



# Flutter 앱 개발 실전 - Dart

---



## [수업 목표]

실전에서 필요한 Dart 지식 이해하기

- 객체 생성 & 비교
- 객체 복사
- 불변 객체(Immutable Object)
- 직렬화(Serialization) & 역직렬화(Deserialization)
- 코드 생성 패키지(Freezed, Json Serializable)
- Dart 키워드

## [목차]

실전 Dart

01. 객체 생성 & 비교

02. 객체 복사

03. 불변 객체

04. JSON & 직렬화

05. 코드 생성기

06. Dart 키워드

최종 코드



모든 토글을 열고 닫는 단축키

Windows :

`ctrl` + `alt` + `t`

Mac :

`cmd` + `alt` + `t`

## 실전 Dart

### ▼ 학습 목표

#### 1. 객체 생성 & 비교

동일한 값을 비교했지만 경우에 따라 결과가 다르게 나오는 이유를 이해해 봅시다.

```
void main() {  
  print('철수' == '철수');           // true  
  print([1] == [1]);                 // false  
  
  print(A(1) == A(1));                // false  
  print(const A(1) == const A(1));  // true  
}  
  
class A {  
  final int value;  
  
  const A(this.value);  
}
```

<https://dartpad.dev/?id=f89097c59d24fe38e9206b63924d41a5>

#### 2. 객체 복사

`a`를 `b`로 복사한 뒤, `a`를 변경 했는데 `b`도 함께 변경되는 이유를 이해해 봅시다.

```
void main() {
  List<int> a = [1];
  List<int> b = a;

  a.add(2);
  print(a); // [1, 2]
  print(b); // [1, 2]
}
```

<https://dartpad.dev/?id=9675bb570ae91f76729b563b984b02be>

### 3. 불변 객체(Immutable Object)

불변 객체를 사용 이유를 이해하고, 사용 방법을 배워봅시다.

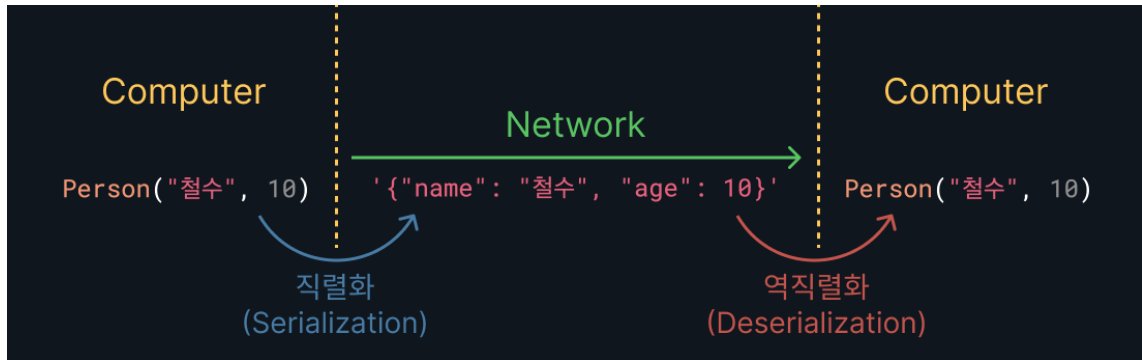
```
void main() {
  List<int> a = const [1];
  List<int> b = a; // 얕은 복사

  // a.add(2); // 불변 객체이므로 수정 불가능(런타임 에러 발생)
  a = List.unmodifiable([...a, 2]); // 불변 객체이므로 새로운 객체 할당만 가능
  print(a == b); // false (메모리 주소 다름)
  print(a); // [1, 2]
  print(b); // [1]
}
```

<https://dartpad.dev/?id=fca0f2d144d74405bfb85b0b128bba49>

### 4. 직렬화(Serialization) & 역직렬화(Deserialization)

커스텀 클래스를 문자열로 변경하고, 문자열을 커스텀 클래스로 변환하는 방법을 배워봅시다.



네트워크 요청 및 응답 이외에도 다양한 상황에 직렬화 & 역직렬화가 활용됩니다.

### 3. 코드 생성 패키지

freezed, json\_serializable가 어떤 코드를 만들어주는지 이해하고 사용해 봅시다.

Before

After

```
@immutable
class Person {
  const Person({
    required this.firstName,
    required this.lastName,
    required this.age,
  });

  factory Person.fromJson(Map<String, Object?> json) {
    return Person(
      firstName: json['firstName'] as String,
      lastName: json['lastName'] as String,
      age: json['age'] as int,
    );
  }

  final String firstName;
  final String lastName;
  final int age;

  Person copyWith({
    String? firstName,
    String? lastName,
    int? age,
  }) {
    return Person(
      firstName: firstName,
      lastName: lastName,
      age: age,
    );
  }

  Map<String, Object?> toJson() {
    return {
      'firstName': firstName,
      'lastName': lastName,
      'age': age,
    };
  }

  @override
  String toString() {
    return 'Person('
      'firstName: $firstName, '
      'lastName: $lastName, '
      'age: $age'
    ')';
  }

  @override
  bool operator ==(Object other) {
    return other is Person &&
      person.runtimeType == runtimeType &&
      person.firstName == firstName &&
      person.lastName == lastName &&
      person.age == age;
  }

  @override
  int get hashCode {
    return Object.hash(
      runtimeType,
      firstName,
      lastName,
      age,
    );
  }
}
```

```
@freezed
class Person with _$Person {
  const factory Person({
    required String firstName,
    required String lastName,
    required int age,
  }) = _Person;

  factory Person.fromJson(Map<String, Object?> json)
    => _$PersonFromJson(json);
}
```

