**[0장 플러터 개발 환경 준비하기 : 2문항]**

**[1. 앱 개발 방식]**

1. 스마트폰 운영체제 : 안드로이드, iOS
2. **멀티 플랫폼 : 안드로이드, iOS 둘 다 돌아가는 앱**
3. 앱 개발 방식 3가지와 그 특징

|  |  |  |  |
| --- | --- | --- | --- |
| 방식 | **네이티브** 방식 | **하이브리드** 방식 | **크로스 플랫폼** 방식 |
| 개발 | 운영체제별 네이티브 개발 환경 | 하이브리드 개발 도구  Ex) 코드바 | 크로스 플랫폼 도구  Ex) 플러터 |
| 앱 결과물 | 네이티브 | 웹UI + 네이티브 코어 | 네이티브 |

|  |  |  |
| --- | --- | --- |
| 운영체제 | 안드로이드 | iOS |
| 개발 도구 | 안드로이드 스튜디오 | Xcode (맥OS에서만 개발 가능) |
| 개발 언어 | 자바 / 코틀린 | Swift / Object-C |

1. 네이티브 방식

**<한 번 개발해서 여러 OS에서 모두 사용 가능한 앱을 개발하는 방법 2가지>**

1. **하이브리드 Hybrid**: 웹 기술(HTML, CSS, JavaScript)로 개발한 후, 네이티브 앱처럼 패키징하여 실행하는 방식(WebView 활용).  
   1) 장점  
   - 개발 속도가 빠르고 비용이 저렴  
   - 유지보수 및 업데이트가 쉬움  
   2) 단점WebView 기반이라 성능이 떨어짐  
   네이티브 앱처럼 부드러운 UI/UX 구현이 어려움  
   네이티브 기능(카메라, GPS 등) 사용이 제한적
2. **크로스플랫폼 cross-platform**: 하나의 코드로 여러 플랫폼(iOS, Android)에서 실행되며, 네이티브 성능을 최대한 활용하는 방식. ex) unity, flutter  
   1) 장점  
   - **네이티브 수준의 성능과 UI/UX 제공** (빌드(앱을 완성하는 과정)할 때 네이티브 코드로 변환되기 때문)  
   - **단일 코드로 여러 플랫폼 지원** (개발 효율성 높음)  
   - **네이티브 API 접근 용이**2) 단점  
   - 네이티브 앱보다는 성능이 약간 떨어질 수 있음  
   - 프레임워크(Flutter, React Native 등) 학습 필요  
   - **특정 기능 구현 시 네이티브 코드 추가 필요**

**[2. 플러터 특징]**

1. 정의  
   : google에서 개발한 **개발 플랫폼**   
   : Skia 그래픽 엔진을 사용하여 다양한 플랫폼에서 구동 가능한 앱 제작 가능  
   : 구글의 차세대 운영체제 퓨시아(Fuchsia)의 공식 프레임워크
2. **플러터의 특징**  
   1) **크로스 플랫폼** 지원 (안드로이드, iOS, 웹, 원도우 데이스톱 앱, 리눅스 지원)  
   2) **고성능** – 화면 구성에 필요한 UI 구성 요소를 플러터가 직접 그려 속도 빠름 (초당 60프레임 애니메이션 보장)  
   3) **사용자 정의 UI 구성 용이** – 안드로이드의 머티리얼 디자인과 iOS의 쿠퍼티노 디자인의 UI 구성 요소를 모두 제공하고 그걸 조합 가능.  
   4) **빠른 개발** (피드백과 디버깅 – 핫리로드Hot-Reload)  
    \* **핫리로드** : 수정한 코드를 즉시 앱에 반영하는 기능  
   5) 강력한 커뮤니티 및 생태계  
   6) **낮은 진입장벽** (네이티브 개발에 비해 배워야 할 개념이 상대적으로 적음)  
   7) **훌륭한 개발 도구 지원** – 안드로이드 스튜디오, 인텔리제이, vsCode에서도 플러그인 설치하여 개발 가능  
   8) **Dart 언어** 사용  
   \* Dart : 웹프론트엔드 개발 언어였다 현재는 플러터 개발에 주로 사용됨. 자바/js와 유사한 문법.  
    **JavaScript의 동적 언어의 단점인 성능과 일관성의 문제를 보완하기 위해서 Google에서 설계한 정적언어**

**[1장 생애 첫 모바일 앱 생성하기 : 1문항]**

**[3. 환경 구성 검사]**

1. **flutter doctor**  
   : **플러터를 개발할 환경 구성이 잘 되었는지 검사해주고 가이드를 해주는 명령어**
2. flutter doctor 명령 실행방법 3가지  
   1) 터미널(cmd)에서 명령어 실행  
   2) 안드로이드 스튜디오 하단에 있는 “Terminal” 탭에서 명령어 실행   
   \* 위 두 방법은 윈도우에 환경 변수를 등록해야 작동함.(플러터 SDK bin폴더 등록)  
   3) 안드로이드 스튜디오 왼쪽 프로젝트 창에서 “프로젝트명 > pubspec.yaml” 클릭, 우측 상단에 표시된 4가지 링크 중 “Flutter doctor” 클릭, 출력 결과는 하단 “Flutter” 창에서 확인 가능  
   \* 이 방법은 플러터 SDK를 환경 변수에 추가하지 않아도 작동하여 편리함.  
   \* doctor --verbose 옵션이 붙은 명령이 자동 수행되어 더 자세한 결과를 보여줌
3. flutter doctor 명령어 실행 결과  
   0) V : 정상, ! : 경고, X : 설정 안 됨  
   1) 플러터 SDK 설정 정보  
   2) 안드로이드 스튜디오 버전 정보  
   3) 안드로이드 개발을 위한 SDK 설정 정보  
   4) Xcode, VSCode 등 다른 툴의 설치 정보 표시  
   5) 연결된 기기나 에뮬레이터가 있는지에 대한 정보

