0장

앱 개발 방식

1.네이티브앱

모바일 os 에서 플랫폼에서 요구하는 네이티브 언어로 개발된 앱

2.모바일 웹

모바일 크기에 맞게 화면을 제공하는 반응/적응형 웹(pc중심)

3.모바일 웹 앱

모바일 웹과 비슷하지만 구동방식이 앱처럼 보이게 한 웹(모바일중심)

4.하이브리드 앱

네이티브+웹

컨텐츠 영역은 웹앱, 패키징은 모바일 운영체제별로 구현하는 방법

5.크로스 플랫폼 앱

하나의 네이비트 언어로 안드로이드와 ios플랫폼에서 동작 할 수 있도록 개발된 앱, 네이티브 언어가 아닌 언어로 작성된 코드를 플랫폼이 이해할 수 있는 코드로 변환 시켜주는 역할

플러터 특징

-구글에서 개발한 개발 플랫폼으로, Dart 언어를 사용하며, Skia 그래픽 엔진을 사용해서 다양한 플랫폼에서 구동 가능한 앱을 만들 수 있음

1.크로스플랫폼 지원

2.고성능

3.사용자 정의 UI 구성 용이

4.빠른 개발

5.강력한 커뮤니티 및 생태계

1장

환경 구성 검사

플러터의 환경이 잘 구성되었는지 확인해야 함

- flutter docter 사용
  - 콘솔에서 **flutter doctor** 실행
  - 안드로이드 스튜디오의 Terminal 탭을 누른 후 **flutter docter** 실행
  - 프로젝트 탐색기 - **pubspec.yaml** 선택 후 오른쪽 상단 **Flutter doctor** 선택

2장

변수타입

- 데이터(값)을 저장하는 장소
- 변수 종류를 타입(type) 또는 자료형이라 함
- Dart는 크게 숫자형과 문자열의 기본 변수 타입을 제공 함

int(정수), double(실수), string(문자열), bool(참,거짓),

```
String name;                    // 변수 선언
name = '홍길동';                // 값 할당

bool b = true;                  // 참
bool b2 = i < 10;               // i가 10보다 작을 경우 참, 같거나 클 경우 거짓
bool b3 = s.isEmpty;            // s가 비어있을 경우 참, 비어있지 않을 경우 거짓

int i = 10;                     // 정수
double d = 10.0;                // 실수

num a = 10;                     // 정수와 실수를 같이 쓰고자 할 때 'num'
num b = 20.0;
```

var(타입추론)

```
var i = 10;                 // 추론에 의해 var는 int
var d = 10.0;               // var는 double
var s = 'hello';            // var는 String
var s2 = "hello";           // var는 String
var b = true;               // var는 bool
var b2 = i < 10;            // var는 bool
var b3 = s.isEmpty;         // var는 bool
```

※ var를 쓰는 이유? 가독성, 유연성, 일관성 향상

final, const

- 값이 한번 설정되면 바꿀 수 없는 것을 상수

```
final String name = '홍길동';
name = '임꺽정'; 에러

final name = '홍길동';          // String을 생략할 수 있음
```

※ final과 const의 차이점 : final은 동작중에 값이 고정되나 const는 컴파일 시점에서 값이 고정 됨

타입검사 - is, is!

-변수나 객체의 타입이 특정 타입인지 확인하는 과정

- is : 같은 타입이면 true, 아니면 false
- is! : 같은 타입이면 false, 아니면 true
- 연산 결과 : bool형

형변환 - as

```
var c = 30.5;                   // var는 double
int d = c as int;           // 오류 발생. 이유는 double → int 안됨
```

※ 연산 결과 : 형변환 된 타입

※ C# 언어와 유사

※ 형 변환의 필요성 - 객체지향 언어의 객체(인스턴스)의 형 변환이

필요하기 때문

(!주의! 강의 중 설명한 것으로 출제 - 힌트 : 상속, LLM을 활용할 것)