출처 - <https://pub.dev/packages/freezed>

3. `final`, `const`, `getter`, `setter`, `extends`, `mixin`, `extension`

- `final` & `const` 차이점
- `getter` & `setter` 사용방법
- `extends` & `mixin` & `extension` 사용 방법

#### ▼ 패키지 & 확장 프로그램



강의에서 다루는 패키지

- `equatable`
- `build_runner`
- `freezed`
- `json_serializable`



강의에서 다루는 VSCode 확장 프로그램

- `Flutter freezed Helpers`
- `Build Runner`

## 01. 객체 생성 & 비교

#### ▼ 학습 목표



동일한 값을 비교했지만 결과가 각각 다르게 나오는 이유를 이해해 봅시다.

```
void main() {  
  print('철수' == '철수');           // true  
  print([1] == [1]);                 // false  
  
  print(A(1) == A(1));               // false  
  print(const A(1) == const A(1));  // true  
}  
  
class A {  
  final int value;  
  
  const A(this.value);  
}
```

<https://dartpad.dev/?id=f89097c59d24fe38e9206b63924d41a5>

#### ▼ 객체 생성



객체를 생성하는 경우, 값을 메모리에 저장하고 메모리 주소를 반환합니다.

'철수';

철수

← 메모리

0x0000000001 ← 메모리 주소

집을 지은 뒤 집 주소를 알려주는 것과 같습니다.



**값(Value)** : 메모리에 저장된 데이터

**참조(Reference)** : 값이 저장된 메모리 주소



변수는 **참조(메모리 주소)**를 저장합니다.

```
String name = '철수'; // 철수라는 값이 담긴 메모리 주소를 변수(name)에 저장
```

철수

0x0000000001 → name

#### ▼ 객체 비교



객체를 비교하는 두 가지 방법이 있습니다.

```
'철수' == '철수';
```

철수

← 메모리

0x0000000001 ← 메모리 주소

1. **값 비교(Value Equality)** : 메모리에 담긴 **값**을 비교하는 방법
2. **참조 비교(Reference Equality)** : **메모리 주소**를 비교하는 방법





Dart는 기본적으로 **참조 비교(Reference Equality)**를 사용합니다.

```
void main() {  
  print('철수' == '철수'); // true (메모리 주소가 같음)  
  print([1] == [1]);      // false (메모리 주소가 다름)  
}
```

<https://dartpad.dev/?id=b7a956c50d5624c686959046c6a9d018>

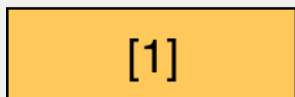
- `'철수' == '철수'` : 두 객체의 메모리 주소가 동일합니다.
- `[1] == [1]` : 두 객체의 메모리 주소가 다릅니다.

#### ▼ 메모리 할당 규칙

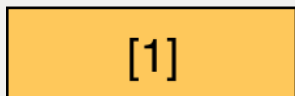


객체 생성시, **가변 객체(Mutable Object)**는 항상 새 메모리를 할당하고 **불변 객체(Immutable Object)**는 값이 동일하다면 기존에 생성한 객체를 재활용합니다.

#### 가변 객체 생성

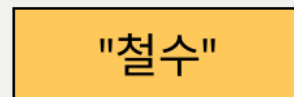


0x0000000002 → a



0x0000000003 → b

#### 불변 객체 생성



0x0000000001 → a  
0x0000000001 → b



위 규칙을 바탕으로 아래 코드를 해석해 봅시다.

```
void main() {  
  print('철수' == '철수'); // true (메모리 주소가 같음)  
  print([1] == [1]);      // false (메모리 주소가 다름)  
}
```

<https://dartpad.dev/?id=b7a956c50d5624c686959046c6a9d018>

- "철수" : **불변 객체**로 메모리에 동일한 값이 있다면 재할당X → 기존 메모리 주소 반환
- [1] : **가변 객체**로 항상 새로운 메모리에 할당 → 새로운 메모리 주소 반환

#### ▼ 가변 객체



**가변 객체(Mutable Object)**란, 메모리에 할당한 뒤 값을 변경할 수 있는 객체입니다.



**가변 객체**로 `List`, `Set`, `Map`, `커스텀 클래스` 등이 있습니다.

```
List a = [];  
a.add(1); // 메모리 주소를 유지하며 값을 변경
```

```
Set a = {};  
a.add(1); // 메모리 주소를 유지하며 값을 변경
```

```
Map<String, dynamic> b = {};  
b['name'] = '철수'; // 메모리 주소를 유지하며 값을 변경
```

```
A a = A(1);  
a.value = 2; // 메모리 주소를 유지하며 값을 변경
```

```
class A {  
    int value;  
    A(this.value);  
}
```



**가변 객체**는 값이 변경될 수 있으므로 값이 동일해도 새롭게 생성 → **항상 다른 메모리 주소 반환**

```
void main() {  
  print([1] == [1]);           // false (메모리 주소가 다름)  
  print({1} == {1});           // false (메모리 주소가 다름)  
  print({"key": 1} == {"key": 1}); // false (메모리 주소가 다름)  
  print(A(1) == A(1));         // false (메모리 주소가 다름)  
}  
  
class A {  
  final int value;  
  const A(this.value);  
}
```

