

앱 개발 입문

성균관대학교 컬처애편테크놀로지융합전공 하계 부트캠프

2일차 - 2023. 06. 23 (금)

강의 슬라이드 링크

https://github.com/kunny/skku-bootcamp-2023-summer/blob/main/_slides/day2.pdf

오늘 강의에서 다룰 내용

- 플러터 앱 제작에 사용하는 다트(Dart) 프로그래밍 언어의 기본 문법을 살펴봅니다.
- 1일차 강의에서 생성한 플러터 프로젝트의 동작을 바꿔봅니다.

Dart 언어 소개

Dart란?

Dart is a **client-optimized language** developed by Google for developing fast apps on any platform. Its goal is to **offer the most productive programming language for multi-platform** development, paired with a **flexible execution runtime platform** for app frameworks.



Dart

주요 특징

- Static type checking
- Sound null safety
- 풍부한 라이브러리 지원
- 다양한 타겟 플랫폼 지원
- 웹 기반 코드 작성 및 실행 도구 지원 (DartPad)

주요 특징 - Static type checking

- **컴파일 시점**에 변수의 타입을 확인하므로, 실행 시점에 올바르지 않은 타입의 값이 대입될 확률을 줄여줍니다.
- **타입 추론**을 지원하므로, 변수에 대입된 값을 통해 타입을 유추할 수 있다면 타입을 생략하고 변수를 선언할 수 있습니다.

```
// int 타입 변수 선언 후 초기값 0 대입
```

```
int foo = 0;
```

```
// 변수 타입을 선언하지 않았지만, 초기값의 타입을 통해 String으로 유추합니다.
```

```
var bar = 'foo'
```

```
// 필요한 경우 동적 타입을 사용할 수도 있습니다.
```

```
dynamic baz = 0;
```

```
baz = 'random string';
```

```
baz = 0.1234;
```

주요 특징 - Sound null safety

Non-zero value



null



0



undefined



주요 특징 - Sound null safety

- 변수의 null값 허용 여부를 **컴파일 시점**에 판단합니다.
- null값을 체크하지 않아 발생할 수 있는 런타임 오류를 미연에 방지합니다.

```
// null 값을 허용하지 않는 Tissue 변수
```

```
Tissue nonNullTissue = new Tissue();
```

```
// 성공: null이 아니므로 항상 호출 가능합니다.
```

```
nonNullTissue.pop();
```

```
// null 값을 허용하는 Tissue 변수
```

```
Tissue? nullableTissue;
```

```
// 오류: nullableTissue는 null일 수 있기에 pop() 함수를 호출하기 전에 null 여부를 확인해야 합니다.
```

```
nullableTissue.pop();
```