**[2장 다트 문법 : 10문항]**

**[4. 변수 타입(자료형)]**

1. **type/자료형** : 변수 종류  
   - **int** : 정수, **double** : 실수, **String** : 문자열, **bool** : true/false  
   **\* String 자료형은 큰따옴표/작은따옴표 모두 사용 가능**  
   int a = 5; // 정수형  
   double b = 3.1415; // 실수형  
   String s = '문자열1'; String s = "문자열2"; // 작은따옴표/큰따옴표 둘 다 가능  
   bool b1 = true; bool b2 = a>5; bool b3 = s.isEmpty; // s가 비어있으면 참, 비어있지 않은 경우 거짓  
   \* double 타입에 int 타입 대입 시 오류 발생(업캐스팅 미지원)  
   - **num** : 숫자형(int, double을 포함)  
   num n1 = 10; // num은 정수와 실수 모두 넣을 수 있음  
   n1 = 10.25;
2. **타입 추론**- **var** : 가독성, 유연성, 일관성 향상1) **선언 및 초깃값 대입 동시 진행 -> 타입 추론 발생 (컴파일 시점)**var i = 10; // Dart 언어가 변수 i의 타입을 int로 추론.  
   i = 20; // 변수 i의 타입이 int이므로 정수값 20 대입 가능  
   i = ‘hello’; // 오류! 변수 i의 타입이 int로 확정되었으므로 String값은 대입 불가  
   2) **변수 선언만 한 경우 -> dynamic 타입으로 추론(런타임에 허용)**var j; // 초기값이 없으므로 dynamic 타입(동적 타입, 런타임에서 결정됨)  
   j = 10; j = “Hi”; j = true; // 변수 j의 타입이 고정되지 않았으므로 다양한 타입의 값 대입 가능. j.runtimeType으로 현재 들어 있는 값의 타입 확인 가능  
   ※ [참고] **dynamic**  
   1) dynamic의 단점  
   - 코드 안정성 문제. 컴파일 타임에 타입 검사 X -> 런타임에 오류 가능성 있음, 버그 찾기가 어려움  
   - IDE에서 자동완성, 타입 힌트 가능이 제한됨  
   - 정적 타입 언어의 장점을 없앰.  
   2) dynamic을 사용하는 경우  
   - JSON에서 값 꺼낼 때 ex) Map<String, dynamic>  
   - 여러 타입을 받아야 할 함수 매개변수 ex) void printValue(dynamic value)  
   - 아주 복잡한 로직에서 타입을 확실히 모를 때 ex) 일부 유연한 로직 처리  
   \* 현대의 PL은 타입 추론(컴파일러가 변수의 타입을 자동으로 추론)을 제공함

**[5. final, const]**

1. 상수  
   : 값이 한 번 설정되면 바꿀 수 없는 것. 변하지 않는 값  
   : final, const 키워드로 상수 선언  
   1) final과 const 키워드의 공통점  
   - **재선언, 재할당 불가**  
   - **상수 선언 시 변수의 타입(String) 등은 생략이 가능. (타입추론 가능)**

|  |  |  |
| --- | --- | --- |
| 키워드 | 값 결정 시점 | 특징 |
| **final** | **런타임** | 한 번만 할당 가능, 실행 중 계산되는 값(런타입 상수) 대입 가능 ex) final now = Datetime.now();  선언 후 초기화 가능. 단, 값 변경은 불가  참조하는 객체의 내용 변경 가능 |
| **const** | **컴파일타임** | 완전히 불변, 컴파일타임에 결정되는 값만 대입 가능,  선언과 동시에 초기화 필수  참조하는 객체의 내용 변경 불가 |

* 컴파일타임  
  : 코드를 실행하기 전에, 코드를 분석하고 번역할 때
* 런타임  
  : 실제로 코드가 실행될 때

**[6. 타입검사, 형변환 (!주의! 강의 중 설명한 것으로 출제 - 힌트 : 상속, LLM을 활용할 것)]**

1. 타입 검사 : 연산 결과가 bool타입  
   **is** : 같은 타입이면 true, 아니면 false  
   **is!** : 같은 타입이면 false, 아니면 true  
   var v = 5;  
   print(v is int); // true, var로 선언과 동시에 초기화하여 컴파일 타임에 이미 int로 확정되어 불필요한 비교라고 경고 표시됨.  
   print(v is double); // false, int와 double은 서로 다른 타입. dart.pad는 dart언어를 js로 변환하기 때문에 true로 나오지만 그건 오류임.  
   print(v is String); // false  
   print(v is bool); // false
2. 형변환(type casting) **as** : 연산 결과가 형변환된 타입  
   : 다른 타입끼리는 변환이 안 되고 더 상위 개념으로 변환 가능  
   : **상속** 구조를 가지고 있는 타입(int-num)을 분기할 때 씀: int, double 등의 타입이 아니라 dynamic이나 object같이 타입이 불확실한 **객체(인스턴스)의 형변환** 시 사용  
   ex) var c = 30.5; int d = c as int; // 오류 발생. double타입의 변수 c는 as를 사용하여 int로 타입변환 불가. double과 int는 num클래스를 상속받은 형제관계이기 때문!  
   int a = 5; num b = a as num; // int는 부모타입인 num타입으로 형변환 가능  
   dynamic c = 5; int d = c as int; // 가능, **타입이 확정되지 않은 dynamic 타입은 다운 캐스팅이 가능**

**[7. 익명함수, 람다식]**

1. 함수  
   : 특별한 기능을 하는 코드 묶음.  
   \* 모듈화 : 코드를 함수로 묶는 것.  
   0) 함수와 메서드의 차이  
   - 함수 : 전역에서 접근 가능  
   - 메서드 : 클래스 내부 또는 인스턴스를 통해서만 접근 가능  
   1) 함수 형태  
   반환타입 함수명(매개변수타입 매개변수명) {  
    return 반환값;  
   }  
   \* 인수 : 함수 호출 시 매개변수로 입력되는 값  
   \* 매개변수는 없거나 여러 개일 수 있음.  
   \* 반환타입 void 지정 시, return 생략 가능.  
   \* **반환 타입 생략 가능. (타입 추론)**
2. 익명 함수 anonymous function  
   **(인수명){실행코드 또는 반환값}**  
   : 함수 코드를 변수 값처럼 취급해 변수에 담을 수 있도록 함.  
   : 일반 함수에서 반환타입과 함수명을 생략한 형태 (**return 키워드 필요**)  
   var list = [1,2,3,4];  
   list.forEach(**(item) {  
    print(item);  
   }**); **// (item) {…} : 익명함수**
3. 람다lambda 표현식 (람다식, 화살표 함수)  
   **(인수명) => 실행코드 또는 반환값**  
   : 익명 함수보다 단순한 함수 코드일 경우 람다 형태로 간단하게 표현 가능  
   : 매개변수가 1개일 땐 괄호 생략 가능. // 인수명 => 실행코드  
   : **람다식에서는 반환값이 자동 반환되므로 return 키워드 사용 금지!**: **람다식(화살표 함수)는 익명함수의 한 형태임.**ex1)  
   (number) => number%2 == 0;   
   ex2)   
   var list = [1,2,3,4];  
   list.forEach(**(item) => print(item)**);

**[8. 선택 매개변수]**

1. **선택 매개변수** (이름이 있는 매개변수, Named Parameter)  
   : 함수 정의에서, 선택적으로 함수에 전달하고자 하는 매개변수의 경우 {}로 표시   
   **반환타입 함수명(**필수 매개변수, **{선택 매개변수}) {}**: **호출 시 매개변수명:을 값 앞에 꼭 써야 함**. -> 옵션이 많을 때 가독성 좋음  
   : 필수 매개변수와 선택 매개변수 함께 사용 가능. 단, **필수 매개변수가 무조건 앞**임.  
   : **선택 매개변수는 기본값 지정 가능**.  
   : **기본값이 지정되지 않은 선택 매개변수에 값을 전달하지 않으면 null이 들어감.  
   (null처리 필수!)**void something(**{String? name, int age=10}**) {}  
   void something2(String? name, **{int? age}**) {}  
   void main() {

something(name:’홍길동’, age:10); // 홍길동, 10

something(name:’홍길동’); // 홍길동, 10 (생략 시 기본값으로 지정됨)  
 something(age:10); // null, 10  
 something();  
 something2(age:10); // 에러, 필수 매개변수가 생략됨  
 something2(10); // 에러, 선택 매개변수명이 없음.