<https://dartpad.dev/?id=898cefab71d9213952ed98e467d549bd>

#### ▼ 불변 객체



**불변 객체(Immutable Object)**는, 메모리에 할당된 뒤 **값을 변경할 수 없는 객체**입니다.



**불변 객체**로 `String`, `int`, `double`, `bool`, `const`로 선언된 객체 등이 있습니다.

```
String a = '철수';
a = '영희'; // '철수'를 변경한게 아니라 '영희'를 생성(메모리 주소가 바뀜)
```

```
int a = 1;
a = 2; // 1을 변경한게 아니라 2를 생성(메모리 주소가 바뀜)
```

```
List a = const [];
// a.add(1); // const로 생성되었기 때문에 수정 불가능
```

```
A a = const A(1); // const 생성자로 호출하여 불변 객체로 생성
// a.value = 2; // const로 생성된 객체는 수정 불가능
```

```
class A {
  final int value; // 모든 속성이 final로 선언되어야 불변 객체
  const A(this.value); // 생성자에 const를 붙여서 불변 객체
}
```



`const`는 컴파일 타임에 고정 값인 객체 앞에만 선언할 수 있습니다.

- **런타임(Runtime)**: 앱을 실행하고 있는 시점
- **컴파일 타임(Compile Time)**: 앱 실행 전 소스 코드를 기계어로 변환하는 시점



**불변 객체**는 값이 변경되지 않으므로 동일한 값의 객체를 여러번 생성할 필요 없음  
→ **값이 동일하면 동일한 메모리 주소 반환**

```
void main() {
  print('철수' == '철수');           // true (기존 값을 재활용하여 메모리 주소가 동일)
  print(1 == 1);                     // true (기존 값을 재활용하여 메모리 주소가 동일)
  print(const [1] == const [1]);     // true (기존 값을 재활용하여 메모리 주소가 동일)
  print(const A(1) == const A(1));   // true (기존 값을 재활용하여 메모리 주소가 동일)
  print(A(1) == A(1));               // false (const로 생성되지 않은 경우, 가변 객체와 같이 생성)
}

class A {
  final int value;
  const A(this.value);
}
```

<https://dartpad.dev/?id=bded105055d457f79b380faf46d76003>

- 모든 가변 객체도 `const` 키워드로 생성되면 불변 객체가 됩니다.
- `const` 생성되지 않은 클래스는 가변 객체와 같이 생성 됩니다.



Flutter에선 값이 변경되지 않는 위젯은 `const` 키워드를 붙여서 **불변 객체로 생성**하는게 좋습니다.

- 값이 동일한 객체들을 **중복 생성하지 않아 메모리 절약**
- 한 번 화면에 그린 뒤 갱신할 필요가 없어 **불필요한 렌더링 최소화**

```
@override
Widget build() {
  return SizedBox(height: 3);
}
```

const 키워드를 붙이라는 안내 문구



메모리 할당 규칙을 알면 다음 코드를 이해할 수 있습니다.

```
void main() {
  print("=====가변 객체(Mutable Object)=====");
  print([1] == [1]); // false (메모리 주소가 다름)
  print({1} == {1}); // false (메모리 주소가 다름)
  print({"key": 1} == {"key": 1}); // false (메모리 주소가 다름)
  print(A(1) == A(1)); // false (메모리 주소가 다름)

  print("=====불변 객체(Immutable Object)=====");
  print(const [1] == const [1]); // true (메모리 주소가 같음)
  print(const {1} == const {1}); // true (메모리 주소가 같음)
  print(const {"key": 1} == const {"key": 1}); // true (메모리 주소가 같음)
  print(const A(1) == const A(1)); // true (메모리 주소가 같음)
  print(1 == 1); // true (메모리 주소가 같음)
  print("1" == "1"); // true (메모리 주소가 같음)
  print(1.1 == 1.1); // true (메모리 주소가 같음)
  print(true == true); // true (메모리 주소가 같음)
}

class A {
  final int value;
  const A(this.value);
}
```

<https://dartpad.dev/?id=ecdeea4b65f1a94201de4101794f8f51>

## ▼ 값 비교



메모리 주소를 비교하는 **참조 비교(Reference Equality)**가 아닌 값을 기준으로 판단하는 **값 비교(Value Equality)**를 구현하는 방법을 실습을 통해 배워봅시다.

## ▼ 프로젝트 생성

1. 앞으로 프로젝트를 진행할 폴더를 생성하겠습니다. 바탕화면에서 `flutter_practice` 라는 폴더를 생성해 주세요.



Flutter 프로젝트는 경로상에 한글이 있으면 정상적으로 작동하지 않습니다.

2. VSCode를 열고 `view` → `command palette` 를 실행한 뒤 `Dart: New Project` 를 선택해 주세요.



단순히 Dart 문법을 실행하기 위해, flutter 프로젝트가 아니라 Dart 프로젝트를 만들었습니다.

```
>dart new
```

**Dart: New Project**

recently used

3. `Console Application` 을 선택해 주세요.

Which Dart template?

**Bare-bones Web App** web

A web app that uses only core Dart libraries.