익명함수

이름이 없는 함수. 보통 콜백 함수로 사용

- 익명 함수는 함수 코드를 변수 값처럼 취급해 변수에 담을 수 있도록 함

```
var list = [1, 2, 3, 4];
list.forEach((item) {                    // (item) { ... } 익명 함수
  print(item);
});
```

람다식

한 줄로 표현할 수 있는 간단한 함수를 =>를 이용해 표현하는 문법입니다

```
(number) => number %2 == 0;
```

```
var list = [1, 2, 3, 4];
list.forEach((item) => print(item));
```

※ 익명 함수보다 단순한 함수 코드일 경우 람다 형태로 간단하게 표현 가능

선택 매개변수

- 선택적으로 함수에 전달하고자 하는 매개변수의 경우 { } 로 표시

```
void somthing({String name, int age}) {}

void main() {
  something(name: '홍길동', age: 10);
  something(name: '홍길동');
  something(age: 10);
}
```

클래스 정의

객체의 설계도, 설계도에 의해 만들어진 형태를 객체, 또는 클래스를 인스턴스화 했다고함

```
class Person {
  String name;
  int age;

  void addOneYear() {
    age++;
  }
```

상속 vs 인터페이스

상속

상속은 한 클래스가 다른 클래스로부터 속성과 메서드를 물려받는 것입니다.

- `extends` 키워드 사용
- 단일 상속만 가능 (한 번에 하나의 클래스만 상속)
- 부모 클래스의 코드를 재사용
- 자식 클래스는 부모 클래스를 "is-a" 관계로 가짐

✅ 상속 (`extends`) – 장단점

✔ 장점

장점	설명
✅ 코드 재사용	부모 클래스의 기능을 그대로 사용하거나, 약간만 수정하여 재사용 가능
✅ 구조 명확	계층 구조(class hierarchy)가 뚜렷해서 이해하기 쉬움
✅ 공통 기능 묶기 용이	여러 클래스에 공통 기능을 하나의 상위 클래스로 묶을 수 있음

❌ 단점

단점	설명
❌ 단일 상속만 가능	Dart에서는 클래스 하나만 상속 가능 (다중 상속 불가)
❌ 강한 결합도	부모 클래스 변경 시, 자식 클래스에 영향이 큼
❌ 유연성 부족	구조가 고정되면 다른 기능 추가가 어려워질 수 있음

인터페이스

인터페이스는 클래스가 특정 기능을 구현하도록 강제하는 설계도 같은 개념

Dart에서는 별도의 `interface` 키워드가 없고, 기존 클래스나 **abstract class**를 인터페이스처럼 사용합니다.

- `implements` 키워드 사용
- 여러 개의 인터페이스를 동시에 구현 가능 (다중 구현
- 인터페이스를 구현할 경우, 모든 메서드를 반드시 재정의해야 함
- "can-do" 관계 (예: 이 클래스는 `Comparable`을 구현할 수 있음)

✅ 인터페이스 (`implements`) – 장단점

✔ 장점

장점	설명
✅ 다중 구현 가능	여러 개의 인터페이스를 동시에 구현할 수 있음 (유연한 구조 설계)
✅ 낮은 결합도	인터페이스는 구현을 강제할 뿐, 내부 동작은 독립적으로 작성 가능
✅ 명확한 역할 분리	클래스의 책임(Roles)을 잘게 나눌 수 있음 (SOLID 원칙 중 Interface Segregation)

❌ 단점

단점	설명
❌ 재사용 불가	인터페이스는 구현을 가지지 않으므로 코드 재사용이 불가
❌ 구현 부담	모든 인터페이스 메서드를 반드시 재정의해야 하므로 코드량 증가 가능
❌ 복잡한 설계 가능성	너무 많은 인터페이스가 생기면 구조가 오히려 난해해질 수 있음