}

**[9. 클래스 정의]**

1. 객체 object (인스턴스 instance)  
   : 저장 공간에 할당되어 값을 가지거나 식별자에 의해 참조되는 공간  
   : 변수, 자료 구조, 함수 또는 메서드 등이 객체가 될 수 있음.  
   : 객체를 메모리에 작성하는 것 -> 인스턴스화  
   : 고유한 쓰임새를 상태와 행위로 정의한 것  
   **상태 = 값(필드)  
   행위 = 기능(메서드)**-> 필드(값)과 메서드(기능)으로 구성된 단위.
2. 객체 지향 프로그래밍  
   : 객체와 객체의 상호작용으로 프로그램이 동작하게 하도록 프로그래밍하는 것
3. 클래스  
   : 객체의 설계도.  
   : 클래스class에서 객체(=인스턴스instance)를 생성하는 것 -> 인스턴스화  
   1) 클래스 정의  
   class 클래스명 { // 클래스명은 대문자로 시작  
    필드; // 프로퍼티 property라고도 함  
    메서드;  
   }  
   2) 클래스에서 인스턴스 생성 : new 키워드  
   var 레퍼런스변수명 = **new** 클래스명(); **// new 키워드는 생략 가능**3) 객체의 프로퍼티/메서드 접근 : .연산자  
   **인스턴스변수.필드명;  
   인스턴스변수.메서드명();**
4. **접근 지정자**   
   : **\_로 private를 표현**. java에 있는 그런 접근 지정자 키워드는 따로 없음.: 변수/메서드/클래스 앞에 ‘\_’를 붙이면 외부에서 접근 불가  
   : 붙이지 않으면 외부에서 접근 가능  
   : 외부라는 것은 해당 클래스가 정의되어 있지 않은 다른 파일을 말함.  
   : 접근 지정자를 사용하더라도 해당 클래스가 정의되어 있는 파일 내에서는 직접 접근 가능  
   : \_가 붙은 프라퍼티(속성)을 외부에서 접근하려면 게터/세터가 필요함.
5. 생성자  
   : 클래스명과 같은 이름의 메서드, 인스턴스화할 때 호출됨.  
   : **dart는 기본 생성자(클래스명();)를 제공**: dart는 생성자 오버로딩을 미지원 -> **사용자 정의 생성자 추가 시 기본 생성자 사용 불가. -> 선택 매개변수 사용 시 기본 생성자 형태도 호출 가능**Person({this.name, this.\_age}); // 자바에서 쓰는 생성자 형태도 가능함.  
     
   class Person {  
    String \_name = ""; // ?로 null 처리 String? \_name;  
    int \_age = 0;  
    // 생성자  
    Person(this.\_name, this.\_age);  
    // 게터 세터  
    String get name => \_name;  
    int get age => \_age;  
    void addOneYear() {  
    \_age++;  
    }  
   }  
   : dart에서는 클래스에 실제 필드가 없어도, 직접 게터/세터를 선언해서 “가짜 속성”처럼 만들 수 있음.

**[10. 상속 vs 인터페이스 (!주의! 강의 중 설명한 것으로 출제 - 힌트 : 서로 장단점이 반대임, LLM을 활용할 것)]**다형성 : 같은 타입(부모 타입)으로 다양한 형태의 객체를 다룰 수 있는 성질.

1. **상속 – 기능 물려받기**  
   : 부모 클래스의 상태(변수)와 행위(메서드)를 자식 클래스가 그대로 물려받는 것  
   : 자식 키워드에서 extends 키워드로 상속 받음  
   1) 슈퍼클래스(=부모클래스)  
   2) 서브클래스(=자식클래스)  
    - **부모클래스의 모든 필드+메소드 사용 가능**.   
   - super 키워드로 슈퍼클래스에 접근 가능. 자기 자신은 this로 참조.  
   3) 특징  
   - **코드 재사용 (기능 그대로 상속)**- 하나의 클래스만 상속 가능 (**단일 상속**)  
   - 부모클래스의 메서드를 @override로 재정의 가능  
   4) 상속의 필요성  
   - **공통 기능이 있는 클래스를 만들고, 중복 없이 관리하고 싶을 때**  
   - **다형성** 활용 (**부모 타입으로 자식 참조 가능**, 포함관계-자식이 부모에 포함됨-)  
   List<Hero> heroes = [h1, h2]; // h1 : 서브클래스, h2 : 부모클래스(Hero)  
   5) 상속의 단점  
   - **상속 구조가 복잡해지면 유지보수가 어려움**
2. **인터페이스(=추상클래스 abstract class) – 기능 약속**  
   : 함수 이름만 정의된 틀 (=추상 메서드의 집합)  
   \* 추상 메서드 : 선언만 되고 정의가 없는 메서드 ({} 생략됨)  
   **: 추상 클래스 자체는 인스턴스화 불가.**: **implements** 키워드로 추상 클래스를 구현한 후에 인스턴스화 가능  
   1) 특징  
   - **여러 인터페이스 동시 구현 가능** (implements A, B, C)  
   - 추상 클래스 구현 시 **모든 추상메서드를 재정의(@override) 필수**- **필드 상속 불가**, 메서드만 “틀” 제공 (코드 재사용 불가)  
   - 조합식 구조로 비교적 단순함.  
   - 다형성 활용(인터페이스로 구현 클래스 참조 가능)  
   List<Monster> monsters = [g1, b1];  
   monsters.forEach((m) => m.attack()); // 여기서 각 구현클래스의 메소드가 실행됨  
   2) 인터페이스의 필요성  
   - **여러 클래스에서 공통 규칙만 지정**하고 싶을 때  
   - **다른 기능들을 조합하거나 다중 기능 구현하고 싶을 때**
3. **믹스인  
   with 키워드** : 상속하지 않고 다른 클래스의 기능을 가져오거나 오버라이드 가능