**Console Application** console

A command-line application.

**Dart Package** package

A package containing shared Dart libraries.

**Server app** server-shelf

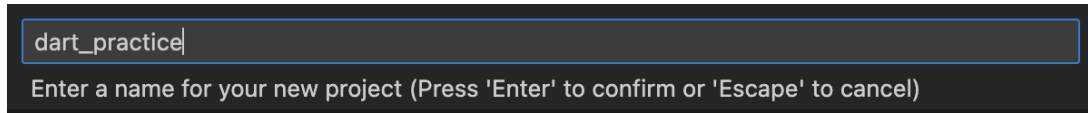
A server app using package:shelf.



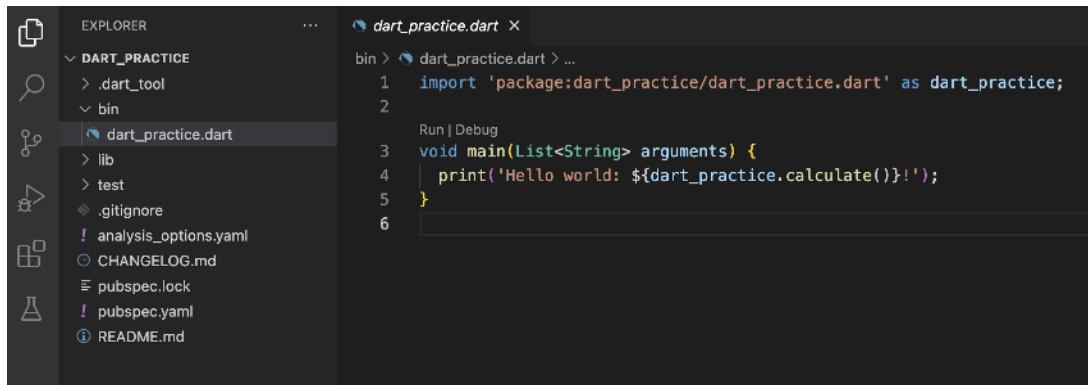
Console Application은 명령어(Command Line)로 작동하는 Dart 프로그램을 의미합니다.



- 위에서 생성한 폴더를 선택한 뒤, 프로젝트 이름을 `dart_practice` 라고 작성하고 엔터를 눌러주세요.



다음과 같이 프로젝트가 생성 됩니다.



Flutter 프로젝트가 아니라 Dart 프로젝트이므로 android, iOS, macOS 등 다른 네이티브 폴더들이 없습니다.



일반적으로 Command Line Application은 `lib` 폴더 밑에 소스 코드를 넣고, `bin` 폴더 밑에 프로그램의 시작 파일을 둡니다.



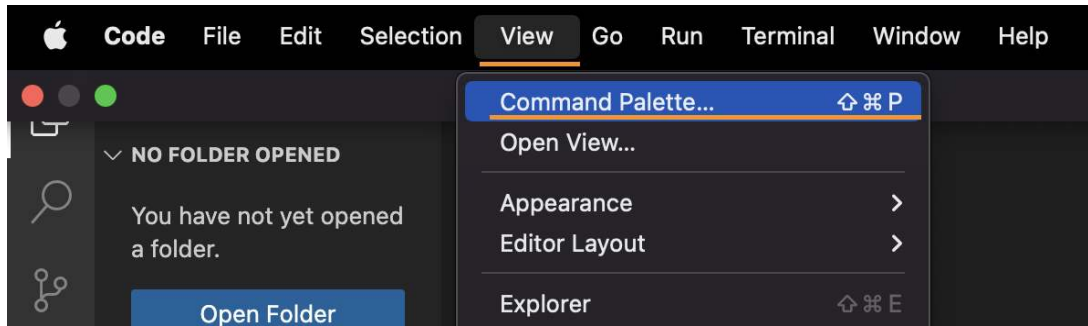
간단한 Dart 실습을 위한 프로젝트이므로 `bin` 폴더에서 진행하겠습니다.

#### ▼ VSCode 설정

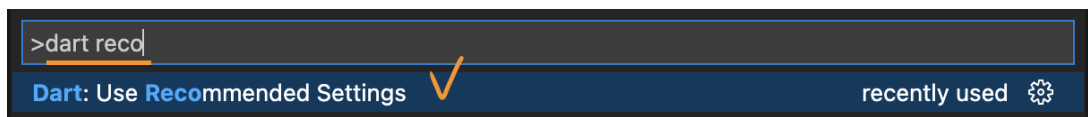


시작하기 앞서 Dart 개발 시 유용한 설정을 적용해 봅시다.

- VSCode를 실행한 뒤 `View` → `Command Palette` 를 선택해 주세요.