상속: 부모 클래스의 기능을 그대로 또는 일부 수정해서 재사용하고 싶을 때

인터페이스: 특정 기능을 강제 구현하도록 하고 싶을 때, 또는 여러 역할을 한 클래스에 부여하고 싶을 때

✅ 상속 vs 인터페이스 요약 비교표

항목	상속 (`extends`)	인터페이스 (`implements`)
목적	기능 재사용	계약(Contract), 기능 구현 강제
키워드	`extends`	`implements`
다중 구현 여부	❌ 단일 상속만 가능	✅ 다중 인터페이스 구현 가능
메서드 재정의 여부	선택적 (필요 시 오버라이드)	필수 (모든 메서드 구현 필요)
예시 관계	`Dog is an Animal`	`Printer implements Printable`

List, Map, Set 표기법

Lsit

순서가 있는 연속된 자료를 표현할 때 사용함

List<String> items = ['짜장','라면','볶음밥'];

var items = ['짜장','라면','볶음밥'];

print(items[2]);

print(items[3]); // 에러!

var items = ['짜장','라면','볶음밥'];

var items2 = ['떡볶이',...items,'순대']; // 떡볶이, 짜장, 라면,순대

Map

키와 값으로 쌍 지어진 사전 형태의 자료 구조를 표현할 때 사용함
Dictionary라고도 함

```
var cityMap = {
  '한국': '부산',
  '일본': '도쿄',
  '중국': '북경'
};

cityMap['한국'] = '서울';

print(cityMap.length);      // 3
print(cityMap['중국']);     // 북경
print(cityMap['미국']);     // null

cityMap['미국'] = '워싱턴';  // 새로운 값 추가
print(cityMap['미국']);     // 워싱턴
```

Set

중복되지 않는 집합을 표현할 때 사용

```
// Set<String> citySet = { ...생략... };과 같음
var citySet = {'서울', '수원', '오산', '부산'};

citySet.add('안양');       // 추가
citySet.remove('수원');    // 삭제

print(citySet.contains('서울'));   // true
print(citySet.contains('도쿄'));   // false
```

함수형 프로그래밍 특징 : 일급객체, forEach, where, map

일급 객체

값이나 개체가 언어 내에서 동등한 대우를 받는다는 것을 "일급 객체"라 함

- 함수를 변수에 저장 할 수 있음
- 함수를 함수 인자에 넘길 수 있음
- 함수를 함수의 반환값으로 사용할 수 있음

5. 비교: forEach vs where vs map

특징	forEach	where	map
목적	각 요소에 대해 작업 수행	조건에 맞는 요소 필터링	각 요소를 변환하여 새로운 컬렉션 생성
반환 값	없음	새로운 필터링된 컬렉션	새로운 변환된 컬렉션
원본 데이터 변경 여부	변경되지 않음	변경되지 않음	변경되지 않음
사용 사례	요소 출력 또는 상태 업데이트	특정 조건을 만족하는 데이터 추출	데이터 가공 및 변환

컬렉션 if 또는 컬렉션 for

Dart에서 컬렉션 if와 컬렉션 for는 리스트, 맵, 셋과 같은 컬렉션을 동적으로 생성하거나 수정할 때 사용되는 강력한 기능입니다. 이 두 기능은 코드를 간결하고 유연하게 작성하는 데 매우 유용합니다.

```dart
void main() {
  bool isSpecialUser = false;

  var userList = [
    'User1',
    'User2',
    if (isSpecialUser) 'SpecialUser' else 'RegularUser',
  ];

  print(userList); // 출력: [User1, User2, RegularUser]
}
```

2. 컬렉션 for

특징

- 컬렉션 내에 `for` 문을 사용하여 반복적으로 요소를 추가할 수 있습니다.
- 기존 리스트나 셋의 데이터를 변환하거나 새로운 컬렉션을 생성하는 데 유용합니다.

```dart
void main() {
  var friends = ['John', 'Sam', 'Mike'];

  var family = [
    'Ryan',
    'Gigi',
    for (var friend in friends) '*new* $friend', // friends 리스트의 각 요소를 반복적으로 추가
  ];

  print(family); // 출력: [Ryan, Gigi, *new* John, *new* Sam, *new* Mike]
}
```