**[11. List, Map, Set 표기법]**

1. 컬렉션  
   : 기본 제공하는 자료 구조
2. **List**: 연속된 자료형 표현. **중복값 허용, 순서 있음(인덱스로 값 접근)  
   List<데이터타입> 변수명 = [데이터1, 데이터2…];**  
   1) 명시적으로 데이터 타입 지정 -> 지정한 타입의 data만 저장 가능 **List<String>** items = [‘가’, ‘나’, ‘다’];  
   2) var 또는 List<dynamic> 타입 -> 서로 다른 타입의 자료형 저장 가능  
    **var** list = [1, 'hello', true];  
   3) 인덱스로 값 접근 시 0번 인덱스부터 시작. 길이 초과 불가.
3. **Map (=Dictionary)**  
   **: “키”:”값” 쌍 형태, 키로 값 접근, 순서 없음**  
   : **요청한 키에 해당하는 값이 없으면 null 반환**: **키는 중복 불가  
   Map<키타입, 값타입> 변수명 = {키1:값1, 키2:값2…};**  
   1) 명시적으로 데이터 타입 지정 -> 지정한 타입의 data만 저장 가능 **Map<String, String>** cityMap = {‘한국’:’부산’, ‘일본’:’도쿄’};  
   2) var 또는 Map<dynamic,dynamic> 타입 -> 서로 다른 타입의 자료형 저장 가능  
    **var** mixMap = {‘a’:1, 2:’도쿄’, true:false};  
   3) Map 값 변경/추가  
    **cityMap[‘한국’] = ‘서울’; // 값 변경** cityMap[‘미국’] = ‘워싱턴’; // 값 추가
4. **Set**: **중복되지 않는** 집합 표현, **순서 없음  
   Set<데이터타입> 변수명 = {값1, 값2…};**  
   1) 명시적으로 데이터 타입 지정 -> 지정한 타입의 data만 저장 가능 **Set<String>** citySet = {’서울’, ’도쿄’};  
   2) var 또는 Set<dynamic> 타입 -> 서로 다른 타입의 자료형 저장 가능  
    **var** mixSet = {‘a’, 1, true};  
   \* (주의!) 빈 Set이나 빈 Map 작성 시 값 없이 {}만 작성하면 Map으로 인식  
    var mySet = <String>{}; // Set<String> mySet = {}와 같음, Set으로 정상 인식  
    vat mySet2 = {}; // Map<dynamic, dynamic>으로 인식
5. 스프레드 연산자 **…**: 컬렉션을 펼쳐주는 연산자. 다른 컬렉션 안에 컬렉션을 삽입할 때 사용.  
   \* 동일한 컬렉션끼리만 스프레드 연산자 사용 가능.
6. 컬렉션 길이 : **.length**
7. List, Set에서 사용 가능한 함수 (Map은 불가)  
   1) .add(요소) : 요소를 추가 (List에서는 끝에, Set에서는 중복 자동 제거  
   2) .remove(요소) : 요소를 삭제 (key 넣으면 Map에서도 사용 가능)  
   3) .contains(요소) : 요소 포함 여부 확인

**[12. 함수형 프로그래밍 특징 : 일급객체, forEach, where, map]**



1. 함수형 프로그래밍 functional programming  
   : 자료 처리를 수학적 함수의 계산으로 취급하는 프로그래밍 패러다임  
   (상태(sum++, list.add())와 가변 데이터를 기피함  
   => 새로운 결과를 리턴하는 방식(map, where, reduce 등)을 선호)  
   현대 언어가 가지고 있는 특징 : 함수형 언어의 특징을 가짐
2. **일급 객체 (First-Class Citizen/Object)**: 값이나 개체가 언어 내에서 동등한 대우를 받는다는 것.  
   : 프로그래밍 언어에서 ‘**값처럼 자유롭게 다룰 수 있는 대상**’을 말함  
   : **변수에 저장, 함수의 인자로 전달, 함수의 반환값으로 사용 가능한 존재.  
   (이 3가지 조건이 모두 만족되어야 함)**ex) 함수, 클래스, 객체, 리스트, Map, 숫자, 문자열 등 값들  
   \* if, for 같은 제어문은 변수에 저장이 불가하므로 일급 객체가 아님.  
   // int를 매개변수로 받는 함수를 인자로 받는 함수

void something(Function(int i) f) {

f(10); // 전달받은 함수 f에 10을 전달해 실행

}

// 일반 사용자 정의 함수

void myPrintFunction(int i) {

print('내가 만든 함수에서 출력한 $i');

}

void main() {

// (1) 함수 이름 그대로 전달 → 일급 객체이므로 가능

something(myPrintFunction); // 출력: 내가 만든 함수에서 출력한 10

// (2) 람다식으로 전달 → 함수는 값처럼 다뤄짐

something((i) => myPrintFunction(i)); // 출력: 내가 만든 함수에서 출력한 10

// (3) 람다식으로 직접 print 사용

something((i) => print(i)); // 출력: 10

// (4) print 자체도 함수이므로 바로 전달 가능

something(print); // 출력: 10

}

1. forEach  
   **.forEach(E element){}** // E는 모든 타입이 가능함을 의미.  
   : 리스트의 각 요소를 꺼내서 **함수(익명 함수)** 형태로 처리 (일급 객체의 특징)  
   : 내부 반복 수행. // 일반 for문 : 외부반복  
   1) 일반 for문  
   final items = [1, 2, 3, 4, 5];  
   for (int i = 0; i < items.length; i++) {  
    print(items[i]);  
   }  
   2) forEach문  
   items**.forEach**(print); // 일급객체  
   items.forEach((e) { // 익명함수  
    print(e);  
   });  
   items.forEach((e) => print(e)); // 람다식
2. **where()**  
   : 조건을 필터링할 때 사용하는 함수  
   **Iterable<T> where(bool Function(T element) test)**: **컬렉션(List 등)의 각 요소를 하나씩 검사해서, 조건을 만족(true)하는 것만 남기고,  
   true인 것만 모아서 Iterable<T>로 반환함.**  
   \* 함수형 프로그래밍을 지원하는 함수들은 결과를 반복가능한 타입(Iterable)로 반환하며 메서드 체인(.연산자를 찍고 연속적으로 사용하는 것)으로 연결하여 사용 가능  
   var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
   nums.where((n) => n % 2 == 0).forEach(print); // 2 4 6 8 10
3. **map()**  
   : 반복되는 값을 다른 형태로 변환하는 함수  
   **Iterable<E> map<E>(E Function(T element) f)**: **컬렉션의 각 요소를 변형해서 새로운 값을 만듦**.  
   : 원래 값 -> 새 값으로 변환 (1:1 대응)  
   : 변형 결과들이 들어있는 Iterable 반환  
   nums.where((n) => n % 5 == 0).**map((n) => ‘숫자$n’)**.forEach(print);  
   // 숫자5 숫자10