2. `dart reco` 라고 입력한 뒤 하단에 추천되는 `Dart: Use Recommended Settings` 를 선택해 주세요. (해당 명령어는 중복 실행해도 괜찮습니다)



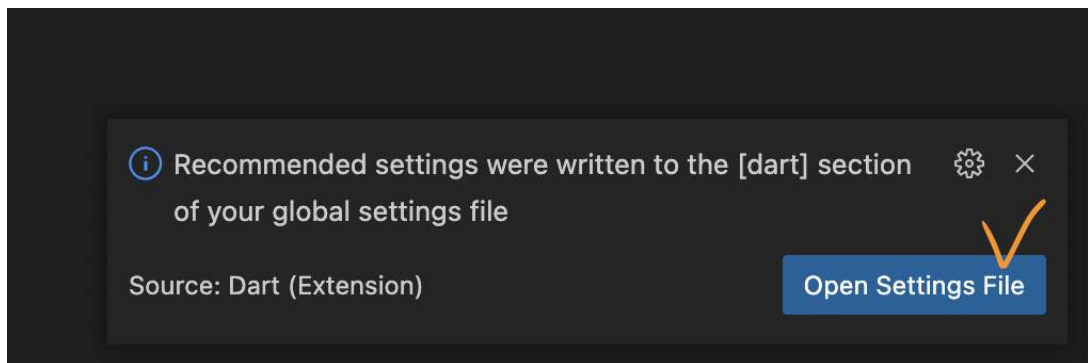
VSCode 에서 Dart 문법 사용시 저장 시 자동 줄 정렬 기능과 같이 기본적으로 사용하면 좋은 편의 기능들이 적용됩니다. 설정에 대한 상세한 내용은 아래 링크를 참고해 주세요.

#### Recommended Settings

There are some settings in VS Code that you may wish to change from the defaults for a better experience editing Flutter code. You can set copy these from the JSON to your VS Code User Settings or by run the Dart: Use Recommended Settings

<https://dartcode.org/docs/recommended-settings/>

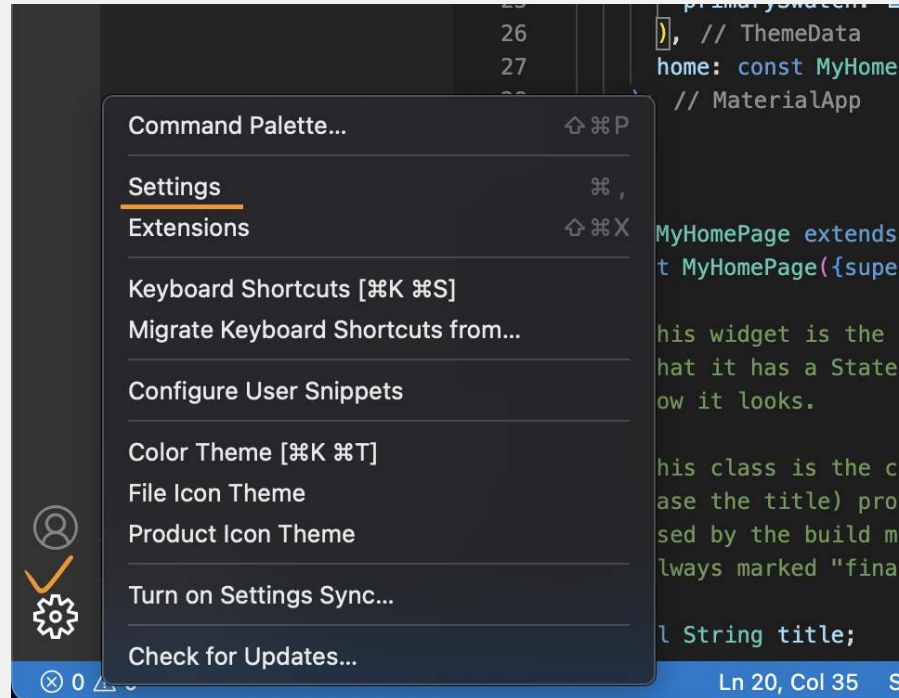
3. 설정이 적용되면 우측 하단에 설정이 잘 적용 되었다고 아래와 같은 배너가 뜹니다. `Open Settings File` 을 클릭하여 설정 파일을 열어주세요.



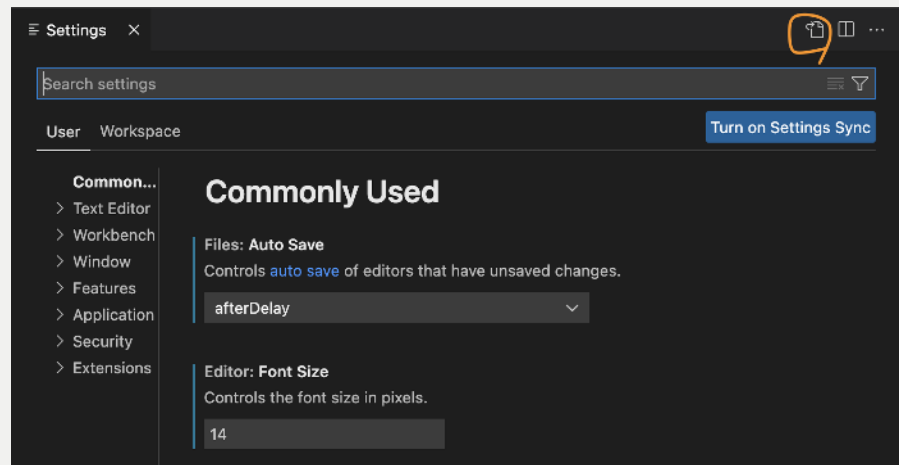


만약 해당 배너가 사라져 버렸다면 다음과 같이 진행하여 설정 파일을 열 수 있습니다.

1. 왼쪽 하단에  → **Settings** 를 클릭해 주세요.



2. 우측 끝에서 세 번째 아이콘을 클릭해 주세요.