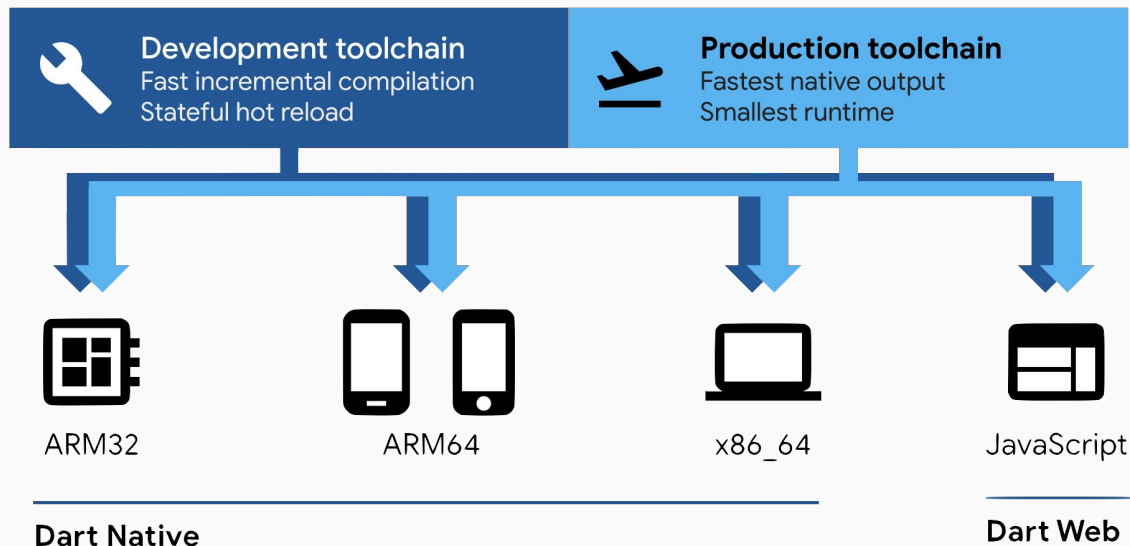
주요 특징 - 풍부한 라이브러리 지원

- 멀티플랫폼을 지원하는 [core 라이브러리](#)와 더불어 네이티브 및 웹 플랫폼에 특화된 라이브러리도 제공합니다.
- [pub.dev](#)를 통해 다양한 서드파티 라이브러리를 받을 수 있습니다.

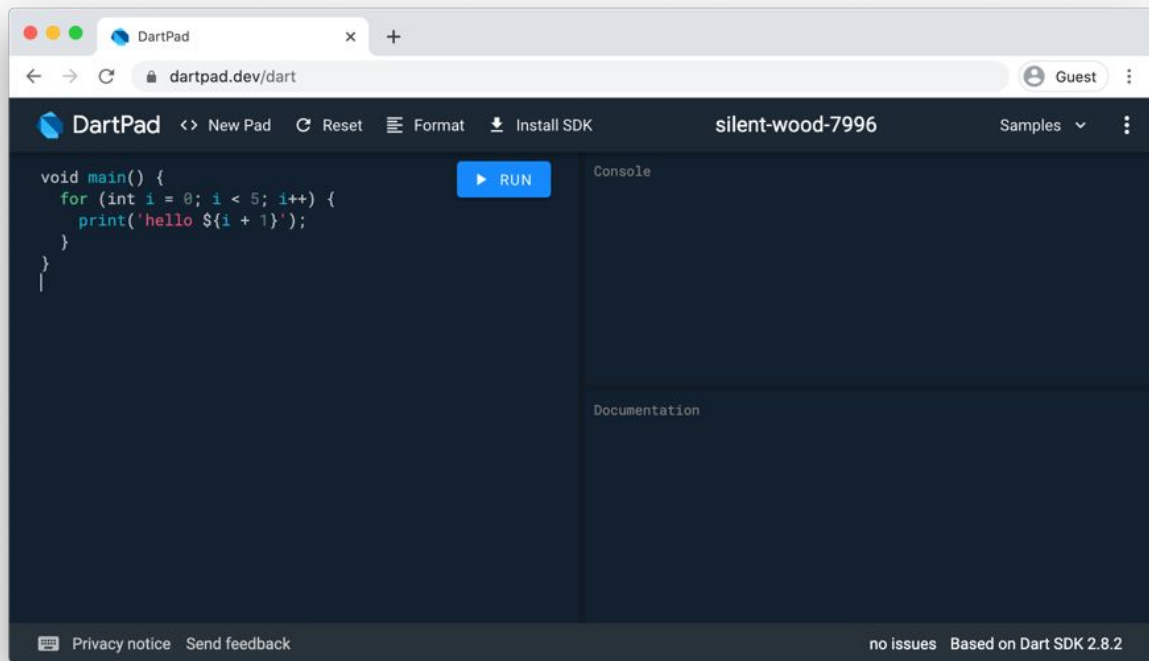


주요 특징 - 다양한 타겟 플랫폼 지원

- **Native:** (모바일 및 데스크탑) 닥트 코드를 타겟 기기에 맞는 기계어로 변환하여 실행합니다.
- **Web:** 웹 컴파일러가 닥트 코드를 자바스크립트로 변환하여 실행합니다.