**[13. 컬렉션 if 또는 컬렉션 for]**



1. 컬렉션 if  
   : List, Set, Map을 만들 때 **조건에 따라 특정 요소를 포함하거나 제외하는 문법  
   var list = [1, 2, if (조건) 값];**\* if (조건)이 true일 때만 뒤에 있는 값이 리스트에 포함됨  
   \* 조건이 false면 해당 항목은 무시됨  
   **\* if (조건) 뒤에 중괄호{}가 없는 것**에 유의!  
   bool promoActive = false;  
   print([1, 2, 3, 4, 5, if (promoActive) 6]); // [1, 2, 3, 4, 5]
2. 컬렉션 for  
   컬렉션 안에서 for문을 써서 **다른 컬렉션을 순회하며 요소 생성**하는 문법  
   **var result = [for (var item in list) 표현식];**\* 기존 리스트의 각 항목을 순회하면서 → 표현식 결과를 새로운 리스트에 넣음  
   var listOfInts = [1, 2, 3];  
   var listOfStrings = ['#0', for (var i in listOfInts) '#$i'];  
   print(listOfStrings); // ['#0', '#1', '#2', '#3']

**[3장 프로젝트 구조와 앱 구조 : 2문항]**

**[14. pubspec.yaml 관련 설명 (!주의! LLM을 이용해 상세히 파악할 것!)]**

1. pubspec.yaml  
   : Flutter(또는 Dart) 프로젝트의 **핵심 설정 파일**: 패키지 매니저(pub)가 참조하는 설정파일  
   : 프로젝트 전체의 **정보, 의존성, 환경, 자산(asset)** 등을 정의하는 곳
2. pubspec.yaml의 주요 구조와 의미  
   name: my\_project # 프로젝트 이름  
   description: A new Flutter project. # 설명  
   version: 1.0.0+1 # 앱 버전 (앱스토어, 빌드 관리용)  
     
   environment: # Dart SDK의 최소~최대 버전 지정  
    sdk: '>=3.0.0 <4.0.0'  
     
   dependencies: # **앱 실행 시 필요한 패키지(http, provider 등\_** flutter:  
    sdk: flutter  
    **http**: ^0.13.5  
     
   dev\_dependencies: **# 개발 중에만 사용하는 패키지 (예: 테스트, 코드 생성)**  
    **flutter\_test**:  
    sdk: flutter  
     
   flutter:  
    **assets: # 이미지, 폰트 등 자산 등록** - assets/images/  
   \* publish\_to : pub.dev에 공개할지 여부 설정 (none이면 비공개)
3. pubspen.yaml의 역할 요약

|  |  |
| --- | --- |
| 역할 | 설명 |
| 설정파일 | 앱의 전반적인 설정을 기록 |
| 의존성 관리 | 어떤 패키지를 쓸지 명시하고 자동 설치 |
| 자산 등록 | 이미지, 폰트 등 프로젝트 외부 리소스를 사용할 수 있게 함 |
| 환경 제약 | Dart SDK 버전 제약 설정으로 빌드 오류 방지 |

1. pubspec.yaml이 .json 말고 .yaml인 이유  
   : 설정 파일은 사람이 자주 보고 고치기 때문에 YAML이 더 적합

|  |  |
| --- | --- |
| **YAML(.yaml)** | JSON(.json) |
| **가독성 좋음** (사람이 보기 쉬움) | 컴퓨터가 해석하기 쉬움 |
| **주석 지원** (시험 포인트!) | 주석 ❌ |
| **들여쓰기만으로 구조 표현** | 중괄호, 쉼표 필요 |
| **간결하고 명확** | 상대적으로 구조가 복잡함 |
| Docker, GitHub Actions, Firebase 등도 사용 | 대부분의 설정 파일이 YAML로 구성됨 |

1. pubspec.yaml 수정 후 해야 할 일은?  
   : flutter pub get 명령어 실행
2. pubspec.yaml 우측 상단 링크 4개 역할  
   1) Pub get : 외부 라이브러리를 추가한 후 프로젝트에 다운로드하여 적용함  
   2) Pub upgrade : 지정된 버전 중 가장 최신 버전을 검색함  
   3) Pub outdated : 설치된 라이브러리 버전과 최신 버전, 널 안전화 버전, 개발 중인 버전 등의 정보를 표시함  
   4) Flutter doctor : 플러터 환경 설정이 잘 되었는지 검사

**[15. 샘플 앱의 기본구조에서 출제 (예: StatelessWidget, StatefulWidget 의 기본 구조, 각각의 동작성 차이)]**

1. StatelessWidget  
   : 상태(State)가 **변하지 않는 위젯**  
   → 값이 고정되어 있고, UI **불변 => 재사용 가능**- 렌더링 효율성이 좋음  
   - 고정된 UI에서 사용 ex) MyApp  
   - build()만 있음. 상태 못 바꿈  
    \* build() 메서드 : 위젯 생성 시 호출됨. 실제로 화면에 그릴 위젯을 작성해 반환- 프로퍼티로 변수를 가지지 않음.(상수는 가능)
2. StatefulWidget  
   : 상태(State)가 **변할 수 있는 위젯**  
   → 값(상태)이 바뀌면 setState()로 **UI 변경(리빌드)**- 동적인 상태(State)를 관리해야 하는 UI 요소(버튼, 입력창, 애니메이션 등)를 표현할 때 사용 ex) MyHomePage  
   - createState() + 상태 클래스 필요- 상태 관리, 동적인 데이터 처리, UI 재구성
3. MaterialApp 클래스  
   : build() 메서드에서 반환하는 인스턴스로 Flutter 애플리케이션의 기본적인 설정과 구조를 제공하는 데 사용  
   -UI의 기본 구조를 제공  
   -테마 설정 theme:  
   -내비게이션 관리  
   -지역화 지원  
   -디버깅 도구 제공
4. 샘플 앱 구조 요약 (계수기 앱 /lib/main.dart)  
   import 'package:flutter/material.dart'; // Flutter의 기본 UI 구성 요소(Material 디자인)를 사용

void main() {

**runApp(MyApp()); // 앱 실행 시작**

}  
// 시작 클래스. 머티리얼 디자인 앱 생성

**class MyApp** **extends StatelessWidget** {

**@override**

Widget **build**(BuildContext context) {

return **MaterialApp**( // 앱 전체 스타일, 라우팅 관리

home: **MyHomePage(**title : ‘Flutter Demo Home Page’**)**, // 표시할 화면의 인스턴스

);

}

}  
// 시작 클래스가 실제로 표시하는 클래스. 카운터 앱 화면

**class MyHomePage extends StatefulWidget** {  
 // 생성자  
 MyHomePage({Key key, this.title}) : super(key:key);  
 final String title;

**@override**

**State<MyHomePage> createState() => \_MyHomePageState(); // 상태를 갖는 위젯 생성. 1번만 실행됨.**

}  
// 위 MyHomePage 클래스의 상태를 나타내는 State 클래스  
// 상태 클래스는 변경 가능한 상태를 프로퍼티 변수로 표현함.