4. 설정파일(**setting.json**) 파일이 열리면 다음과 같은 내용이 보입니다.



아래 이미지에 주황색 영역이 **Dart: Use Recommended Settings** 로 추가된 내용입니다. 개인적으로 기타 설정을 하신 경우 일부 내용이 다를 수 있으며 그대로 진행하셔도 무관합니다.

```

{} settings.json ×
Users > devclass > Library > Application Support > Code > User > {} settings.json > ...
1  {}
2  "[dart]": {
3      "editor.formatOnSave": true,
4      "editor.formatOnType": true,
5      "editor.rulers": [
6          80
7      ],
8      "editor.selectionHighlight": false,
9      "editor.suggest.snippetsPreventQuickSuggestions": false,
10     "editor.suggestSelection": "first",
11     "editor.tabCompletion": "onlySnippets",
12     "editor.wordBasedSuggestions": false
13 },
14 }
```

13번째 줄에 심표는 직접 추가해 주세요.

5. **setting.json** 파일에 편의 설정을 몇가지 더 추가하도록 하겠습니다.

아래 코드 스니펫을 복사하여 아래 이미지 기준으로 13번째 **}** 뒤에 붙여 넣어주세요.

▼ 코드스니펫 - **settings.json** : 추가 설정

```

"dart.flutterCreateOrganization": "com.example", /
>window.confirmBeforeClose": "keyboardOnly",    /
>explorer.compactFolders": false,                /
>files.autoSave": "afterDelay",                  /
>editor.codeActionsOnSave": {
    "source.organizeImports": true,                /
    "source.fixAll": true                          /
}
```

```
{ settings.json •
Users > devclass > Library > Application Support > Code > User > {} settings.json > ...
1
2
3 "[dart]": {
4   "editor.formatOnSave": true,
5   "editor.formatOnType": true,
6   "editor.rulers": [
7     80
8   ],
9   "editor.selectionHighlight": false,
10  "editor.suggest.snippetsPreventQuickSuggestions": false,
11  "editor.suggestSelection": "first",
12  "editor.tabCompletion": "onlySnippets",
13  "editor.wordBasedSuggestions": false
14 },
15 "dart.flutterCreateOrganization": "com.example", // Flutter default organization(AOS applicationId / iOS PRODUCT_BUNDLE_IDENTIFIER)
16 "window.confirmBeforeClose": "keyboardOnly", // 단축키로 VSCode 종료시 확인 팝업
17 "explorer.compactFolders": false, // 왼쪽 폴더 경로 전체 표시
18 "files.autoSave": "afterDelay", // 자동 저장
19 "editor.codeActionsOnSave": {
20   "source.organizeImports": true, // 저장시 사용하지 않는 Import 정리
21   "source.fixAll": true // 저장시 fix all (const 같은거 알아서 붙여줌)
22 }
```



위 이미지 기준 14번째 줄에 `dart.flutterCreateOrganization` 는 프로젝트 고유 식별자를 미리 적어놓는 설정입니다.

- 일반적으로 회사 도메인의 역순( `kr.co.devstory` )으로 입력하고, `hello` 라는 프로젝트를 만들면 `kr.co.devstory.hello` 라는 프로젝트 고유 식별자가 생성 됩니다.
- 프로젝트 고유 식별자를 Android에서는 `Application Id` / iOS에서는 `Bundle Identifier` 라고 부르며, **스토어에 출시하기 위해선 다른 앱들과 해당 값이 중복되면 안됩니다.**



`Explorer: Compact Folders` 는 VSCode에서 아래와 같이 폴더를 보는 방식을 다르게 설정 합니다.

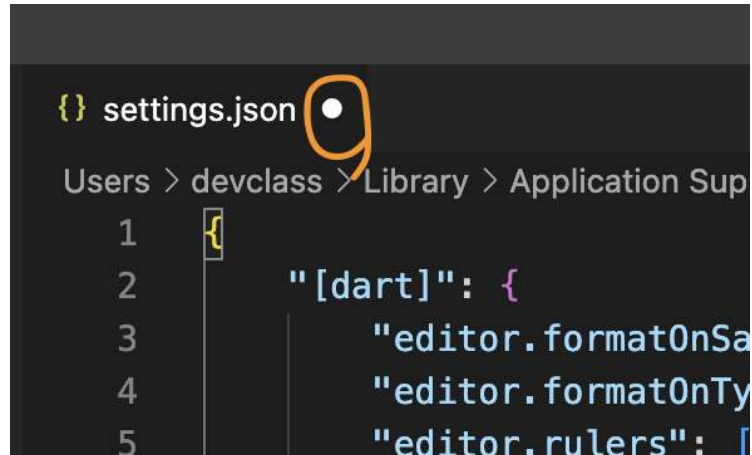
true	false
✓ folder1/folder2/folder3	✓ folder1 ✓ folder2 ✓ folder3

수업 자료는 오른쪽(false) 설정을 따라 작성 되었습니다.

6. 저장(**Ctrl/Cmd + S**)해 주세요.



아래 이미지와 같이 파일 이름 옆에 흰색 동그라미 표시가 있다면, 변경 사항이 있지만 저장이 안된 상태입니다.



#### ▼ 값 비교 구현

1. **bin** 폴더에 **custom\_class** 폴더를 만들고 밑에 **value\_equality.dart** 파일을 생성한 뒤 다음 코드를 작성해 주세요.