주요 특징 - 웹 기반 코드 작성 및 실행 도구 지원 (DartPad)



<https://dartpad.dev>

Dart 언어 주요 문법

기본 구문

변수 (Variables) - 선언 방법

```
// String 타입 값 'Bob'을 갖는 변수 name을 선언합니다.  
// 타입을 선언하지 않아도 자동으로 String 타입으로 추론됩니다.  
var name = 'Bob';  
  
// 변수에 다양한 타입의 값을 저장하려면 Object 혹은 dynamic 타입을 사용합니다.  
Object name = 'Bob';  
// ... or ...  
dynamic name = 'Bob';  
  
// 타입을 명시적으로 선언할 수도 있습니다.  
String name = 'Bob';
```

변수 (Variables) - null 안전성

// null값을 허용하는 변수. 기본값으로 null이 할당됩니다.

```
String? name;
```

// null을 허용하지 않는 변수. 일반적으로 변수 선언 시점에 값을 할당해줍니다.

```
String name = 'foo';
```

```
void foo() {
```

```
    String a = 'foo';
```

```
    int b;
```

// null을 허용하지 않는 지역 변수는 변수를 사용하기 전에만 값을 할당하면 됩니다.

```
    if (countStringLength) {
```

```
        b = a.length;
```

```
    } else {
```

```
        b = 0;
```

```
    }
```

```
    print(b);
```

```
}
```

변수 (Variables) - late 변수

```
// 코드 구조상 변수 선언 시점에 값을 할당할 수 없지만,  
// null 값을 허용하지 않는 변수를 선언하려면 late 키워드를 추가해줍니다.  
late String description;  
  
void main() {  
    // 필요한 시점에 변수에 값을 할당합니다. 만약 변수에 값을 할당하지 않고 사용하게 되면  
    // 변수 참조 시점에 런타임 오류가 발생할 수 있습니다.  
    description = 'foo';  
    print(description);  
}
```


변수 (Variables) - final, const

// 런타임 시점에 변수에 할당된 값을 바꿀 수 없는 **final** 변수를 선언합니다.

// **final** 변수 선언시 **var** 혹은 타입 키워드를 생략할 수 있습니다.

```
final name = 'Bob';
```

// 타입을 함께 사용할 수도 있습니다.

```
final String nickname = 'Bobby';
```

// 오류: **name** 변수에 이미 값이 할당되었기에 다른 값을 할당할 수 없습니다.

```
name = 'Alice';
```

// 컴파일 시점에 값이 할당되고, 이후 값을 바꿀 수 없는 **const** 변수를 선언합니다.

// 고정된 값을 여러 번 참조해야 할 때 유용합니다.

```
const bar = 1000000;
```

```
const double atm = 0.01325 * bar;
```

산술 연산자 (Arithmetic operators)

연산자/	의미
+	덧셈
-	뺄셈
-expr	피연산자의 부호를 바꿉니다.
*	곱셈
/	나눗셈
~/	나눗셈 (소숫점 아래 결과를 버린 정수 반환)
%	나머지
++var	var = var + 1 (표현식은 var + 1을 반환)
var++	var = var + 1 (표현식은 var를 반환)
--var	var = var - 1 (표현식은 var - 1을 반환)
var--	var = var - 1 (표현식은 var를 반환)

산술 연산자 (Arithmetic operators)

```
var a = 2;  
var b = 3;  
var c = a + b;  
print(c); // 5  
  
var d = -c;  
print(d); // -5  
  
var e = a * b;  
var f = c / a;  
var g = c ~/ a;  
print(e); // 6  
print(f); // 2.5  
print(g); // 2
```

```
var h = c % a;  
print(h); // 1  
  
print(a++); // 2  
print(a); // 3  
print(++a); // 4  
print(a); // 4  
  
print(b++); // 3  
print(b); // 4  
print(++b); // 5  
print(b); // 5
```

항등 및 비교 연산자 (Equality and relational operators)

연산자	의미
<code>==</code>	같음
<code>!=</code>	같지 않음
<code>></code>	큼
<code><</code>	작음
<code>>=</code>	크거나 같음
<code><=</code>	크거나 작음

항등 및 비교 연산자 (Equality and relational operators)

```
print(2 == 2); // true
```

```
print(2 < 4); // true
```

```
print(2 > 4); // false
```

```
print(2 != 3); // true
```

```
print(2 <= 4); // true
```

```
print(2 >= 4); // false
```

타입 테스트 연산자 (Type test operators)

연산자	의미
as	타입 변환
is	객체가 특정 타입인 경우 true
is!	객체가 특정 타입을 포함하지 않을 경우 true

타입 테스트 연산자 (Type test operators)

```
var theAnswerToLifeTheUniverseAndEverything = 42;  
print(theAnswerToLifeTheUniverseAndEverything is int); // true  
print(theAnswerToLifeTheUniverseAndEverything is! num); // false
```

```
dynamic foo = 'foo';  
String asString = foo as String;  
print(foo is String); // true  
print(asString is String); // true
```