3장

name: 패키지의 이름입니다. 어떤 곳이든 필수적으로 포함시켜야 하는 속성입니다.

description: 해당 패키지의 대한 설명을 적는 곳입니다. 개인 패키지일 경우 옵션이지만 패키지를게시하려면 필수적으로 포함시켜야합니다

publish_to: 배포할 곳을 지정합니다. 기본 값은 https://pub.dev/이며 none은 패키지로서 배포를 하지 않겠다는 뜻입니다.

version: 패키지의 버전을 의미합니다. 기본값은 1.0.0+1 지정 안할 시

environment: Dart SDK 환경을 설정하는 속성입니다. sdk는 environment의 하위속성으로 sdk 버전을나타냅니다.

dependencies: 패키지의 의존성을 작성하는 곳입니다. 주로 외부 패키지를 가져다 쓰기 위해 사용됩니다.

dev_dependencies: 패키지가 사용하는 개발용 의존성을 작성하는 곳입니다.

1. StatelessWidget

특징

불변 상태: StatelessWidget은 상태가 없는 정적인 UI를 표현합니다.

간단한 구조: 생성자에서 값을 받아 빌드 메서드에서 UI를 렌더링합니다.

효율적: 상태 관리가 필요하지 않으므로 성능이 더 우수합니다.

사용 사례: 텍스트, 아이콘, 버튼 등 사용자 입력이나 데이터 변화에 영향을 받지 않는 요소.

동작 방식
build() 메서드는 한 번 호출되며, 이후 UI는 변경되지 않습니다.

주로 정적인 레이아웃을 정의할 때 사용됩니다.

2. StatefulWidget

특징

가변 상태: StatefulWidget은 상태를 가지며, 상태가 변경될 때마다 UI를 재빌드합니다.

State 객체 관리: 상태는 State 클래스에서 관리됩니다.

동적인 UI 구성: 사용자 입력이나 데이터 변화에 따라 UI가 업데이트됩니다.

사용 사례: 카운터, 폼 입력, 애니메이션 등 동적인 요소.

동작 방식
createState() 메서드로 상태 객체를 생성하며, 이후 setState()를 호출하면 UI가 재빌드됩니다.

사용자와의 상호작용이나 데이터 업데이트가 필요한 경우 사용됩니다.

3. StatelessWidget vs StatefulWidget 비교

특징	StatelessWidget	StatefulWidget
상태 관리 여부	불변 상태	가변 상태
성능	더 효율적	상대적으로 낮음
사용 사례	정적인 UI 요소	동적인 UI 요소
생명주기 관리	없음	`initState`, `dispose` 등 생명주기 메서드 포함
재구성 방식	한 번 빌드 후 고정	`setState()` 호출 시 재빌드

4장

Container
너비, 높이가 있는 영역
width, height, padding, margin, child 속성
※ child 속성으로 자식 위젯을 담을 수 있음

Column
수직 방향으로 위젯들을 배치

Row
수평 방향으로 위젯들을 배치

Stack
위젯들을 순서대로 겹치게 함

SingleChildScrollView
화면 크기를 넘어가면 스크롤이 생기게 함

ListView, ListTile
리스트를 표시하는 위젯
ListTile 위젯을 이용해 리스트 아이템을 쉽게 작성할 수 있음

GridView 열 수를 지정하여 그리드 형태로 표시하는 위젯

PageView
여러 페이지를 좌우로 슬라이드하여 넘길 수 있게 하는 위젯

AppBar, TabBar, Tab, TabBarView
AppBar에 TabBar를 배치하고 Tab 및 body에 TabBarView를 배치하여 탭으로 이동하는 화면을 구성할 수 있음