**class \_MyHomePageState extends State<MyHomePage> {**

int \_counter = 0; // 화면에 표시할 상탯값인 카운터 숫자

void \_incrementCounter() {

**setState(() {**

**\_counter++; // 상태 변경 → 화면 다시 그림, StatefulWidget만 가능**

**});**

}  
// 화면에 UI를 그리는 메서드. 그려질 위젯을 반환

**@override**

Widget **build**(BuildContext context) {

**return Scaffold**( // 머티리얼 디자인 기본 뼈대 위젯

**appBar**: AppBar( // 상단 앱바  
title: Text(widget.title),

),

**body**: Center( // 표시할 내용

child: Column(

children: [

Text('버튼을 누른 횟수:'),

Text('$\_counter'),

],

),

),

floatingActionButton: FloatingActionButton(

onPressed: \_incrementCounter, // 버튼 누르면 \_counter++

child: Icon(Icons.add),

),

);

}

}

1. 위젯에서 위젯으로 값 전달  
   : 생성자를 이용해 전달  
   ex) MaterialApp의 home -> MyHomePage클래스 생성자 -> 필드 상수 String title -> widget.title
2. 텍스트, 스크린샷, 직사각형, 폰트이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.상태 변경  
   : State클래스가 제공하는 setState() 메서드는 build()메서드가 다시 실행되게 함.  
   : \_incrementCounter()에 의해 setState()이 호출됨  
   : setState() 호출된에 따라 Scaffold 영역이 갱신됨.
3. Scaffold 클래스  
   : 머티리얼 디자인 앱을 만들 때 뼈대가 되는 위젯.  
   Scaffold {  
    appBar: AppBar( // 머리티얼 디자인 앱에서 상단의 제목과 메뉴를 표시하는 영역  
    title: Text(widget.title),  
    ),  
    body: ,  
    floatingActionButton: ,  
   };

**[4장 기본 위젯I : 6문항]**

**[16, 17. 화면 배치에 쓰는 위젯 : 2문항]**

1. 위젯 Widget  
   : 화면을 그리는 모든 디자인 요소  
   : UI를 구성하는 기본 단위  
   : 화면에 표시되는 모든 것을 정의함  
   : 버튼, 텍스트, 이미지, 레이아웃 등 모든 구성 요소가 위젯으로 표현됨  
   \* 앱은 위젯의 조합으로 구성되는 것
2. **Container** : 너비, 높이가 있는 영역  
   color, width, height, padding, margin, child 속성  
   ※ child 속성으로 자식 위젯을 담을 수 있음  
   Container(  
    color: Colors.red,  
    width:100, height:100,  
    padding: const EdgeInsets.all(8), // int넣으면 자동 double 형변환  
    margin: const EdgeInsets.all(8.0), // 상하좌우 모두 적용  
   )
3. **Column** : 수직 방향으로 위젯들을 배치, 표시할 위젯 크기만큼의 가로길이를 가짐  
   Column(  
    children: [ ... ],  
    mainAxisAlignment: MainAxisAlignment.start, // 주축, 세로 방향 기준(css와 동일)  
    crossAxisAlignment: CrossAxisAlignment.center, // 교차축, 가로 방향 중앙 정렬  
   )
4. **Row** : 수평 방향으로 위젯들을 배치  
   Row(  
    children: [ ... ],  
    mainAxisSize: MainAxisSize.max, // 가로로 최대 크기, 남은 공간 모두 차지  
    mainAxisAlignment: MainAxisAlignment.spaceBetween, // 가로 방향 양 끝 맞춰서 간격 같게  
    crossAxisAlignment: CrossAxisAlignment.center, // 세로 방향 중앙 정렬  
   )
5. 텍스트, 스크린샷, 직사각형, 폰트이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.**Stack**: 위젯들을 순서대로 겹치게 함  
   Stack(  
    children: [  
    Positioned(...), // 이거 없으면 기본은 왼쪽상단 정렬   
    Widget(),  
    ]  
   )
6. **SingleChildScrollView**: 화면 크기를 넘어가면 스크롤이 생기게 함(특히 column에서)  
   SingleChildScrollView(  
    child: ListBody( // 또는 Column  
    children: [],  
    )  
   )  
   \* 하나의 자식 위젯을 가짐.(SingleChild), ListBody는 스크롤 가능 영역이 가로로 꽉참.  
   (Column은 표시할 위젯의 크기만큼 가로 길이를 가짐)
7. **텍스트, 스크린샷, 스마트폰이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.ListView, ListTile** : 리스트를 표시하는 위젯  
   **ListView**(  
    scrollDirection: Axis.vertical, // 기본 세로 스크롤  
    children: [  
    **// 여기에 ListTile, Container, Text 등의 위젯이 들어감.**  **ListTile(**  
    leading: Icon(Icons.star), // 왼쪽 아이콘  
    title: Text('제목입니다'), // 가운데 제목  
    trailing: Icon(Icons.navigate\_next), // 오른쪽 아이콘 >  
    onTap: () {}, // 탭했을 때 동작  
    )  
    ],  
   )  
   : ListTile 위젯을 이용해 리스트 아이템을 쉽게 작성할 수 있음
8. **GridView** : **열 수를 지정**하여 그리드 형태로 표시하는 위젯  
   **GridView.count**(  
    **crossAxisCount**: 열수,  
    **children**: [  
    Container(color: Colors.red),  
    Container(color: Colors.blue),  
    // ...  
    ],  
   )
9. **PageView** : 여러 페이지를 좌우로 슬라이드하여 넘길 수 있게 하는 위젯  
   **PageView**(  
    children: [  
    Container(color: Colors.red), // 화면 꽉 채움 (컨테이너가 한 페이지를 구성)  
    Container(color: Colors.green),  
    Container(color: Colors.blue),  
    ],  
   )  
   \* Tab과 연동하여 사용하지 않으면 좌우로 슬라이드가 가능한지 사용자가 모를 수 있어 단독으로는 잘 안 씀. PageController와도 같이 씀.
10. 스크린샷, 텍스트, 운영 체제, 소프트웨어이(가) 표시된 사진