대입 연산자 (Assignment operators)

연산자	의미
a = b	b를 a에 대입
a ??= b	a가 null이면 b를 대입하고, 그렇지 않으면 a의 값 유지
a op= b	a = a op b

대입 연산자 (Assignment operators)

```
String? name;
```

```
name ??= 'foo';
```

```
print(name); // name이 null이었으므로 'foo' 값이 들어갑니다.
```

```
name ??= 'bar';
```

```
print(name); // name이 null이 아니므로 'bar' 값이 들어가지 않고 기존 값인 'foo'를  
유지합니다.
```

```
int a = 10;
```

```
a *= 10;
```

```
print(a); // 10에 10을 곱한 값인 100이 대입됩니다.
```

논리 연산자 (Logical operators)

연산자	의미
!	논리 부정
	논리 OR
&&	논리 AND

논리 연산자 (Logical operators)

```
print(!true); // false
```

```
print(true == true); // true
```

```
print(true || false); // true
```

```
print(false || false); // false
```

```
print(true && false); // false
```

```
print(true && true); // true
```

조건 연산자 (Conditional operators)

연산자	의미
<code>condition ? expr1 : expr2</code>	condition이 참이라면 expr1의 값을, 그렇지 않다면 expr2의 값 반환
<code>expr1 ?? expr2</code>	expr1이 null이 아니면 expr1의 값을, 그렇지 않다면 expr2의 값 반환

조건 연산자 (Conditional operators)

```
var a = 5;  
var b = 3;  
var result = a > b ? 'a is bigger than b' : 'a is same or smaller than b';  
print(result); // 'a is bigger than b'
```

```
String? selection = null;  
var result = selection ?? 'default';  
  
print(result); // 'foo'
```

기타 연산자

연산자	의미
()	함수 호출
[]	요소 접근
?[]	null을 허용하는 변수의 요소 접근
.	멤버 접근
?.	null을 허용하는 변수의 멤버 접근
!	null을 허용하지 않는 타입으로 변환

기타 연산자

```
void foo() { }
```



```
void test() {  
    List<int>? arr;  
    var bar = 'bar';  
  
    foo(); // foo 함수 호출  
    print(arr?[0]); // 아직 arr에 값이 할당되지 않았으므로 null 출력  
    print(bar.length); // String클래스의 length 프로퍼티 호출  
  
    arr = [1, 2, 3];  
    print(arr?[0]); // arr이 null이 아니므로 첫번째 요소의 값인 1 출력  
    print(arr![0]); // arr이 null이 아니므로 null을 허용하지 않는 타입으로 변환 후 요소 접근 가능  
}
```

주석 (Comments)

- 코드 내에 부가 설명을 넣을때 사용합니다.
- 단일 행(`//`) 및 여러 행(`/* */`)에 걸쳐 주석을 작성할 수 있습니다.

```
void main() {  
    // TODO: refactor into an AbstractLlamaGreetingFactory?  
    print('Welcome to my Llama farm!');  
  
    /* This is a lot of work. Consider raising chickens.  
    Llama larry = Llama();  
    larry.feed();  
    larry.exercise();  
    larry.clean();  
    */  
}
```


라이브러리 및 import 구문

다른 개발자가 작성한 코드를 내가 작성한 코드로 불러올 때 사용합니다.

```
// 머티리얼 디자인을 따르는 위젯이 정의된 라이브러리를 불러옵니다.  
import 'package:flutter/material.dart';  
  
// 머티리얼 디자인을 따르는 StatelessWidget 클래스를 사용합니다.  
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  ...  
}
```

키워드 (keywords)

Dart언어에서 특수한 목적으로 사용하는 단어로, 다른 용도로 사용할 수 없습니다.

