (render tree) dựa trên các widget này.



Hình 5.12: Cấu trúc widget của ứng dụng *Hello World*

5.4.3 Một số widgets phổ biến trong Flutter

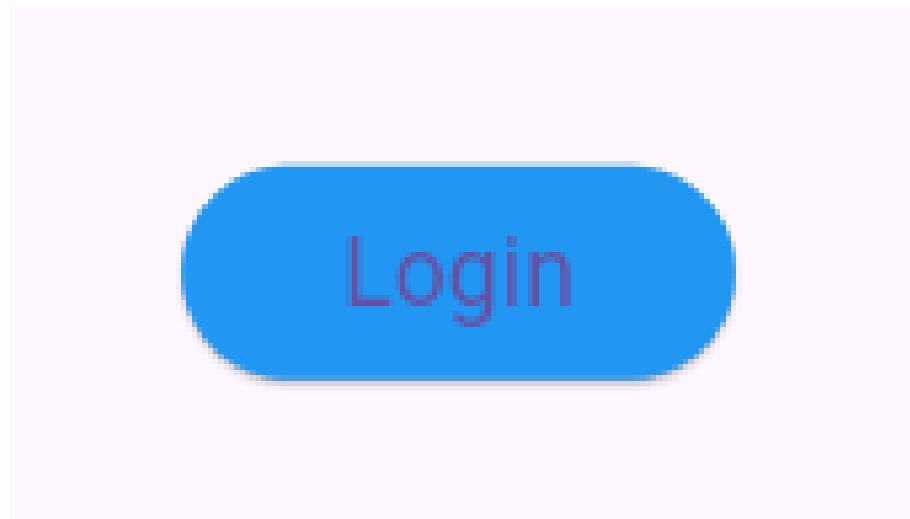
Đây là các thành phần cơ bản giúp tạo cấu trúc ứng dụng như nút, biểu tượng, trường nhập liệu, menu...

- **Button:** Buttons cung cấp cho người dùng khả năng thực hiện các hành động, đưa ra lựa chọn, gửi biểu mẫu, lưu dữ liệu, mở trang mới,... bằng cách click vào nó. Nút được phân loại thành nhiều loại: **Elevated Button**, **Floating Action Button**, **Outlined Button**, **Icon Button**, **Text Button**,...

Ví dụ:

```
ElevatedButton(  
    onPressed: () {  
        print('Clicked');  
    },  
    style: ElevatedButton.styleFrom(  
        backgroundColor: Colors.blue,  
    ),  
    child: Text('Login'),  
,
```

Hình ảnh minh họa:



Hình 5.13: Elevated Button trong Flutter

Trong mỗi Widget lại có các properties riêng biệt, giúp cho bản thân nó trở nên sinh động hơn khi hiển thị ra giao diện. Dưới đây là ví dụ về một số thuộc tính điển hình của widget **ElevatedButton**:

Thuộc tính	Mô tả thuộc tính
autofocus	Nhận vào giá trị boolean xác định nút có được focus mặc định khi hiển thị hay không
clipBehaviour	Xác định nội dung của nút có bị cắt (clip) nếu vượt quá kích thước không
focusNode	Đại diện cho node focus của widget
ButtonStyle	Xác định kiểu hiển thị (style) của nút
onLongPress	Hành động sẽ thực hiện khi người dùng nhấn giữ nút
enabled	Nhận vào giá trị boolean xác định nút có hoạt động hay không
hashCode	Xác định mã băm (hashCode) của nút
Key	Điều khiển cách một widget thay thế widget khác trong cây widget
onFocusChanged	Hàm sẽ được gọi khi focus của nút thay đổi
onHover	Hành động được thực hiện khi người dùng di chuột qua nút

Bảng 5.1: Các thuộc tính của ElevatedButton trong Flutter

- **TextField:** Cho phép người dùng nhập văn bản. Được sử dụng rộng rãi trong các biểu mẫu đăng nhập, đăng ký, tìm kiếm, bình luận,...

Ví dụ:

```
TextField(
    decoration: InputDecoration(
        border: OutlineInputBorder(),
        labelText: 'Họ tên',
        hintText: 'Nhập họ và tên',
        prefixIcon: Icon(Icons.person),
    ),
),
```

Trong ví dụ trên, màn hình hiển thị một trường nhập liệu có biểu tượng Icons.email, nhãn “Email”, khung viền và gợi ý nhập liệu.

Hình ảnh minh họa:



Hình 5.14: Textfield trong Flutter

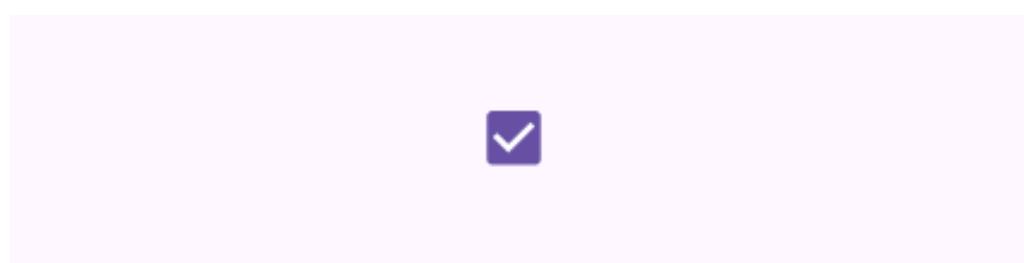
- **Checkbox:** Widget này cho phép người dùng chọn hoặc bỏ chọn một tùy chọn.

Thường được sử dụng trong các biểu mẫu hoặc danh sách tùy chọn.

Ví dụ:

```
Checkbox(  
  value: isChecked,  
  onChanged: (bool? newValue) {  
    setState(() {  
      isChecked = newValue!;  
    });  
  },  
)
```

Trong ví dụ trên, màn hình hiển thị một ô checkbox, value cho biết trạng thái hiện tại của checkbox (đã được chọn hay chưa), khi thay đổi trạng thái thì hàm onChanged sẽ được gọi.



Hình 5.15: Checkbox trong Flutter

- **Text:** Widget Text là một thành phần cơ bản trong Flutter, được sử dụng để hiển thị văn bản trên giao diện người dùng

Ví dụ:

```
Text(  
'Hello world!',
```

```
        style: TextStyle(  
            color: Colors.black,  
            fontSize: 40,  
            backgroundColor: Colors.white,  
            fontWeight: FontWeight.bold,  
) ,  
)
```

Trong ví dụ trên, tham số đầu tiên là một chuỗi văn bản, đây là nội dung văn bản sẽ được hiển thị trên màn hình, tham số style để định dạng kiểu của văn bản. Văn bản hiển thị sẽ có kích cỡ chữ 40 (logical pixels), màu đen, màu nền văn bản màu trắng, độ đậm bold.



Hello world!

Hình 5.16: Minh họa về widget Text

- **Row:** Dùng để sắp xếp các widget con theo chiều ngang, thường được sử dụng để hiển thị các thành phần nằm cạnh nhau. Row có tham số bắt buộc là children để chứa các widget con, thứ tự đặt vào trong children sẽ ảnh hưởng tới thứ tự các widget đó xuất hiện trên màn hình.