    AI가 생성한 콘텐츠는 부정확할 수 있습니다.AppBar, TabBar, Tab, TabBarView AppBar에 TabBar를 배치하고 Tab 및 body에 TabBarView를 배치하여 탭으로 이동하는 화면을 구성할 수 있음  
    class MyHomePage extends StatelessWidget {  
     @override  
     Widget build(BuildContext context) {  
     **return DefaultTabController(** // Scaffold를 감싸고 탭 수 지정  
     length: 3,  
     **child**: **Scaffold(** appBar: **AppBar**(  
     title: Text('Tab'),  
     **bottom: TabBar(** // Scaffold의 bottom 프로퍼티에 TabBar 지정  
     **tabs**: <Widget>[ // tabs 프로퍼티는 Tab의 리스트 지정  
     **Tab**(icon: Icon(Icons.tag\_faces)),  
     Tab(text: '메뉴2'), **//** 탭에는 아이콘, 글자 둘 다 표시 가능  
     Tab(icon: Icon(Icons.info), text: '메뉴3'),   
     ],  
     ),  
     ),  
     **body:** **TabBarView**( // Scaffold의 body 프로퍼티에는 TabBarView 배치  
     **children:** <Widget>[ // children 프로퍼티에 표시할 화면 배치  
     Container(color: Colors.yellow),  
     Container(color: Colors.orange),  
     Container(color: Colors.red),  
     ],),),);}}
11. BottomNavigationBar : 화면 하단에 2~5개의 탭 메뉴를 구성할 수 있게 해주는 위젯  
    : 최소 2개의 탭메뉴 필요!!  
    **Scaffold**(  
     **bottomNavigationBar: BottomNavigationBar**(  
     **items**: [  
     **BottomNavigationBarItem**(  
     **icon**: Icon(Icons.home),  
     **label**: '홈',  
     ),  
     BottomNavigationBarItem(  
     icon: Icon(Icons.settings),  
     label: '설정',  
     ),],),)

**[18, 19. 위치, 정렬, 크기를 위한 위젯 : 2문항]**

1. Center : 화면 중앙으로 정렬시키는 위젯  
   Center(  
    **child**: Text('중앙!'), // child 필수!  
   )
2. Padding : 안쪽 여백을 표현할 때 사용하는 위젯  
   Padding(  
    padding: const EdgeInsets.all(16.0), // or .only(), .fromTRB(왼쪽, 위, 오른쪽, 아래)  
    child: ,  
   )  
   : EdgeInsets.all() 전체 / only() 원하는 방향별 지정 / fromTRB() 네방향을 각각 모두 지정
3. Align : 자식 위젯을 부모 위젯 내에서 정렬 방향을 지정하는 위젯  
   Align(  
    **alignment**: Alignment.bottomRight,  
    **child**: 자식위젯,  
   )
4. **Expanded** : 자식 위젯의 크기를 최대한으로 확장시켜주는 위젯  
   : **Row/Column 내에서** 남은 공간을 채움 (단독 사용 불가!)   
   **Expanded**( **// Row/Column의 Children:[] 내에서 사용!  
    flex:**2, **// Expanded를 여러 위젯에 동시에 적용 시 flex에 정수값을 지정하여 비율 지정. 미지정 시, 기본값 1**  
    **child**: Container(color: Colors.blue),  
   )
5. SizedBox 자식(child) 위젯을 특정 사이즈로 조정하고자 할 때 사용하는 위젯  
   SizedBox(  
    width: 가로길이,  
    height: 세로길이,  
    **child**: 위젯, // child 없으면 빈공간임.  
   )
6. 직사각형, 텍스트, 스크린샷, 디자인이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.Card : 카드 형태의 모양을 제공해주는 위젯  
   Card(  
    **shape: RoundedRectangleBorder**( // 보더 종류  
    **borderRadius**: BorderRadius.circular(16.0), // 모서리 라운드 정도  
    )  
    **elevation**: 실숫값, // 정수 입력 시 자동 변환, 그림자 깊이  
    **child**: 위젯,  
    ),  
   )

**[20, 21. 버튼 계열 위젯 : 2문항]**

1. **텍스트, 폰트, 일렉트릭 블루, 공구이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.ElevatedButton** : 입체감(그림자)을 가지는 일반적인 버튼 위젯  
   ElevatedButton(  
    **onPressed**: () {}, // 버튼이 눌렸을 때 실행될 코드  
    **child**: Text('눌러줘'),  
   )   
   \* onPressed 필수! null이면 버튼 비활성화됨.
2. TextButton : 투명 배경에 평평한 텍스트 버튼 위젯  
   TextButton(  
    **onPressed**: () {},  
    **child**: Text('TextButton'),  
   )
3. 상징이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.IconButton : 아이콘을 표시하는 버튼 위젯 (텍스트 없음!)  
   IconButton(  
    **icon**: Icon(Icons.favorite), // 아이콘 지정  
    color: Colors.red, // 아이콘 색상 지정  
    iconSize: 100.0, // 기본값 24.0  
    **onPressed**: () {},  
   );  
   \* child 프로퍼티가 없음! 자식 위젯 포함 불가!
4. FloatingActionButton : 입체감 있는 둥근 버튼 위젯   
   FloatingActionButton(  
    **onPressed: \_incrementCounter**, // 버튼 눌리면 실행되는 메서드  
    tooltip: ‘Increment’, // 사용자가 버튼을 길게 터치할 때 표시할 글자를 지정  
    **child**: Icon(Icons.add), // 아이콘 인스턴스를 정의하여 버튼 모양 표시  
   ),  
   : 단독 사용 또는 floatingActionButton 프로퍼티에 사용 가능.
5. onPressed 메소드의 다른 형태 (참고)  
   onPressed: () => \_incrementCounter(); // 람다식  
   onPressed: () {  
    return \_incrementCounter(); // 익명함수  
   }

**[5장 기본 위젯II: 5문항]**

**[22. 입력 관련 : 1문항]**

1. TextField : 글자를 입력받는 위젯
2. CheckBox와 Switch : 선택 체크, 체크해제를 지원하는 위젯
3. Radio와 RadioListTile : 선택 그룹 중 하나를 선택할 때 사용하는 위젯
4. DropDownButton : 여러 아이템 중 하나를 고를 수 있는 형태의 위젯

**[23. 다이얼로그 등 위젯 : 1문항]**

1. AlertDialog : 사용자 확인을 요구하거나 메시지를 표시하는 용도
2. DatePicker : 날짜를 선택할 때 사용
3. TimePicker : 시간을 선택할 때 사용하는 위젯

**[24. 이벤트 : 1문항]**

1. GestureDetector/InkWell : “글자나 그림 같이 이벤트 속성이 없는 위젯에서 이벤트를 사용하고자 할 때 사용

**[25. 애니메이션 : 1문항]**

1. Hero : “페이지 전환시 연결되는 애니메이션 지원
2. AnimatedContainer : 한 화면 내에서 변경된 프로퍼티에 의해 애니메이션
3. SliverAppBar와 SliverFillRemaining : 화면 헤더를 동적으로 표현