BottomNavigationBar
하단에 2~5개의 탭 메뉴를 구성할 수 있게 해주는 위젯

Center
중앙으로 정렬시키는 위젯

Padding
안쪽 여백을 표현할 때 사용하는 위젯

Align 자식 위젯의 정렬 방향을 지정하는 위젯

Expanded 자식 위젯의 크기를 최대한으로
확장시켜주는 위젯

SizedBox 자식 위젯을 특정 사이즈로 조정하고자 할
때 사용하는 위젯

Card 카드 형태의 모양을 제공해주는 위젯

ElevatedButton 입체감을 가지는 일반적인 버튼 위젯

TextButton 평평한 텍스트 버튼 위젯

IconButton 아이콘을 표시하는 버튼 위젯

FloatingActionButton 입체감 있는 둥근 버튼 위젯

Text 글씨를 표시하는 위젯

Image 이미지를 표시하는 위젯

Icon 아이콘을 표시하는 위젯

Progress 로딩 중이거나 오래 걸리는 작업을 표시할 때 사용하는 위젯

CircleAvatar 프로필 화면 등에 사용되는 원형 위젯

5장

TextField

“글자를 입력받는 위젯”
InputDecoration으로 다양한 입력 형태 선택

CheckBox와 Switch

“선택 체크, 체크해제를 지원하는 위젯”

Radio와 RadioListTile

“선택 그룹 중 하나를 선택할 때 사용하는 위젯”

DropDownButton

“여러 아이템 중 하나를 고를 수 있는 형태의 위젯”

AlertDialog
“사용자 확인을 요구하거나 메시지를 표시하는 용도”

DatePicker
“날짜를 선택할 때 사용”

TimePicker
“시간을 선택할 때 사용하는 위젯”

GestureDetector, InkWell
“글자나 그림 같이 이벤트 속성이 없는 위젯에서 이벤트를 사용하고자 할 때 사용”

InkWell은 선택시 선택된 물결 표시
GestureDetector는 물결 표시 없음

Hero
“페이지 전환시 연결되는 애니메이션 지원”

AnimatedContainer

“한 화면 내에서 변경된 프로퍼티에 의해
애니메이션”

SliverAppBar와 SliverFillRemaining
“화면 헤더를 동적으로 표현”

SliverAppBar와 SliverList
“ListView를 사용하며 Sliver 효과를 주고자 할때”

쿠퍼티노 디자인(1/3) - 쿠퍼티노 기본 UI
“머티리얼 디자인 대신 쿠퍼티노 디자인 적용”

AppBar → CupertinoNavigationBar
Switch → CupertinoSwitch
ElevatedButton → CupertinoButton

쿠퍼티노 디자인(2/3) - CupertinoAlertDialog
“쿠퍼티노 스타일의 AlertDialog”

쿠퍼티노 디자인(3/3) - CupertinoPicker
“iOS에서 자주 사용되는 피커”