▼ 코드스니펫 - **value\_equality.dart** : 시작 코드

```
class A {
  int value;

  A(this.value);
}

void main() {
  print(A(1) == A(1));
  print(A(1));
}
```

```
bin > custom_class > value_equality.dart > ...
1  class A {
2    int value;
3
4    A(this.value);
5  }
6
7  Run | Debug
8  void main() {
9    print(A(1) == A(1));
10   print(A(1));
11 }
```



기존 `dart_practice.dart` 파일은 삭제하였습니다.

2. `main()` 함수 위에 있는 `Run` 버튼을 눌러 `main()` 함수를 실행하면, Debug Console 에서 결과가 출력됩니다.

```
bin > custom_class > value_equality.dart > ...
1  class A {
2      int value;
3
4      A(this.value);
5  }
6
7  Run | Debug
8  void main() {
9      print(A(1) == A(1));
10     print(A(1));
11 }
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

false  
Instance of 'A'  
Exited



Dart 기본적으로 **참조 비교(Reference Equality)**를 사용하기 때문에, 가변 객체인 A 인스턴스는 생성할 때 마다 다른 메모리에 할당되어 `false`를 반환합니다.



A 클래스의 인스턴스를 `print()`에 전달하면, 클래스의 속성이 아닌 어떤 클래스의 인스턴스인지만 출력됩니다.

- 다음 코드를 추가하여 `print()`에 클래스를 전달했을 때, A 클래스의 속성이 출력되도록 만들어 봅시다.

▼ 코드스니펫 - `value_equality.dart` : `toString` override



```
@override
String toString() {
  return "A($value)";
}
```

```
bin > custom_class > value_equality.dart > ...
1  class A {
2    int value;
3
4    A(this.value);
5
6    @override
7    String toString() {
8      return "A($value)";
9    }
10 }
11
12 void main() {
13   print(A(1) == A(1));
14   print(A(1));
15 }
16
```

Run | Debug

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

false  
A(1)  
Exited



Dart의 모든 클래스는 Object 클래스를 부모로 상속받고 있습니다.

`toString()` 메소드의 기본 동작을 `@override` 하면 `print()` 시 출력되는 결과를 변경할 수 있습니다.

#### A tour of the Dart language

This page shows you how to use each major Dart feature, from variables and operators to classes and libraries, with the assumption that

[https://dart.dev/guides/language/language-tour#\\_operators](https://dart.dev/guides/language/language-tour#_operators)



# Dar

4. Object 클래스에서 비교 연산자(==)를 수행하는 방법도 정의하고 있으며, 이를 `@override` 하여 값 비교를 수행하도록 만들 수 있습니다.

▼ 코드스니펫 - `value_equality.dart` : `override == & hashCode`

```
@override
bool operator ==(Object other) {
  return identical(this, other) ||
    other is A && runtimeType == other.runtimeType &
}

@override
int get hashCode => value.hashCode;
```

```
bin > custom_class > value_equality.dart > ...
1  class A {
2      int value;
3
4      A(this.value);
5
6      @override
7      bool operator ==(Object other) {
8          return identical(this, other) ||
9             other is A && runtimeType == other.runtimeType && value == other.value;
10     }
11
12     @override
13     int get hashCode => value.hashCode;
14
15     @override
16     String toString() {
17         return "A($value)";
18     }
19 }
20
21 void main() {
22     print(A(1) == A(1));
23     print(A(1));
24 }
25
```

Run | Debug

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

true  
A(1)  
Exited

위와 같이 `==` 와 `hashCode` 를 정의하면 값 비교를 구현할 수 있습니다.



비교 연산자(`==`)를 다음과 같이 수정하였습니다.

- 메모리 주소가 같으면 `true` 반환(참조 비교)
  - `identical(this, other)`
- 또는(`||`) 아래 조건을 모두 만족하면 `true` 반환(값 비교)
  - `other is A` : 비교 대상이 A 또는 A의 하위 클래스인지 확인
  - `runtimeType == other.runtimeType` : 타입이 같은지 확인
  - `value == other.value` : 속성이 같은지 확인 (int는 불변 객체이므로 참조 비교의 결과와 값 비교의 결과가 동일)



`hashCode` 는 Map, Set 등의 해시 기반 자료형에서 값을 찾을 때 사용하며, 비교 연산자(`==`)를 수정하는 경우 함께 수정해야 합니다.

`hashCode` 를 수정하지 않는 경우, 해시 기반 자료형이 의도와 다르게 동작하게 됩니다.

```
class A {
  int value;

  A(this.value);
  @override
  bool operator ==(Object other) {
    return identical(this, other) ||
      other is A && runtimeType == other.runtimeType && value == other.value;
  }
}

void main() {
  Map<A, int> map = {A(1): 1};
  print(map[A(1)]); // null
}
```

<https://dartpad.dev/?id=6b3880c14746d533f4183c1fbf1f8802>

```
class A {
  int value;

  A(this.value);
  @override
  bool operator ==(Object other) {
    return identical(this, other) ||
      other is A && runtimeType == other.runtimeType && value == other.value;
  }

  @override
  int get hashCode => value.hashCode;
}

void main() {
  Map<A, int> map = {A(1): 1};
  print(map[A(1)]); // 1
}
```

<https://dartpad.dev/?id=1a1ac3f48d20fa7a2b9fd468d5916844>

상세한 이유는 [공식 문서](#)를 참고해 주세요.



`hashCode` 기반의 자료형은 `hashCode` 값이 변경되지 않는다는 전제하에 구현되어 있기 때문에 불변 객체로 만드는게 좋습니다.