`abstract`

`else`

`import`

`show`

`as`

`enum`

`in`

`static`

`assert`

`export`

`interface`

`super`

`async`

`extends`

`is`

`switch`

(이후 생략)

<https://dart.dev/language/keywords>

Dart 언어 주요 문법

자료형

주로 사용하는 내장 타입 (Built-in types)

- 숫자 (int, double)
- 문자열 (String)
- 논리 (bool)
- 리스트/배열 (List)
- 집합 (Set)
- 맵 (Map)
- null

내장 타입 - int, double, String, bool

// 정수를 저장할 수 있는 변수를 선언합니다.

```
int a = 5;
```

// 부동 소수점(floating-point) 숫자를 저장할 수 있는 변수를 선언합니다.

```
double b = 1.3;
```

// 문자열을 저장할 수 있는 변수를 선언합니다.

```
String c = 'foo';
```

// 부울 값을 저장할 수 있는 변수를 선언합니다.

```
bool d = false;
```

내장 타입 - List

// 문자열을 저장할 수 있는 리스트를 선언합니다.

```
var a = ['foo', 'bar', 'baz'];
```

// 리스트의 두번째 요소에 접근합니다.

```
print(a[1]); // 'bar'
```

// 리스트의 길이를 반환합니다.

```
print(a.length);
```

// 스프레드 연산자(...)를 사용하여 a 리스트에 있는 항목을 b 리스트에 복사합니다.

```
var b = ['a', 'b', ...a];
```

```
print(b.length); // 5
```

내장 타입 - Set

// 문자열을 저장할 수 있는 집합을 선언합니다.

```
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
```

// 문자열을 저장할 수 있는 빈 집합을 선언합니다.

```
var names = <String>{}; // or Set<String> names = {};
```

// 집합에 새 항목을 추가합니다.

```
names.add('foo');
```

내장 타입 - Map

```
// Key-value 값을 저장할 수 있는 Map 변수를 선언합니다.  
// Key로 int 타입을, Value로 String 타입을 사용합니다.  
var nobleGases = {  
    2: 'helium',  
    10: 'neon',  
    18: 'argon',  
};  
  
// Key가 2인 값을 반환합니다.  
print(nobleGases[2]); // 'helium'
```


제너릭 (Generics)

- 타입만 다를 뿐 동일한 작업을 구현하는 경우, 타입을 인자로 받는 하나의 구현체만 작성할 수 있습니다.
- 받을 수 있는 타입을 명시하므로, 올바르지 않은 타입이 들어와 발생할 수 있는 오류를 방지할 수 있습니다.
- 사용할 타입은 `<Type>`을 통해 전달합니다.

```
var names = <String>[];  
names.addAll(['Seth', 'Kathy', 'Lars']);  
names.add(42); // 오류: 문자열 타입을 받는 리스트에 숫자를 넣을 수 없습니다.
```

```
var data = <String, String> {};  
data['name'] = 'foo'; // 성공  
data['phone'] = 12345; // 오류: 숫자 타입 값을 넣을 수 없습니다.  
data[123] = 'bar'; // 오류: 키 값으로 문자열 타입을 사용해야 합니다.
```

Dart 언어 주요 문법

함수

함수 개요

- 반복되는 작업, 혹은 항상 연이어 실행되는 일련의 명령을 묶어줍니다.
- 함수 실행시 사용할 인자를 받을 수 있고, 실행 후 값을 반환할 수 있습니다.

```
// int 타입 인자 2개를 받아 int타입 값을 반환하는 함수입니다.
```

```
int plus(int a, int b) {  
    return a + b;  
}
```

```
// 아무 인자도 받지 않고, 값도 반환하지 않는 함수입니다.
```

```
void doSomething() {  
    print('doing something');  
}
```

명명된 인수 (Named Parameters)

- 함수 호출시 인수 이름을 함께 사용하여 값을 전달할 수 있습니다.
- 명명된 인수는 중괄호 ({ }) 내부에 선언합니다.
- 필수로 받아야 하는 인수와 그렇지 않은 인수를 함께 선언할 때 유용합니다.

```
// 명명된 인수를 사용하는 함수를 선언합니다.
```

```
void doSomething({String? a, int? b}) { ... }
```

```
// 명명된 인수에 값을 전달하려면 인수 이름을 명시해야 합니다.
```

```
doSomething(a: 'foo', b: 10);
```

```
// 인수 순서를 바꿀 수도 있습니다.
```

```
doSomething(b: 10, a: 'foo');
```

명명된 인수 (Named Parameters)

// required 키워드를 붙여 항상 값을 전달하게끔 만들어 줍니다.

```
void doSomething({required String a, int? b}) {  
    print('a: $a b: $b');  
}
```