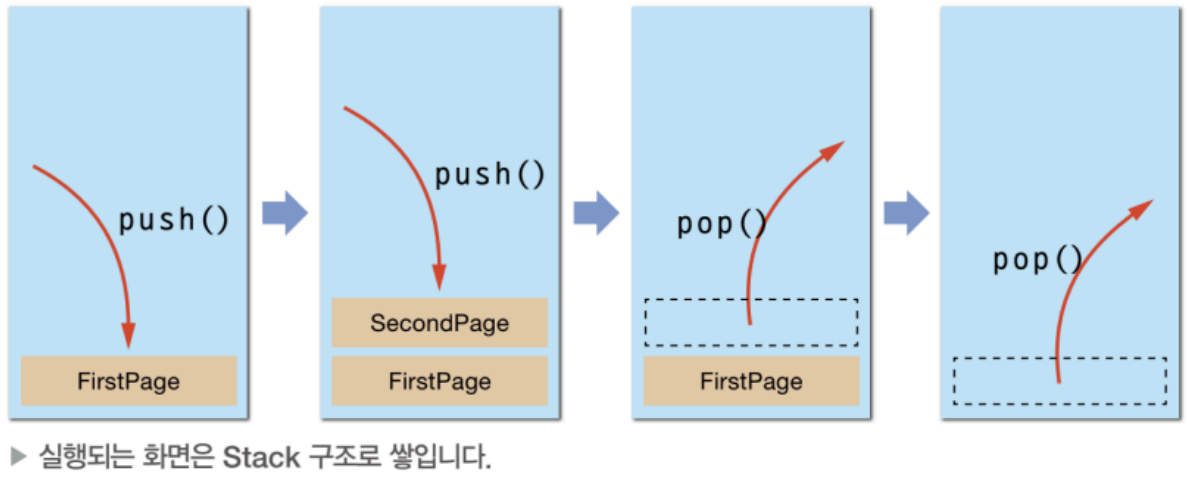
6장

내비게이션의 동작성

1. 기본 개념

Navigator는 화면(또는 페이지)을 스택(stack) 형태로 관리합니다.

새로운 화면으로 이동할 때 해당 화면을 스택에 추가(push)하고, 이전 화면으로 돌아갈 때 스택에서 제거(pop)합니다



▶ 실행되는 화면은 Stack 구조로 쌓입니다.

push / pop 사용법

1. Navigator.push

- `push` 메서드는 새로운 화면(Route)을 현재 화면 위에 추가하여 이동합니다.
- 스택(stack) 구조를 사용하며, 새로운 화면이 스택의 맨 위에 쌓입니다.

```
Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => SecondScreen()),
);
```

2. Navigator.pop

- `pop` 메서드는 현재 화면(Route)을 스택에서 제거하고 이전 화면으로 돌아갑니다.
- 스택의 맨 위에 있는 화면을 제거하여 이전 화면이 다시 활성화됩니다.

```
Navigator.pop(context);
```

routes를 이용한 방법

1. Named Routes 이용하기

```
void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (context) => FirstScreen(), // 첫 번째 화면
      '/second': (context) => SecondScreen(), // 두 번째 화면
    },
  ));
}
```

```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('First Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, '/second'); // Named Route로 이동
          },
          child: Text('Go to Second Screen'),
        ),
      ),
    );
  }
}
```

8장

Null Safety: 변수의 null 가능성을 명시적으로 선언하여 컴파일 타임에 null 관련 오류를 사전에 방지하는 기능

- Non-Nullable 타입: 기본적으로 모든 변수는 null을 허용하지 않습니다.

- Nullable 타입: 변수 타입 뒤에 ?를 붙이면 null을 허용합니다.

ex) `int? nullableNumber = null; // null 값을 가질 수 있음`

*Generic 타입에서도 Nullable과 Non-Nullable을 명시적으로 선언할 수 있습니다.

-late 키워드

초기화를 미룰 필요가 있는 Non-Nullable 변수는 late 키워드를 사용합니다. 이는 반드시 초기화 전에 값을 설정해야 합니다.

ex)
```
late String message;
message = "Hello, Flutter!";
```

- Null 체크 연산자

! 연산자: 변수가 null이 아님을 확신할 때 사용합니다.

```
String? nullableMessage = "Hello";
print(nullableMessage!); // nullableMessage가 null이 아니라고 확신
```

? 연산자: Nullable 객체의 멤버에 안전하게 접근합니다.

```
String? nullableMessage;
print(nullableMessage?.length); // null이면 null 반환
```

?? 연산자: 변수 값이 null일 경우 대체값을 제공합니다.

```
String? nullableMessage;
print(nullableMessage ?? "Default Message");
// nullableMessage가 null일 경우 "Default Message" 반환
```

- 초기화 연산자

??= 연산자를 사용하여 변수가 null일 때만 값을 설정할 수 있습니다.

```
String? nullableMessage;

nullableMessage ??= "Hello, Flutter!";
print(nullableMessage); // "Hello, Flutter!"
```