Ví dụ:

```
Row (  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: const [  
        Text ('Hello'),  
        Text ('Flutter!'),  
        Text ('!!!'),  
    ],  
)
```

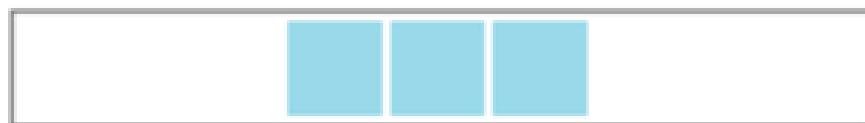
Kết quả minh họa của chương trình trên:



Hình 5.17: Minh họa về widget Row

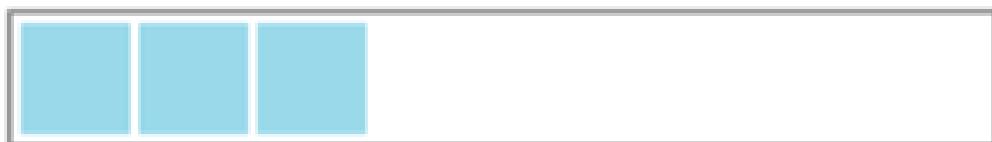
Bên cạnh đó, tham số mainAxisAlignment sẽ ảnh hưởng đến vị trí hiển thị của các widget trong children:

- MainAxisAlignment.center: Các items sẽ nằm ở giữa hàng



Hình 5.18: Thuộc tính *mainAxisAlignment* có giá trị *center*

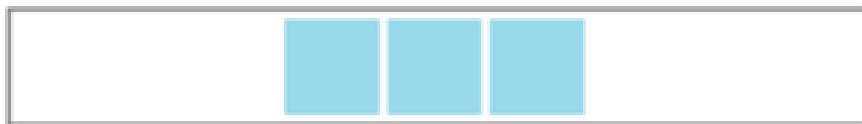
- MainAxisAlignment.start: Các items sẽ nằm ở vị trí bắt đầu của hàng đó.



Hình 5.19: Thuộc tính *mainAxisAlignment* có giá trị *start*

Tương tự, nếu mainAxisAlignment là end thì các phần tử sẽ được sắp xếp ở cuối hàng đó.

- MainAxisAlignment.spaceBetween: Các items cách đều nhau, phần tử đầu và cuối sẽ ở 2 lề của Row đó.



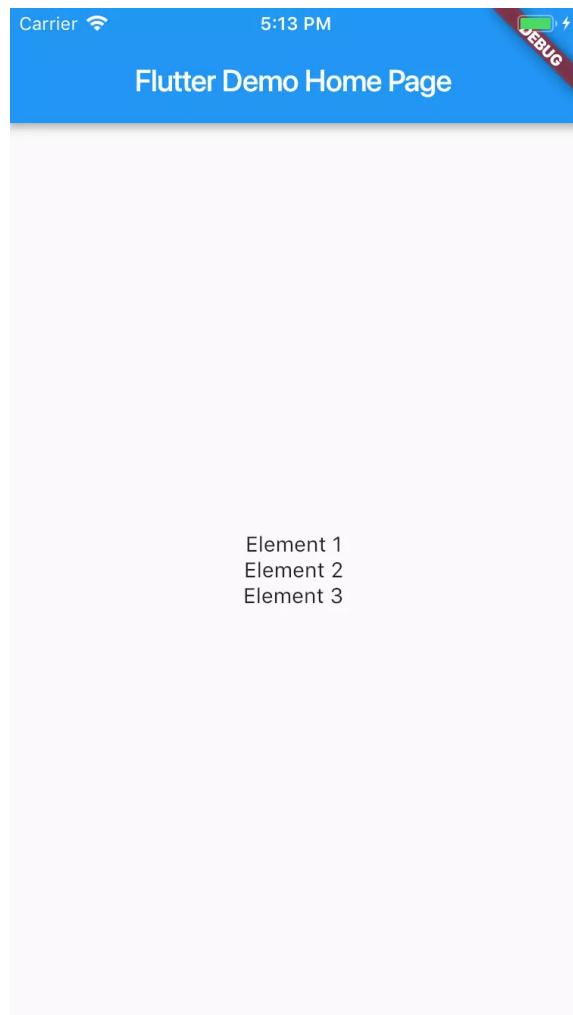
Hình 5.20: Thuộc tính *mainAxisAlignment* có giá trị *spaceBetween*

- **Column:** Column là một widget trong thư viện widgets của Flutter, thuộc nhóm layout widgets, giúp sắp xếp các widget con theo một ngăn xếp dọc. Trục chính (main axis) của Column là trục dọc (vertical), nghĩa là các widget con được xếp từ trên xuống dưới. Trục phụ (cross axis) là trục ngang (horizontal), dùng để căn chỉnh các widget con theo chiều ngang.

Ví dụ:

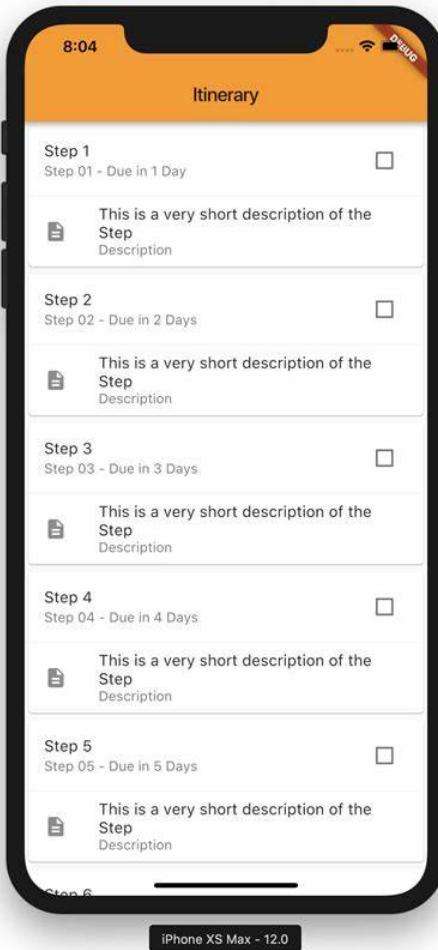
```
Column(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: <Widget>[  
    Text("Element 1"),  
    Text("Element 2"),  
    Text("Element 3"),  
  ],  
) ,
```

Column gồm 3 widget Text bên trong nó và mainAxisAlignment được đặt thành center, có nghĩa là các widget con được căn giữa theo chiều dọc.



Hình 5.21: Minh họa về widget Column

- **ListView:** Widget này cho phép hiển thị danh sách các widget con có thể cuộn được theo chiều dọc hoặc ngang, thường được sử dụng để hiển thị danh sách dài các mục.



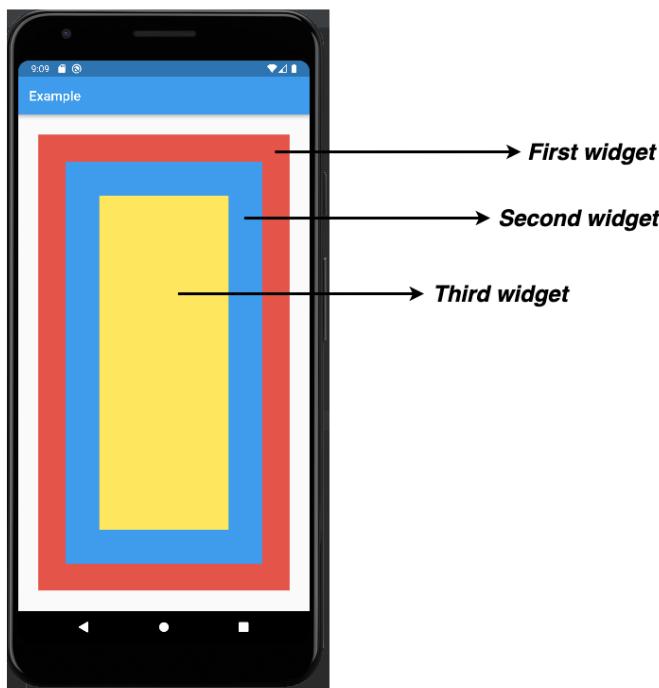
Hình 5.22: Minh họa về widget ListView

Ví dụ:

```
ListView(
    children: <Widget>[
        ListTile(title: Text('Element 1')),
        ListTile(title: Text('Element 2')),
        ListTile(title: Text('Element 3')),
    ],
)
```

Trong ví dụ trên, ListView là widget danh sách cuộn, gồm 3 widget con là ListTile hiển thị nội dung văn bản.

- **Stack:** Cho phép chồng các widget con lên nhau, thường được sử dụng để tạo giao diện phức tạp với các thành phần chồng lấp.