// 명명된 인수의 기본값을 할당할 수 있습니다.

```
void doSomethingElse({required String a, int b = 10}) {  
    print('a: $a b: $b');  
}
```

// b에 값을 전달하지 않으면 null이 할당됩니다.

```
doSomething(a: 'foo'); // a: foo b: null
```

// b에 값을 전달하지 않으면 기본값이 할당됩니다.

```
doSomethingElse(a: 'bar'); // a: bar b: 10
```

익명 함수 (Anonymous function)

- 이름 없이 인자 및 본체로만 이루어진 함수이며, 함수 타입 인자에 값을 대입할 때 주로 사용합니다.
- 하나의 표현식으로만 구성된 함수는 람다식 형태로 간결하게 표현할 수 있습니다.

```
const list = ['apples', 'bananas', 'oranges'];
list.map((item) {
  return item.toUpperCase();
}).forEach((item) {
  print('$item: ${item.length}');
});
```

```
const list = ['apples', 'bananas', 'oranges'];
list.map((item) => item.toUpperCase())
  .forEach((item) {
    print('$item: ${item.length}');
  });
```

함수 타입

- 함수를 타입처럼 사용할 수 있습니다. 함수 타입을 인자로 받아 비동기 작업의 완료 여부를 확인할 때 사용하는 리스너를 구현할 수 있습니다.

```
void main() {  
    doSomething((bool result) {  
        print('Result: $result');  
    });  
}  
  
void doSomething(Function(bool result) onComplete) {  
    // 오랜 시간이 걸리는 작업 실행  
    ...  
  
    // onComplete 함수를 호출하면서 true를 인자로 전달  
    onComplete(true);  
}
```

Dart 언어 주요 문법

흐름 제어

반복문 (Loops) - for

시작, 증감 및 종료 조건과 매 반복마다 실행할 내용을 작성할 수 있습니다.

```
var message = StringBuffer('Dart is fun');

// i에 초기값 0을 대입하고, 매 반복마다 1씩 증가시킨 후의 값이 5보다 작다면 계속
반복합니다.
for (var i = 0; i < 5; i++) {
    message.write('!');
}

print(message); // Dart is fun!!!!
```

반복문 (Loops) - for-in

Iterable 타입은 for-in 연산자를 사용하여 각 요소를 순회할 수 있습니다.

```
const words = ['Foo', 'Bar', 'Baz'];
```

```
for (var word in words) {  
    print(word);  
}
```

```
// 출력:
```

```
// Foo
```

```
// Bar
```

```
// Baz
```

반복문 (Loops) - while, do-while

반복 종료 조건만 정의하는 반복문입니다. 반복 횟수와 상관없이 특정 작업을 계속해서 반복해야 할 때 사용합니다.

```
// isDone()이 false를 반환하기 전까지 계속 반복합니다.
```

```
while (!isDone()) {  
    doSomething();  
}
```

```
// 반복 종료 조건과 상관없이 아래 블록은 최소 한 번 실행됩니다.
```

```
do {  
    printLine();  
} while (!atEndOfPage()); // atEndOfPage()가 true가 아닐 때까지 반복합니다.
```

반복문 (Loops) - break, continue

- **break**: 반복을 종료합니다.
- **continue**: 현재 반복문을 종료한 후 다음 반복문을 이어서 실행합니다.

```
while (true) {  
    // shutdownRequested()가 true를 반환하면 반복을 종료합니다.  
    if (shutdownRequested()) break;  
    processIncomingRequests();  
}  
  
for (int i = 0; i < candidates.length; i++) {  
    var candidate = candidates[i];  
    // candidate.yearsExperience가 5보다 작으면 다음 후보자로 넘어갑니다.  
    if (candidate.yearsExperience < 5) {  
        continue;  
    }  
    candidate.interview();  
}
```

분기문 (Branch) - if, else

특정 조건을 만족하면 해당 블록을 실행합니다.

```
if (isRaining()) {  
    you.bringRainCoat();  
} else if (isSnowing()) {  
    you.wearJacket();  
} else {  
    car.putTopDown();  
}
```

분기문 (Branch) - switch

분기해야 할 조건이 많을 때 유용하게 사용할 수 있습니다. **default**를 사용하여 나열된 조건이 아닐 때 실행할 작업을 구성할 수 있습니다.

```
var command = 'OPEN';

switch (command) {
  case 'CLOSE':
    executeClose();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
```