**[26. 쿠퍼티노 디자인 : 1문항]**

1. Cupertino 디자인  
   : Cupertino는 Apple iOS의 디자인 시스템을 Flutter에서 구현한 위젯 집합

**[6장 내비게이션: 3문항]**

**[내비게이션의 동작성 : 1문항]**

1. Flutter의 내비게이션은 Stack 구조로 동작  
   - **push()** → 새로운 화면을 stack에 쌓음 (앞으로 이동)

- **pop()** → 현재 화면을 stack에서 제거함 (뒤로 이동)  
Navigator.push(context, MaterialPageRoute(builder: (context) => SecondPage()));

Navigator.pop(context);  
: Navigator는 화면 간 이동을 관리하는 내장 내비게이터

: **기본 동작은 Stack (후입선출)**

: pop은 항상 **현재 화면 제거** (이전 화면으로 돌아감)

1. StatefulWidget의 상태 initState, build, dispose  
   -initState() : 위젯이 생성될 때 호출  
   -build() : 위젯이 그려져야 할 때 호출  
   -dispose() : 위젯이 완전히 종료될 때(pop 될 때) 호출

**[push / pop 사용법 : 1문항]**

1. push()  
   Navigator.push(  
    context,  
    MaterialPageRoute(builder: (context) => SecondPage()),  
   );
2. pop()  
   Navigator.pop(context);

**[routes를 이용한 방법 : 1문항 (!주의! 강의시 설명한 내용도 포함됨, LLM을 활용할 것)]**

1. 라우터 방식  
   // main.dart 내부  
   MaterialApp(  
    initialRoute: '/',  
    **routes: {  
    '/': (context) => HomePage(),  
    '/second': (context) => SecondPage(),  
    },**  
   );
2. 이동 방식  
   Navigator**.pushNamed(context, '/second');**  
   Navigator.pop(context);
3. routes 방식이 더 좋은 이유  
   : 라우터를 통해 상태 관리, 화면 구성, URL 구조까지 **통합적으로 관리**할 수 있어  
   **대형 앱**이나 **로그인/세션 관리가 중요한 앱**에서 적합

|  |  |
| --- | --- |
| 구분 | 라우터(routes) 방식의 장점 |
| 구조 | 화면 경로를 문자열 키로 정리함 (유지 보수 편리) |
| 통합성 | 앱 전체의 상태 관리, 접근 제어, 네비게이션 히스토리를 하나로 통합 관리 가능 |
| 예측성 | 초기 페이지, 로그인/로그아웃 상태 등을 라우팅 키로 제어 가능 |
| 단점 | 동적 데이터 전달은 push 방식보다 설정이 복잡함 |

**[8장 플러터2 변경점 : 1문항]**

**[널 안전한 코드 : 1문항 (!주의! flutter 3기준으로 널에 안전한 코드를 LLM을 활용해서 숙지할 것!)]**

**Dart Null Safety 정리 (Flutter 3 기준)**

**1. 널 안전성 개요**

* Dart는 null safety(널 안정성)를 지원하여 null 관련 오류를 컴파일 단계에서 방지함.
* nullable 타입과 non-nullable 타입을 구분함.
* Flutter 3부터 null safety는 기본 적용됨 (Flutter 2에서는 선택 사항).

**2. null safety 문법 핵심 6가지**

| **문법** | **예시** | **의미** | **주의 사항** |
| --- | --- | --- | --- |
| ? | String? s | null 허용 타입 | null 확인 후 사용 필요 |
| ! | s!.length | null 아님 단정 | null이면 런타임 에러 발생 |
| ?? | s ?? '기본값' | null이면 기본값 사용 | 변수 자체에는 값 대입 안 됨 |
| ??= | s ??= '기본값' | null이면 변수에 값 대입 | 변수 값 변경됨 |
| late | late String s; | 나중에 초기화 보장 | 사용 전 값 반드시 대입 필요 |
| ?. | user?.name | null이면 null, 아니면 접근 | null 안전하게 접근 가능 |

**3. 물음표( ? ) 관련 비교 요약**

| **비교 항목** | **의미** | **예시** | **설명** |
| --- | --- | --- | --- |
| ? | null 허용 타입 | String? name | null 가능 변수 선언 |
| ?. | null 안전 접근 | name?.length | null이면 null 반환, 아니면 length 접근 |
| ? 뒤에 ?? | null이면 기본값 사용 | name ?? '익명' | null 대체 사용 (읽기 전용) |
| ??= | null이면 대입 | name ??= '익명' | null일 경우 값 저장 (쓰기 전용) |

**4. non-nullable vs nullable vs late 변수**

| **타입** | **선언 예시** | **null 가능성** | **초기화 여부** | **사용 조건** |
| --- | --- | --- | --- | --- |
| non-nullable | String name | ❌ 불가 | ✅ 선언 시 초기화 필수 | 즉시 사용 가능 |
| nullable | String? nickname | ✅ 허용 | ❌ 초기화 없어도 됨 | 사용 시 null 처리 필요 |
| late | late String name | ❌ 불가 | ❌ 선언만 가능 | 사용 전 반드시 대입해야 함 |

**5. non-nullable 변수 선언과 초기화 분리**

* **같은 함수 내**에서는 선언과 초기화를 나눠도 된다:

String name;

name = '홍길동'; // ✅ OK

print(name);

* **초기화 없이 사용하면 컴파일 에러** 발생:

String name;

print(name); // ❌ 오류

* **다른 함수에서 나중에 초기화할 경우에는 late 사용 필수**:

late String name;

void init() {

name = '홍길동'; // OK

}

**6. 핵심 요약**

* ?: null 허용
* !: null 아님 확신 (주의!)
* ??: null이면 기본값 사용 (읽기)
* ??=: null이면 대입 (쓰기)
* ?.: null이면 접근하지 않고 null 반환
* late: 나중에 반드시 초기화할 변수에 사용

Dart에서 null safety는 코드의 안정성과 예측 가능성을 보장하기 위한 기능이며, 각 연산자의 역할과 적용 조건을 구분해서 사용해야 한다.

연산자  
1) 산술 연산자  
+ , - , \*  
/ : 나누기 (double 반환)  
~/ : 몫 (int 반환)  
% : 나머지 (int 반환)  
2) 증감 연산자  
++ --  
3) 비교 연산자 : 연산 결과가 bool타입  
== 같은지  
!= 다른지  
>, <, >=, <=  
4) 논리 연산자 : 연산 결과가 bool타입  
&& || !

문자열 내에 변수 삽입 : $변수  
${변수 포함한 수식}

1. 열거 타입 enum type  
   : 상수를 정의하는 특수한 형태의 클래스  
   1) 열거 타입을 사용하는 이유  
   : 상태 표현 등을 할 때 간결하게 표현  
   2) 사용법  
   enum Status {login, logout}