Hình 5.23: Minh họa về widget Stack

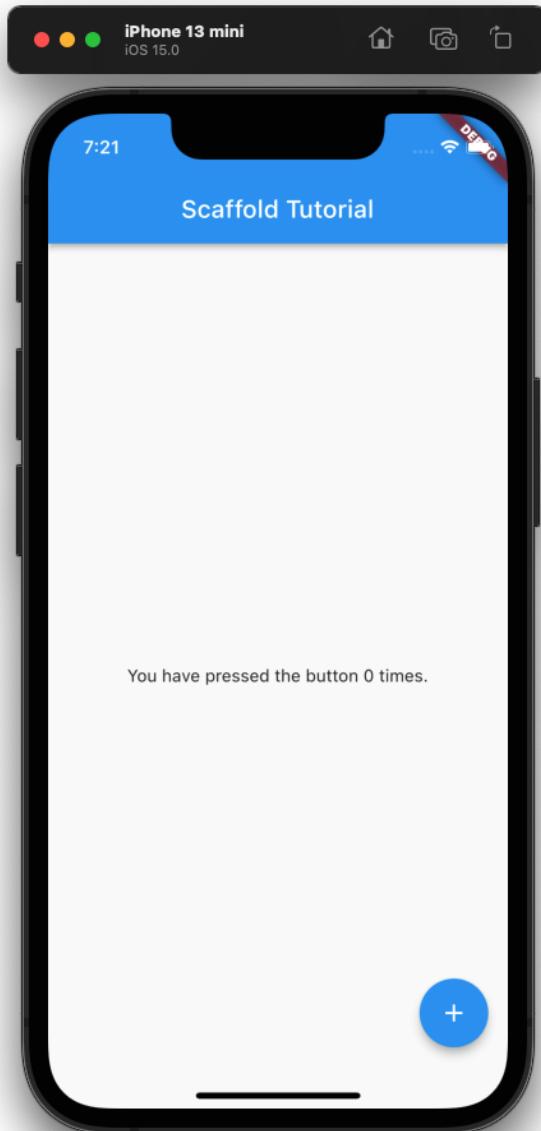
Ví dụ:

```
Stack(
  children: <Widget>[
    Container(width:90, height:90, color:Colors.red),
    Positioned(
      top: 10,
      left: 10,
      child: Container(
        width: 80,
        height: 80,
        color: Colors.green
      ),
    ),
  ],
)
```

Kết quả của đoạn mã là một giao diện hiển thị một hình vuông màu đỏ 90x90 pixel với một hình vuông màu xanh lá 80x80 pixel được định vị cách mép trái và mép trên 10 pixel, trong đó hình vuông màu xanh chồng lấp lên trên hình

vuông đở.

- **Scaffold:** Cung cấp cấu trúc cơ bản cho một màn hình ứng dụng, bao gồm AppBar, Body, Drawer, BottomNavigationBar,...



Hình 5.24: Minh họa về widget Scaffold

Ví dụ:

```
Scaffold(  
    appBar: AppBar(title: Text('Title')),  
    body: Center(child: Text('Main Content')),  
    floatingActionButton: FloatingActionButton(  
        // Floating Action Button code  
    ),  
)
```

```
        onPressed: () { },
        child: Icon(Icons.add),
      ),
)
```

Trong ví dụ trên, Scaffold gồm 3 tham số chính là appBar: thanh tiêu đề ở đầu màn hình, body: nội dung chính của màn hình, và nút nổi lên trên màn hình (floatingActionButton)

5.4.4 Cách tiếp cận khai báo trong Flutter

Lập trình mệnh lệnh (Imperative): Trong phong cách này, lập trình viên mô tả từng bước cụ thể để đạt được kết quả mong muốn. Ví dụ, để cập nhật giao diện, ta cần xác định và thao tác trực tiếp với các thành phần UI, như tìm kiếm widget bằng ID và thay đổi thuộc tính của nó.

Lập trình khai báo (Declarative): Ngược lại, lập trình khai báo tập trung vào việc mô tả *giao diện nên trông như thế nào* dựa trên trạng thái hiện tại. Khi trạng thái thay đổi, hệ thống tự động cập nhật giao diện để phản ánh sự thay đổi đó, mà không cần chỉ định các bước cụ thể để thực hiện.

Flutter áp dụng mô hình UI khai báo (Declarative UI), có nghĩa là:

"Đây là trạng thái hiện tại của ứng dụng, hãy trả về thứ gì đó trên màn hình cho phù hợp tương ứng trạng thái đó".

Giao diện người dùng (UI) trong Flutter được mô hình hóa như một hàm của trạng thái:



Hình 5.25: Mô hình hóa hàm trạng thái

Trong đó **UI** là giao diện người dùng được hiển thị trên màn hình, có thể quan

sát được, còn **state** là trạng thái hiện tại của ứng dụng, ví dụ như giá trị của một biến hoặc dữ liệu người dùng nhập.

Flutter áp dụng **mô hình khai báo**, có nghĩa giao diện được xây dựng dựa trên trạng thái ứng dụng. Điều này giúp đơn giản hóa việc quản lý giao diện và giảm thiểu lỗi, vì không cần xử lý các bước cập nhật UI một cách thủ công .

Cách tiếp cận khai báo giúp đơn giản hóa việc quản lý giao diện người dùng và giảm thiểu lỗi, vì Flutter tự động cập nhật UI khi trạng thái thay đổi. Thay vì điều khiển từng bước như trong cách tiếp cận mệnh lệnh, chỉ cần khai báo trạng thái mong muốn, và Flutter sẽ xử lý phần còn lại. Theo tài liệu chính thức của Flutter, cách tiếp cận này giúp tăng hiệu quả phát triển và đảm bảo tính nhất quán của giao diện. Bên cạnh đó, nó đảm bảo chỉ có một đường dẫn mã cho bất kỳ trạng thái nào của giao diện người dùng, điều này giúp dễ dàng kiểm soát và gỡ lỗi, dễ dàng bảo trì và mở rộng giao diện khi ứng dụng có trạng thái phức tạp hơn.

Flutter sử dụng phương thức `build()` của widget để dựng giao diện người dùng. Nói cách khác, giao diện được coi là một hàm của trạng thái hiện tại, tương ứng với công thức $UI = f(state)$ – tức là, mỗi khi trạng thái (state) thay đổi, Flutter tự động gọi lại `build()` để xây dựng lại cây widget phù hợp với trạng thái mới.

`build()` là phương thức trừu tượng trong lớp Widget (thường được override trong StatelessWidget hoặc trong lớp State của StatefulWidget), nhận một đối tượng BuildContext và trả về một Widget (hoặc cây widget con) mô tả cấu trúc giao diện. Flutter gọi `build()` trong các tình huống sau:

- Khi widget được chèn vào cây widget lần đầu tiên.
- Sau khi gọi `setState()`, báo hiệu rằng trạng thái đã thay đổi.
- Khi các widget kế thừa mà widget phụ thuộc vào thay đổi.
- Sau khi gọi `didUpdateWidget()` hoặc `deactivate()` và widget được chèn lại vào cây.

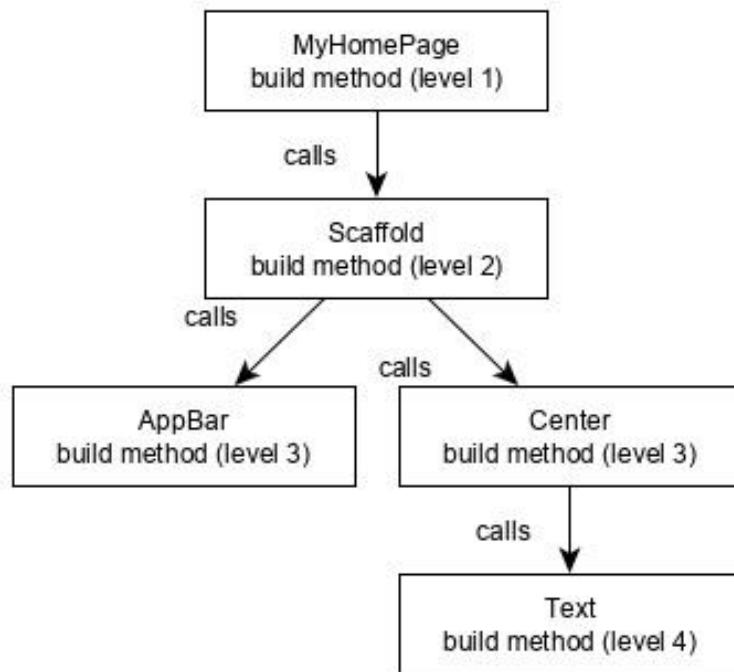
Nguyên tắc hoạt động: Mỗi lần `build()` được gọi, nó phải trả về một cấu trúc widget mới phản ánh trạng thái hiện tại. Flutter sau đó so sánh cây widget mới với cây cũ và cập nhật giao diện một cách hiệu quả