분기문 (Branch) - switch

break를 사용하지 않으면 다음 **case**문을 이어서 실행합니다. 다수의 조건에 대해 동일한 명령을 실행해야 할 때 유용하게 사용할 수 있습니다.

```
var command = 'OPEN';

switch (command) {
  case 'APPROVE':
    executeApprove();
    break;
  case 'OPEN':
  case 'CLOSE':
    executeOpenOrClose();
    break;
  default:
    executeUnknown();
}
```

Dart 언어 주요 문법

클래스

클래스 소개

- Dart에서 모든 객체는 클래스 형태로 표현됩니다.
- 생성자, 인스턴스 변수 및 함수, 정적 변수 및 함수를 포함할 수 있습니다.

```
class Person {  
    // 정적 인스턴스 변수  
    static const tag = 'Person';  
  
    // 정적 변수  
    final String id;  
  
    final String name;  
  
    // 생성자  
    Person(this.id, this.name);  
  
    // 정적 함수  
    static void foo() { ... }  
  
    // 인스턴스 함수  
    void bar() { ... }  
}
```

클래스 - 생성자

- 새 클래스를 만들 때 사용하며, 생성자가 호출될 때 인스턴스 변수가 모두 초기화 되어야 합니다.

```
class Person {  
    final String id;  
    final String name;  
  
    Person(this.id, this.name);  
}
```

```
// Person 생성자를 호출하여 새로운 Person객체를 만듭니다.  
var foo = Person('foo', 'Foo');
```

클래스 - 상속

다른 클래스의 기능을 확장할 때 사용합니다. `extends` 키워드를 사용하여 상속할 클래스를 적어주고, `super`를 사용하여 상속한 클래스에 접근합니다.

```
class Person {  
    final String name;  
  
    Person(this.name);  
}  
  
class Employee extends Person {  
    final String id;  
  
    // 상속한 클래스의 생성자를 호출합니다.  
    Employee(this.id, String name) : super(name);  
}
```

클래스 - 추상 클래스

추상 클래스를 사용하면 함수의 형태만 정의하고, 함수의 구현은 이를 상속하는 다른 클래스에서 정의하도록 할 수 있습니다. **abstract** 키워드를 사용합니다.

```
abstract class Doer {  
    // Define instance variables and methods...  
  
    void doSomething(); // Define an abstract method.  
}  
  
class EffectiveDoer extends Doer {  
    void doSomething() {  
        // Provide an implementation, so the method is not abstract here...  
    }  
}
```

클래스 - 멤버 접근

클래스 멤버 (변수 및 함수)에 접근하려면 `.` 혹은 `?.` 을 사용합니다.

```
class Person {  
    final String id;  
    final String name;  
    Person(this.id, this.name);  
}  
  
var foo = Person('foo', 'Foo');  
// foo객체의 name 변수 값을 출력합니다.  
print(foo.name); // Foo  
  
Person? bar;  
// bar가 null이 아닐 때에만 name값을 출력합니다.  
print(bar?.name); // null
```

클래스 - 열거형 타입 (enumerated type)

한정된 수의 값을 표현할 때 사용합니다.

```
// 빨강, 초록, 파랑 중 하나가 될 수 있는 Color타입 enum 선언
enum Color { red, green, blue }
Color myColor = Color.red;
String message;

switch (myColor) {
    case Color.red:
        message = 'Color: red';
        break;
    case Color.green:
        message = 'Color: green';
        break;
    case Color.blue:
        message = 'Color: blue';
        break;
}
print(message); // Color: red
```

클래스 - 향상된 열거형 타입 (enhanced enumerated type)

열거형 타입에 인스턴스 변수 및 함수를 추가할 수 있습니다.

```
enum Vehicle {  
    car(tires: 4, passengers: 5, carbonPerKilometer: 400),  
    bus(tires: 6, passengers: 50, carbonPerKilometer: 800),  
    bicycle(tires: 2, passengers: 1, carbonPerKilometer: 0);  
  
    const Vehicle({  
        required this.tires,  
        required this.passengers,  
        required this.carbonPerKilometer,  
    });  
  
    final int tires;  
    final int passengers;  
    final int carbonPerKilometer;  
  
    int getCarbonFootprint() => (carbonPerKilometer / passengers).round();  
}
```

Dart 언어 주요 문법

예외 처리

예외 처리 - throw

- 코드에서 의도치 않은 문제가 발생한 경우, 예외(Exception)를 발생시켜 이후 동작을 멈추게 할 수 있습니다.
- 예외를 발생시킬 땐 **throw** 키워드를 사용합니다.

```
switch (command) {  
  case 'OPEN':  
    executeOpenCommand();  
    break;  
  case 'CLOSE':  
    executeCloseCommand();  
    break;  
  default:  
    // 정의되지 않은 명령을 받으면 예외를 발생시킵니다.  
    throw Exception('Unknown command: $command');  
}
```

예외 처리 - try~catch

- 예외가 발생했을 때 이를 적절히 처리하여 더 이상 전파되지 않도록 할 때 사용합니다.
- 예외 종류에 따라 다양한 방법으로 처리할 수 있습니다.

```
try {  
    breedMoreLlamas();  
} on OutOfLlamasException {  
    // A specific exception  
    buyMoreLlamas();  
} on Exception catch (e) {  
    // Anything else that is an exception  
    print('Unknown exception: $e');  
} catch (e) {  
    // No specified type, handles all  
    print('Something really unknown: $e');  
}
```

예외 처리 - finally

- `try~catch` 구문이 완료된 후 마지막에 수행할 코드를 작성할 수 있습니다.

```
try {  
    breedMoreLlamas();  
} catch (e) {  
    print('Error: $e'); // Handle the exception first.  
} finally {  
    cleanLlamaStalls(); // Then clean up.  
}
```

Dart 언어 주요 문법

동시성

동시성

- 함수의 실행이 끝날 때까지 기다리지 않고 비동기로 실행할 수 있는 블록을 만들 수 있습니다.
- `await`을 사용하여 `Future`에서 값을 반환하기를 기다릴 수 있으며, 이는 `async` 함수에서만 사용할 수 있습니다.

```
// await을 사용하기 위해 async 키워드를 추가합니다.  
void main() async {  
  // 5초 뒤에 값을 반환하는 Future를 생성합니다.  
  final messageFuture = Future.delayed(  
    Duration(seconds: 5), () => "delayed message");  
  
  // messageFuture가 값을 반환할 때까지 기다립니다.  
  final message = await messageFuture;  
  print(message); // delayed message  
}
```

`.then`을 사용하면 `async-await`을 사용하지 않고 실행 결과 및 오류 처리를 구현할 수 있습니다.

```
Future.value(10).then((result) {  
  print('Result: $result');  
}, onError: (e, stack) {  
  print('Error: $e');  
});
```

실습: Vehicle.fuelUp() 함수 구현 완성하기

- 기름을 넣을 양인 liters를 인수로 받고, 연료 주입이 가능하다면 주유를 시작합니다.
- 주입 후에는 남은 연료량을 함수의 반환값으로 전달합니다.

```
void main() async {  
    final vehicle = Vehicle(0, 40);  
    await vehicle.fuelUp(10);  
}  
  
class Vehicle {  
    var currentFuelInLiters;  
    final maxFuelInLiters;  
  
    Vehicle(this.currentFuelInLiters, this.maxFuelInLiters);  
  
    Future<int> fuelUp(int liters) async {  
        // TODO: 기름을 넣을 공간이 없으면 예외를 던지고, 그렇지 않다면 연료 주입 후 차량의 기름양을 반환  
    }  
}
```

앱 동작 변경해보기

앱 동작 변경해보기

지난 시간에 만들었던 hello_world 프로젝트를 연 후, 다음과 같이 버튼을 눌렀을 때 숫자가 2씩 증가하게끔 `_incrementCounter()` 함수를 변경해봅니다.

```
void _incrementCounter() {  
  setState(() {  
    _counter+=2;  
  });  
}
```

코드 변경 후 저장 버튼을 누르면, 변경된 동작이 바로 앱에 반영되므로 (Hot Reload) 앱을 재시작하지 않아도 됩니다.

실습: 앱 동작 변경해보기

- 버튼을 눌렀을 때 숫자가 5배씩 증가하게끔 `_incrementCounter()` 함수를 변경해봅니다.
- 주석을 사용하여 수정한 부분에 대한 추가 설명을 넣어줍니다.