Ví dụ minh họa phương thức `build()` và cây widget:

Xét ví dụ dưới đây về một widget đơn giản:

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

Đây là một ví dụ điển hình thể hiện rõ cách Flutter sử dụng phương pháp xây dựng giao diện thông qua `build()` – định nghĩa giao diện dựa trên dữ liệu trạng thái (ở đây là biến `title`) và cấu trúc cây widget. Cây widget được xây dựng như sau:



Hình 5.26: Sơ đồ cây widget

Quá trình dựng UI thông qua `build()` trong Flutter có thể chia thành hai giai đoạn chính:

1. Giai đoạn khởi tạo (Initial Build):

- Khi một widget (chẳng hạn như `MyHomePage`) được chèn vào cây widget lần đầu tiên, Flutter sẽ gọi phương thức `build()` để xây dựng giao diện.
- Trong ví dụ trên, phương thức `build()` trả về một widget `Scaffold`, sau đó tiếp tục gọi đệ quy để tạo các widget con: `AppBar`, `Center`, và cuối cùng là hai `Text` widget.
- Quá trình này có tính chất đệ quy – mỗi widget cha tạo ra và xây dựng các widget con của nó trong cây.

2. Giai đoạn cập nhật (Rebuild):

- Với `StatelessWidget`, nếu các tham số truyền vào (như `title`) thay đổi, widget cũ sẽ bị loại bỏ và thay thế bằng một widget mới → khi đó `build()` được gọi lại để dựng lại cây giao diện tương ứng.

- Với StatefulWidget, khi gọi phương thức `setState()`, Flutter đánh dấu widget cần được cập nhật. Khi đến khung hình tiếp theo (frame), Flutter sẽ gọi lại `build()` cho widget đó để cập nhật giao diện phù hợp với trạng thái mới.
- Ngoài ra, nếu một InheritedWidget (là một widget chứa thông tin chia sẻ cho các widget con) mà widget đang phụ thuộc vào bị thay đổi, Flutter cũng sẽ tự động gọi lại `build()` của các widget con bị ảnh hưởng.

Tính hiệu quả và ưu điểm:

Việc sử dụng mô hình khai báo kết hợp với phương thức `build()` mang lại nhiều lợi ích:

- **Tăng khả năng đọc và bảo trì mã nguồn:** Cấu trúc widget dạng cây dễ hình dung và gắn liền trực tiếp với giao diện. Nhờ đó, lập trình viên có thể dễ dàng hình dung và điều chỉnh bộ cục ứng dụng chỉ bằng cách đọc mã.
- **Giảm thiểu lỗi cập nhật giao diện:** Thay vì phải tự mình cập nhật từng phần giao diện khi dữ liệu thay đổi (như trong lập trình mệnh lệnh), lập trình viên chỉ cần mô tả trạng thái mong muốn. Flutter sẽ tự động xử lý các thay đổi và dựng lại giao diện phù hợp, hạn chế tối đa lỗi logic hoặc thiếu sót khi cập nhật UI.
- **Hỗ trợ công cụ Hot Reload mạnh mẽ trong Flutter:** Vì toàn bộ giao diện được xác định bởi phương thức `build()`, Flutter có thể tái tạo lại nhanh chóng khi mã thay đổi. Điều này giúp lập trình viên thử nghiệm và kiểm tra giao diện gần như ngay lập tức mà không cần khởi động lại ứng dụng.

Hiệu suất và tối ưu hóa khi sử dụng `build()`:

Để đảm bảo hiệu năng cao khi ứng dụng phát triển phức tạp hơn, Flutter triển khai nhiều cơ chế tối ưu liên quan đến cách `build()` được gọi và xử lý:

- **Cơ chế so sánh (reconciliation):** Khi phương thức `build()` được gọi,

Flutter sẽ so sánh cây widget mới với cây cũ để phát hiện sự thay đổi. Chỉ những widget thực sự bị thay đổi mới được dựng lại hoặc cập nhật trên màn hình, nhờ đó tránh lãng phí tài nguyên không cần thiết.

- **Tránh đặt logic nặng trong `build()`:** Vì `build()` có thể được gọi lại thường xuyên, nên không nên thực hiện các tác vụ tốn tài nguyên như truy vấn cơ sở dữ liệu, xử lý phức tạp, hoặc gọi API trực tiếp trong hàm này. Những công việc như vậy nên được xử lý ở nơi khác (ví dụ như trong `initState()` hoặc khi người dùng tương tác), để tránh ảnh hưởng đến tốc độ phản hồi của giao diện.
- **Tách widget hợp lý để giới hạn phạm vi rebuild:** Một cách tối ưu phổ biến là chia nhỏ các phần giao diện thành các widget con riêng biệt, nhất là những phần thường xuyên thay đổi. Điều này giúp giới hạn khu vực cần tái dựng khi trạng thái thay đổi, đồng thời tăng khả năng tái sử dụng và kiểm soát logic tốt hơn.

5.5 Quản lý trạng thái trong Flutter

Quản lý trạng thái (state management) là một trong những yếu tố cốt lõi và không thể thiếu trong vòng đời của một ứng dụng Flutter. Trạng thái là bất kỳ dữ liệu nào có thể thay đổi trong quá trình chạy ứng dụng và ảnh hưởng đến cách giao diện người dùng được hiển thị. Do Flutter áp dụng mô hình UI khai báo, mỗi khi trạng thái thay đổi, UI sẽ cần được xây dựng lại để phản ánh sự thay đổi đó. Do đó, cách quản lý trạng thái hiệu quả sẽ quyết định đến hiệu suất, khả năng mở rộng và độ ổn định của ứng dụng.

Phân loại trạng thái:

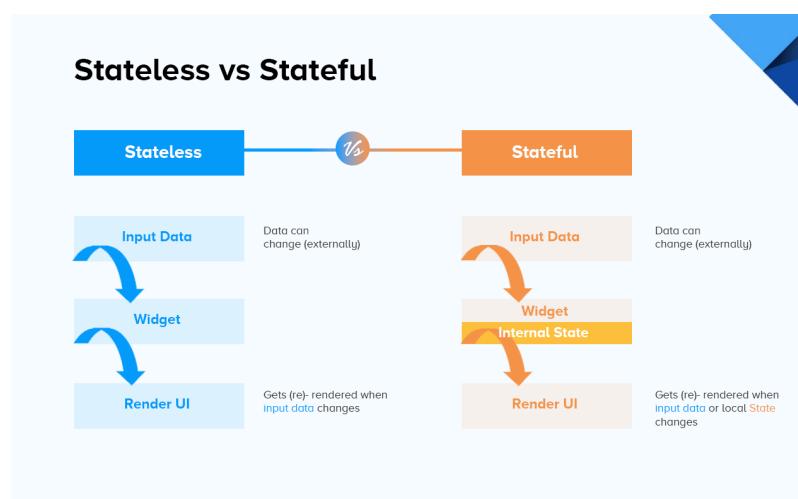
Trong Flutter, state có thể được chia thành hai nhóm chính:

- **Local state (trạng thái cục bộ):** Là trạng thái chỉ tồn tại trong phạm vi một widget duy nhất. Nó thường liên quan đến những thay đổi nhỏ trên UI như tab hiện tại đang chọn, checkbox được tích hay không, giá trị của một biến

boolean,... Trạng thái này thường được quản lý bằng StatefulWidget.

- **Application State (trạng thái toàn cục):** Là loại trạng thái không cục bộ, được chia sẻ giữa nhiều widget hoặc tồn tại xuyên suốt vòng đời của phiên làm việc của người dùng. Ví dụ như thông tin đăng nhập, danh sách sản phẩm trong giỏ hàng, tin nhắn trò chuyện,... Trạng thái này thường yêu cầu các giải pháp quản lý state phức tạp hơn như Provider, Riverpod, Bloc, Redux,...

Cách một widget xử lý và phản ứng với trạng thái được quyết định bởi việc nó là Stateless hay Stateful.



Hình 5.27: Trạng thái của Widget

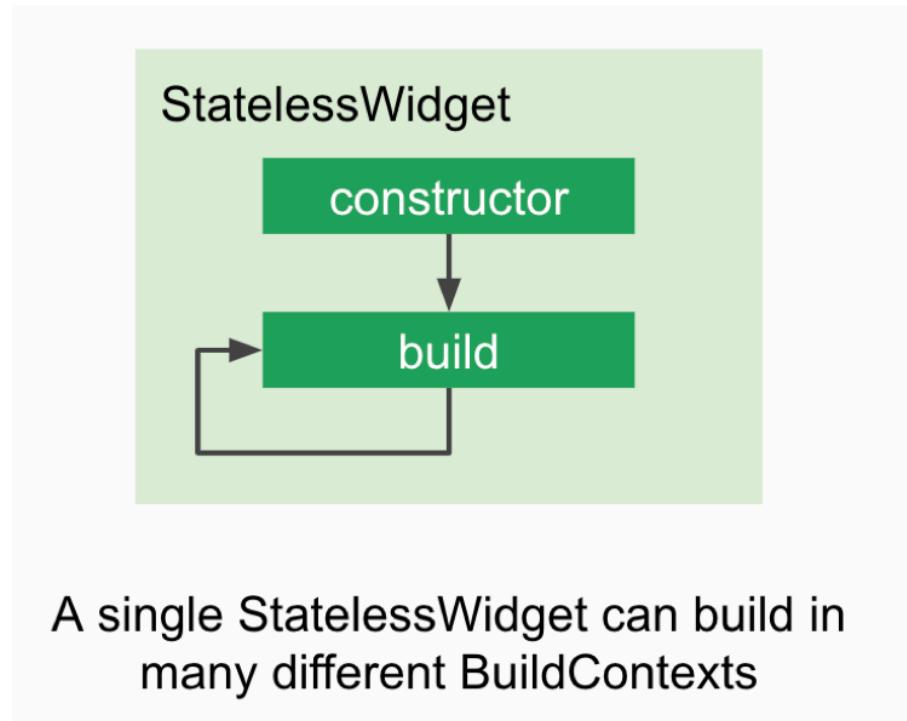
5.5.1 Stateless Widget

Stateless widget là một widget không quản lý trạng thái của chính nó. Khi được khởi tạo, nó nhận các tham số đầu vào thông qua Constructor và chỉ hiển thị UI dựa trên những giá trị đó. Nếu muốn thay đổi thì toàn bộ widget phải được tái tạo từ đầu với tham số mới. Bản thân **Stateless widget** cũng không có hàm createState mà thay vào đó là hàm build(BuildContext)

Vòng đời của Stateless Widget bao gồm hai giai đoạn cơ bản:

1. **Constructor được gọi:** Khi widget được thêm vào cây widget, Flutter khởi tạo một thể hiện mới bằng constructor, nhận các tham số từ widget cha.

2. **Phương thức `build()` được gọi:** Ngay sau khi khởi tạo, Flutter gọi phương thức `build(BuildContext context)` để dựng cây widget con, mô tả cách giao diện của widget này sẽ hiển thị.



Hình 5.28: Vòng đời của StatelessWidget

Như sơ đồ minh họa, StatelessWidget được khởi tạo thông qua constructor và chỉ thực hiện một hành vi duy nhất là gọi hàm `build()` để trả về cấu trúc giao diện. Sau đó, nó không tự thay đổi hoặc cập nhật bất kỳ phần tử nào.

Ví dụ:

```
Text (
    'Hello, Flutter!',
    style: TextStyle(
        fontSize: 20,
        color: Colors.blue
    )
)
```

Ở ví dụ trên, `Text` là một widget bất biến, nó chỉ hiển thị văn bản được truyền vào, nếu cần thay đổi nội dung thì cần phải tạo lại một widget `Text` mới với nội

dung khác.

5.5.2 Stateful widget

Trái ngược với Stateless Widget, một **Stateful Widget** là loại widget có khả năng quản lý trạng thái nội tại (internal state) và tự động cập nhật giao diện khi trạng thái thay đổi. Điều này khiến StatefulWidget rất hữu ích trong các tình huống tương tác động hoặc khi dữ liệu hiển thị cần thay đổi linh hoạt theo thời gian.

Stateful Widget không chỉ có phương thức `build()` mà còn có đối tượng `State` được liên kết xác định một số phương thức để hỗ trợ vòng đời của Widget. Chính đối tượng `State` này mới là nơi lưu trữ và quản lý trạng thái động của widget. Việc thay đổi trạng thái được thực hiện thông qua phương thức `setState()`, giúp Flutter biết rằng giao diện cần được cập nhật lại. StatefulWidget cho phép giao diện thay đổi dựa trên hành động của người dùng, dữ liệu động hoặc sự kiện bất đồng bộ.

Cấu trúc tổng quát của Stateful widget:

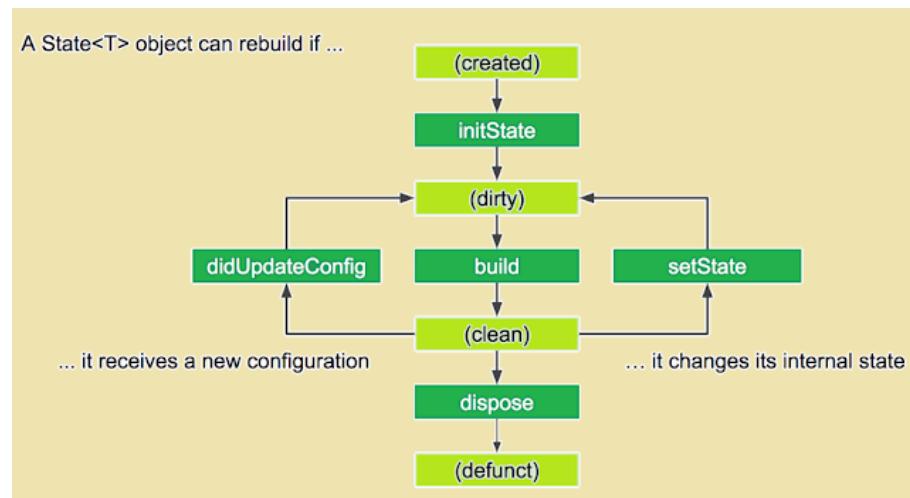
```
class MyWidget extends StatefulWidget {
    @override
    State<MyWidget> createState() => _MyWidgetState();
}

class _MyWidgetState extends State<MyWidget> {
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

Lớp **MyWidget** kế thừa từ `StatefulWidget` đại diện cho phần cấu hình không đổi của widget. Lớp **_MyWidgetState** kế thừa từ `State<MyWidget>` lưu trữ dữ liệu thay đổi và triển khai giao diện.

Vòng đời của Stateful widget:

Stateful Widget có một loạt các phương thức vòng đời trong lớp State, cho phép can thiệp vào từng giai đoạn hoạt động của widget.



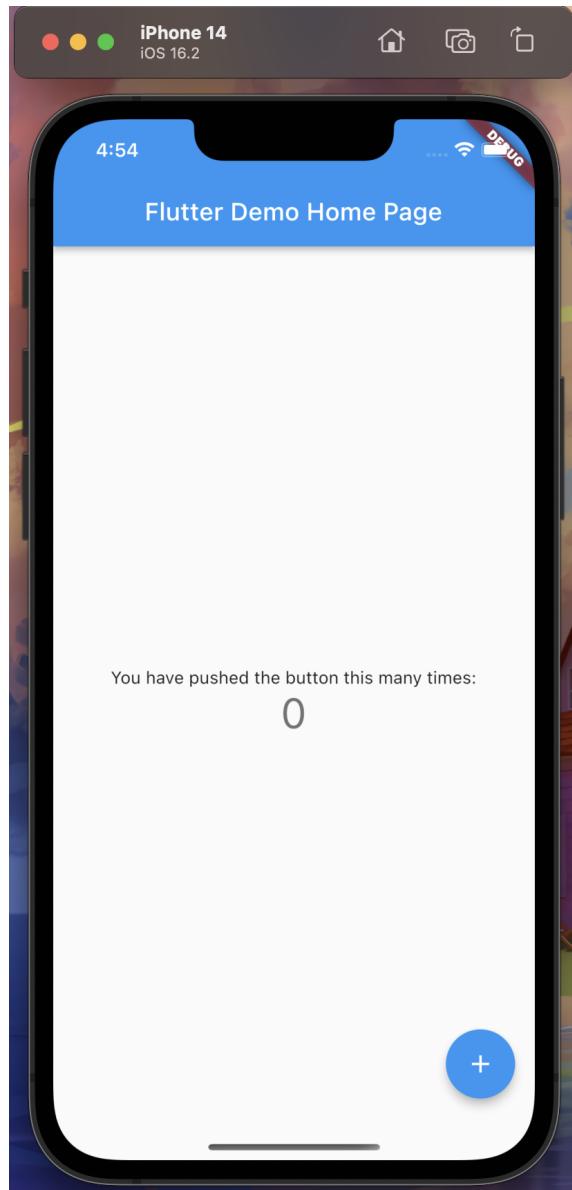
Hình 5.29: Vòng đời của Stateful Widget

- **createState()** : Khi một widget được thêm vào cây widget, Flutter sẽ gọi createState() để tạo một đối tượng trạng thái tương ứng. Đối tượng State này sau đó sẽ được giữ cố định trong suốt vòng đời của widget (trừ khi widget bị loại bỏ vĩnh viễn).
- **initState()** : Là phương thức đầu tiên được gọi trong lớp State, nó chỉ được gọi duy nhất một lần, ngay sau khi createState() hoàn tất và trước khi widget được dựng giao diện. Đây là nơi để khởi tạo các dữ liệu cần thiết, đăng ký lắng nghe sự kiện (listener), thiết lập controller, hoặc bất kỳ công việc nào cần thực hiện một lần duy nhất khi widget được đưa vào cây.
- **didChangeDependencies()** : Sau phương thức initState(), Flutter gọi didChangeDependencies() nếu widget phụ thuộc vào một hoặc nhiều đối tượng kế thừa (InheritedWidget). Phương thức này cũng có thể được gọi lại nếu một InheritedWidget mà widget đang phụ thuộc bị thay đổi. Đây là nơi phù hợp để truy xuất các giá trị phụ thuộc từ context.
- **build()** : Là phương thức cốt lõi của bất kỳ widget nào. Nó được gọi mỗi khi Flutter cần dựng lại giao diện dựa trên trạng thái hiện tại. Mỗi lần

`setState()` được gọi hoặc cấu hình widget thay đổi từ widget cha, Flutter sẽ tự động gọi lại `build()`. `BuildContext` được truyền vào để cho biết vị trí của widget trong cây widget, đồng thời cho phép truy xuất các widget tổ tiên như `Theme`, `MediaQuery`, hoặc `Provider`.

- **`setState()`**: Là cơ chế mà thông qua đó Flutter biết được rằng trạng thái nội tại của widget đã thay đổi. Khi gọi `setState(callback)`, Flutter sẽ đánh dấu widget là “dirty” và lập lịch gọi lại `build()` trong khung hình tiếp theo.
- **`didUpdateWidget()`**: Được gọi khi widget cha rebuild và truyền một phiên bản mới của widget con có cùng `runtimeType` nhưng khác cấu hình. Flutter sẽ tái sử dụng đối tượng `State` hiện tại thay vì tạo mới, và phương thức này sẽ cung cấp quyền truy cập vào phiên bản widget cũ để so sánh và xử lý sự thay đổi.
- **`deactivate()`**: Được gọi khi widget tạm thời bị loại khỏi cây widget, điều này có thể xảy ra khi điều hướng sang màn hình khác. Widget vẫn có thể được tái chèn lại nếu quay trở về màn hình trước trong cùng khung hình.
- **`dispose()`**: Là phương thức cuối cùng được gọi trong vòng đời. Flutter gọi `dispose()` khi widget bị loại bỏ vĩnh viễn khỏi cây widget. Đây là nơi cần giải phóng tài nguyên như controller, stream, animation, hoặc subscription để tránh rò rỉ bộ nhớ.

Xét ví dụ **CounterApp**: Đây là một ứng dụng đơn giản với giao diện gồm một nút bấm ở góc dưới bên phải màn hình. Mỗi lần người dùng nhấn nút “+”, số đếm hiển thị ở giữa màn hình sẽ tăng lên một đơn vị.



Hình 5.30: Counter App Demo

Mã nguồn được tổ chức thành hai lớp:

- **MyHomePage** kế thừa từ StatelessWidget, đóng vai trò là lớp cấu hình.
- **_MyHomePageState** kế thừa từ State<MyHomePage>, lưu trữ trạng thái động và định nghĩa giao diện.

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

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

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(widget.title)),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many
times:'),
            Text('$_counter',
              style: Theme.of(context).textTheme.
headlineMedium),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Phân tích hoạt động theo vòng đời:

- **Giai đoạn khởi tạo:** Khi ứng dụng chạy, Flutter gọi `createState()` từ lớp **MyHomePage**, trả về một thể hiện của **_MyHomePageState**. Sau đó Flutter gọi `initState()` trong `State` để khởi tạo trạng thái ban đầu của biến `_counter` là 0.
- **Giai đoạn dựng UI:** Ngay sau `initState()`, Flutter gọi `build()` để tạo giao diện lần đầu. Kết quả là một giao diện hiển thị số lần nhấn nút.
- **Tương tác người dùng:** Khi người dùng nhấn vào nút “+”, hàm `_incrementCounter()` được gọi, bên trong đó có gọi `setState()`. Lệnh `setState()` báo cho Flutter biết rằng trạng thái `_counter` đã thay đổi → giao diện cần được cập nhật. Flutter đánh dấu widget là “dirty” và trong khung hình tiếp theo, nó sẽ gọi lại phương thức `build()` để hiển thị giá trị mới.

Ta có thể thấy rõ sự liên kết giữa các giai đoạn trong vòng đời và hoạt động thực tế của widget: `initState()` và `build()` xuất hiện khi khởi tạo widget, `setState()` là cầu nối giữa thay đổi dữ liệu và cập nhật UI, `build()` được gọi lại mỗi khi trạng thái thay đổi nhưng không đồng nghĩa với việc toàn bộ giao diện bị dựng lại.

5.5.3 Các giải pháp quản lý trạng thái ứng dụng

Qua ví dụ CounterApp đã trình bày, có thể thấy rõ ràng phương thức `setState()` là một giải pháp đơn giản, hiệu quả cho việc quản lý trạng thái cục bộ (*Local state*) trong Flutter. Khi người dùng tương tác (ví dụ như nhấn nút “+”), hàm `_incrementCounter()` được gọi, và bên trong nó thực thi `setState()` để cập nhật biến `_counter`. Nhờ đó, Flutter đánh dấu lại widget tương ứng là “dirty” và tự động gọi lại phương thức `build()` để làm mới phần giao diện.

Tuy nhiên, khi phát triển các ứng dụng phức tạp hơn, việc quản lý trạng thái trở

nên thách thức. Cụ thể, nếu một trạng thái cần được chia sẻ giữa nhiều widget nằm rải rác trong cây giao diện, việc sử dụng thuận tay `setState()` không còn đủ hiệu quả. Trạng thái có thể bị truyền qua nhiều tầng widget không liên quan (hiện tượng gọi là *prop drilling*), khiến cho mã nguồn trở nên khó đọc, khó bảo trì và dễ phát sinh lỗi không mong muốn.

Một cách tiếp cận ban đầu là đặt trạng thái trong một widget cha đủ lớn, rồi truyền dữ liệu cho các widget con cần sử dụng thông qua constructor hoặc phương thức. Tuy nhiên, cách này nảy sinh một số vấn đề. Thứ nhất, các widget không sử dụng trạng thái cũng có thể bị `rebuild` do ảnh hưởng từ widget cha. Thứ hai, mã nguồn của widget cha trở nên cồng kềnh và khó tách biệt giữa giao diện và logic xử lý. Cuối cùng, khi ứng dụng có các logic nghiệp vụ cần kiểm tra, việc kiểm thử trở nên khó khăn nếu mọi thứ được ràng buộc trực tiếp vào widget UI.

Để giải quyết các bất cập trên, Flutter cung cấp nhiều cơ chế và thư viện hỗ trợ quản lý *Application state* – trạng thái dùng chung trên toàn bộ hoặc nhiều phần của giao diện. Đây là các giải pháp giúp tách biệt dữ liệu và UI, đồng thời tối ưu hiệu suất bằng cách giới hạn vùng ảnh hưởng khi trạng thái thay đổi.

Một trong những lớp nền tảng được sử dụng là **InheritedWidget**. Đây là một widget đặc biệt cho phép truyền dữ liệu từ trên xuống dưới qua cây widget, và các widget con có thể truy cập dữ liệu đó bằng phương thức `of(context)`. Tuy nhiên, việc sử dụng **InheritedWidget** trực tiếp thường không thân thiện với lập trình viên mới, do đó nhiều thư viện đã được xây dựng trên nền tảng này để đơn giản hóa việc quản lý trạng thái.

Các thư viện phổ biến hiện nay:

- **Provider**: Một thư viện được cộng đồng sử dụng rộng rãi, cung cấp một cách tiếp cận đơn giản nhưng mạnh mẽ để chia sẻ và giám sát trạng thái.
- **BLoC (Business Logic Component)**: Áp dụng mô hình lập trình phản ứng (reactive programming), giúp tách biệt hoàn toàn logic xử lý và giao diện bằng

cách sử dụng luồng dữ liệu (Streams).

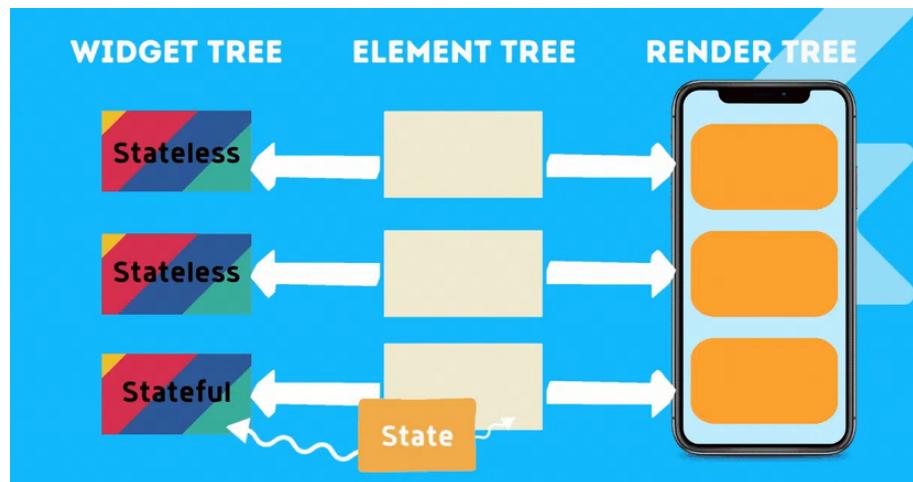
- **flutter_bloc**: Một gói mở rộng dựa trên BLoC, cung cấp các widget sẵn có giúp dễ dàng tích hợp và sử dụng.
- **Redux**: Mô hình quản lý trạng thái một chiều, lấy cảm hứng từ React Redux, phù hợp với các ứng dụng lớn và có luồng dữ liệu phức tạp.
- **MobX**: Một thư viện dựa trên nguyên lý lập trình phản ứng tự động (*observable*), giúp theo dõi và tự động cập nhật giao diện khi dữ liệu thay đổi.



Hình 5.31: Các giải pháp quản lý trạng thái ứng dụng

5.6 RenderObject và RenderTree

Trong Flutter, việc xây dựng giao diện người dùng (UI) không chỉ đơn thuần là tạo ra các widget. Thay vào đó, Flutter tổ chức hệ thống UI thành ba tầng logic: **Widget Tree**, **Element Tree** và **Render Tree**. Mỗi tầng đảm nhận một vai trò cụ thể, giúp tối ưu hóa hiệu suất, quản lý vòng đời giao diện và đảm bảo tính linh hoạt khi phát triển ứng dụng.



Hình 5.32: Các tầng Logic trong hệ thống UI của Flutter

5.6.1 Khái niệm

Widget Tree là tập hợp các widget, được xem như các khối xây dựng cơ bản của giao diện trong Flutter. Widgets là các đối tượng bất biến (immutable), đóng vai trò mô tả cấu hình của UI, chẳng hạn như bố cục, màu sắc hoặc nội dung. Tuy nhiên, widgets không trực tiếp thực hiện việc render mà chỉ cung cấp thông tin cần thiết cho các tầng khác. Khi các widget được lồng ghép, chúng tạo thành một cấu trúc cây gọi là Widget Tree.

Element Tree được tạo ra từ Widget Tree và đóng vai trò trung gian trong việc quản lý vòng đời của các widget. Mỗi element trong Element Tree tương ứng với một widget cụ thể và tham chiếu đến widget đó để truy cập cấu hình. Elements chịu trách nhiệm so sánh các widget mới với các widget hiện có, từ đó quyết định cách cập nhật giao diện mà không cần tạo lại toàn bộ các đối tượng render. Ngoài ra, Elements còn quản lý trạng thái (state) của các widget, đảm bảo rằng các thay đổi trạng thái được phản ánh chính xác trên UI.

Render Tree là tầng cuối cùng, nơi diễn ra quá trình render thực tế. Nó bao gồm các **RenderObject**, là các đối tượng chịu trách nhiệm thực hiện layout (bố trí), painting (vẽ) và xử lý tương tác người dùng trên màn hình. RenderObjects xác định các thuộc tính như kích thước, vị trí, hình học và nội dung của từng thành

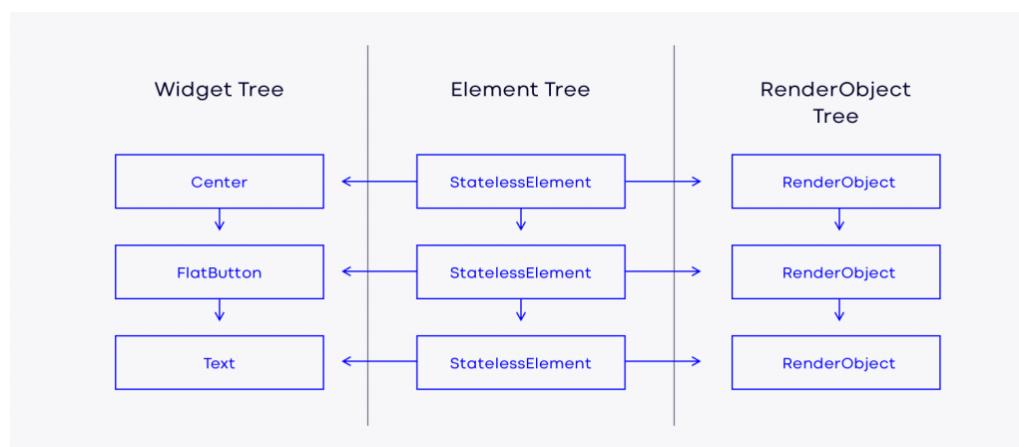
phần UI. Mỗi Element trong Element Tree trỏ đến một RenderObject trong Render Tree, đóng vai trò cầu nối giữa cấu hình và hiển thị thực tế. Chúng cũng hỗ trợ *hit testing*, giúp xác định các khu vực trên màn hình có thể phản hồi các cử chỉ như chạm, vuốt hoặc kéo.

RenderObject là thành phần cốt lõi của Render Tree. Mỗi RenderObject đại diện cho một phần của giao diện và biết cách tự bố trí (layout) bản thân cũng như các con của nó, đồng thời thực hiện việc vẽ (paint) lên màn hình. Ví dụ, một RenderObject như RenderOpacity có thể bọc một RenderObject khác và áp dụng hiệu ứng độ mờ trong quá trình vẽ.

Do chứa logic render phức tạp, RenderObjects là các đối tượng nặng (heavy), đòi hỏi nhiều tài nguyên tính toán. Vì vậy, Flutter tối ưu hóa bằng cách tái sử dụng RenderObjects hiện có thay vì tạo mới mỗi khi UI cần cập nhật. Các phương thức như `markNeedsPaint()` hoặc `markNeedsLayout()` được sử dụng để báo hiệu rằng một RenderObject cần được vẽ lại hoặc bố trí lại.

Render Tree hoạt động trong một hệ tọa độ độc lập với thiết bị (device-independent), sử dụng các pixel logic thay vì pixel vật lý. Điều này cho phép Flutter hiển thị giao diện nhất quán trên các thiết bị có độ phân giải và kích thước màn hình khác nhau.

5.6.2 Mối quan hệ giữa các Tree



Hình 5.33: Mối quan hệ giữa các cây

Ba cây phổi hợp chặt chẽ để xây dựng và cập nhật giao diện một cách hiệu quả:

- **Widget Tree:** Mô tả trạng thái mong muốn của UI thông qua các widget.
- **Element Tree:** Quản lý vòng đời và trạng thái, so sánh các widget mới với các widget hiện có để xác định các thay đổi cần thiết.
- **Render Tree:** Thực hiện render dựa trên cấu hình từ Element Tree, sử dụng các RenderObject để vẽ giao diện.

Sự phân tách này mang lại hiệu quả cao: khi một widget trong **Widget Tree** thay đổi, element tương ứng trong **Element Tree** sẽ kiểm tra xem widget mới có cùng loại (`runtimeType`) với widget hiện có hay không. Quá trình đối chiếu và cập nhật chọn lọc này được gọi là *reconciliation*; nó bảo đảm chỉ những phần thực sự cần thiết của giao diện được dựng lại, nhờ vậy giảm thiểu chi phí tính toán và cải thiện hiệu suất.

Nếu loại widget không đổi, Flutter chỉ cập nhật cấu hình của **RenderObject** hiện tại thay vì tạo mới. Ngược lại, khi loại widget thay đổi, element sẽ huỷ **RenderObject** cũ và tạo một **RenderObject** mới. Ví dụ, khi một widget `Text` đổi nội dung từ “Hello” sang “World” nhưng vẫn giữ nguyên loại, element chỉ cập nhật thuộc tính văn bản của `RenderObject` thay vì dựng lại toàn bộ.

Khi người dùng tương tác với ứng dụng—chẳng hạn nhấn một nút—hành động được các *gesture recognizer* trong `RenderObject` xử lý. Những recognizer này gọi các callback do widget cung cấp, có thể dẫn tới việc thay đổi trạng thái của một *stateful widget*. Khi trạng thái thay đổi, **Widget Tree** được tái dựng, **Element Tree** so sánh phiên bản mới với hiện tại để xác định phần cần cập nhật, rồi **Render Tree** được điều chỉnh tương ứng để phản ánh thay đổi trên màn hình.

Vì sao cần đến ba cây?

Flutter áp dụng mô hình ba cây vì các lý do sau:

- **Hiệu suất:** Widgets nhẹ và có thể được tạo lại thường xuyên mà không tốn

kém, trong khi RenderObjects nặng hơn và được tái sử dụng để tránh tạo lại không cần thiết.

- **Rõ ràng:** Mỗi cây có trách nhiệm riêng biệt, giúp mã nguồn dễ hiểu và dễ bảo trì.
- **An toàn kiểu dữ liệu:** Sự phân tách ngăn chặn việc trộn lẫn logic giữa cấu hình và render, đảm bảo rằng các loại dữ liệu đúng được sử dụng trong từng ngữ cảnh.

Ví dụ minh họa về sử dụng ba cây trong Flutter:

Xét một ứng dụng Flutter đơn giản hiển thị văn bản “Hello” và một nút “Change Content”. Khi người dùng nhấn nút, văn bản chuyển thành “Welcome”.

- **Widget Tree:** Bao gồm các widget như Scaffold, Center, Column với Text (văn bản) và ElevatedButton (nút “Change Content”).
- **Element Tree:** Mỗi widget có element tương ứng. Widget chính là StatefulWidget quản lý nội dung văn bản.
- **Render Tree:** RenderParagraph cho Text và RenderDecoratedBox cho nút.

Khi người dùng nhấn nút “Change Content”, trạng thái văn bản được cập nhật từ “Hello” sang “Welcome”. Widget Tree được xây dựng lại với Text có nội dung mới. Element Tree so sánh và cập nhật element cho Text. Render Tree cập nhật RenderParagraph để vẽ văn bản mới. Chỉ phần Text được vẽ lại, các phần khác không bị ảnh hưởng. Quy trình này đảm bảo rằng chỉ các phần cần thiết được cập nhật, tối ưu hóa hiệu suất.

Kiến trúc *ba cây* (Widget – Element – Render) cùng cơ chế *reconciliation* (là quá trình xác định phần nào của cây widget sẽ được rebuilt) cho phép Flutter vừa bảo tồn trạng thái, vừa giới hạn phạm vi vẽ lại, nhờ đó duy trì hiệu năng cao ngay cả khi UI thay đổi liên tục. Việc tách riêng cấu hình, vòng đời và kết xuất không

PHẦN 5. GIỚI THIỆU FRAMEWORK FLUTTER

chỉ giúp mã nguồn mạch lạc, dễ bảo trì, mà còn tạo tiền đề để các tối ưu nâng cao, tái sử dụng Element, tận dụng bộ nhớ đệm, đem lại trải nghiệm mượt mà cho người dùng đa nền tảng.

TÀI LIỆU THAM KHẢO

- [1] blup.in (2023). A Brief History of Flutter. <https://www.blup.in/blog/a-brief-history-of-flutter>
- [2] A. W. Junaid, "History and Evolution of Dart Programming Language," *awjunaid.com*, Sep. 13, 2023. [Online]. <https://awjunaid.com/dart/history-and-evolution-of-dart-programming-language/>.
- [3] GeeksforGeeks (2025). Flutter Tutorial. <https://www.geeksforgeeks.org/flutter-tutorial/>
- [4] Dart Tutorial - Learn Dart Programming. OOP in Dart. <https://dart-tutorial.com/object-oriented-programming/oop-in-dart/>
- [5] viblo.asia. Những Widget cơ bản trong Flutter. <https://viblo.asia/p/nhung-widget-co-ban-trong-flutter-1Je5Eygy5nL>
- [6] GeeksforGeeks (2023). Flutter – Stateful Widget. <https://www.geeksforgeeks.org/flutter-stateful-widget/>
- [7] Flutter Docs (2025). Flutter architectural overview. <https://docs.flutter.dev/resources/architectural-overview>
- [8] Bailey, T., & Biessek, A. (2021). *Flutter for Beginners – Second Edition: An introductory guide to building cross-platform mobile applications with Flutter 2.5 and Dart.*
- [9] Tashildar, A., Shah, N., Gala, R., Giri, T., & Chavhan, P. (2020). *Application Development Using Flutter*. International Research Journal of Modernization in Engineering Technology and Science, 2(8).
- [10] Đại học Bách Khoa Hà Nội (2020). *Slide bài giảng học phần IT4788 – Phát*

triển ứng dụng đa nền tảng. <https://soict.daotao.ai/courses/course-v1:SoICT+IT4788+2020-2/about>

- [11] Rahul, I. *All In One Flutter Book.* GitHub repository. https://github.com/irahulcse/All_In_One_Flutter_Book