```
class A {
  int value;

  A(this.value);
  @override
  bool operator ==(Object other) {
    return identical(this, other) ||
      other is A && runtimeType == other.runtimeType && value == other.value;
  }

  @override
  int get hashCode => value.hashCode; // value를 이용하여 hashCode 구현
}

void main() {
  A a = A(1);
  Map<A, int> map = {a: 1};
  print(map[a]); // 1

  a.value = 2; // value가 변경되면 hashCode도 변경 될
  print(map[a]); // null (hashCode 기반 자료형이 정상 작동하지 않음)
}
```

<https://dartpad.dev/?id=d229cf4bf679dd970797b651df7a737b>

5. A 클래스의 속성을 변경할 수 없도록, 다음과 같이 수정하여 불변 객체로 정의해 줍시다.

▼ 코드스니펫 - `value_equality.dart` : final & const 추가

```
final int value;

const A(this.value);
```

```
1 class A {
2   final int value;
3
4   const A(this.value);
5 }
```



A 클래스의 속성을 변경할 수 없으므로, `hashCode` 를 변경할 수 없습니다.

6. 최종 소스코드는 다음과 같습니다.

▼ 코드스니펫 - `value_equality.dart` : 최종

```
class A {
  final int value;

  const A(this.value);

  @override
  bool operator ==(Object other) {
    return identical(this, other) ||
      other is A && runtimeType == other.runtimeType
  }

  @override
  int get hashCode => value.hashCode;

  @override
  String toString() {
    return "A($value)";
  }
}

void main() {
  print(A(1) == A(1));
  print(A(1));
}
```

```

class A {
  final int value;
  const A(this.value);

  @override
  bool operator ==(Object other) {
    return identical(this, other) ||
        other is A && runtimeType == other.runtimeType && value == other.value;
  }

  @override
  int get hashCode => value.hashCode;

  @override
  String toString() {
    return "A($value)";
  }
}

```

불변 객체로 선언

값 비교(Value Equality)

객체 로깅

Run | Debug

```

void main() {
  print(A(1) == A(1));
  print(A(1));
}

```

→ true

→ A(1)

## ▼ Equatable 패키지



Equatable 패키지를 이용하면 보다 손쉽게 **값 비교(Value Equality)**를 구현할 수 있습니다.

equatable | Dart Package

Not only is it verbose and tedious, but failure to do so can lead to inefficient code which does not behave as we expect. By default, == returns true if

 <https://pub.dev/packages/equatable>



Equatable 클래스를 상속받고, props에 값을 명시하면 됩니다.

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

void main() {
  print(A(1) == A(1)); // true
}

class A extends Equatable {
  final int value;

  const A(this.value);

  @override
  List<Object> get props => [
    value // 값을 비교하려는 속성 명시
  ];
}
```

실행을 진행해 봅시다.

1. Terminal을 열고 다음 명령어를 실행하여 Equatable 패키지를 설치해 주세요.

▼ 코드스니펫 - **터미널** : Dart equatable 설치



```
dart pub add equatable
```



Dart 프로젝트이므로 `dart pub add <패키지명>` 명령어로 패키지를 설치하였습니다. Flutter 프로젝트에선 `flutter pub add <패키지명>` 으로 설치하시면 됩니다.

```
> test
└─ .gitignore
└─ analysis_options.yaml
└─ CHANGELOG.md
└─ pubspec.lock
└─ pubspec.yaml
└─ README.md

5
6 environment:
7   sdk: '>=2.18.4 <3.0.0'
8
9 # dependencies:
10 #   path: ^1.8.0
11
12 dev_dependencies:
13   lints: ^2.0.0
14   test: ^1.16.0
15 dependencies:
16   equatable: ^2.0.5
17
```



`pubspec.yaml` 파일에 위와 같이 equatable 패키지가 추가되었습니다.

2. `custom_class` 폴더에 `equatable.dart` 파일을 만들고, 다음 코드 스니펫을 붙여 넣어주세요.

▼ 코드스니펫 - `equatable.dart` : 시작

```
class A {
  final int value;

  const A(this.value);
}

void main() {
  print(A(1) == A(1));
  print(A(1));
}
```

```

DART_PRACTICE
├── .dart_tool
├── bin
│   ├── custom_class
│   └── equatable.dart
│       └── value_equality.dart
├── lib
├── test
├── .gitignore
├── analysis_options.yaml
├── CHANGELOG.md
├── pubspec.lock
└── pubspec.yaml

bin > custom_class > equatable.dart > ...
1  class A {
2      final int value;
3
4      const A(this.value);
5  }
6
Run | Debug
7  void main() {
8      print(A(1) == A(1));
9      print(A(1));
10 }
11

```



Equatable 패키지 설명에 나온 것과 같이, 앞서 설명한 가변 객체의 `hashCode` 변경 문제를 고려하여 **Equatable 패키지는 불변 객체를 전제로 설계** 되었기 때문에 클래스 A를 불변 객체로 만들었습니다.

We can now compare instances of `Person` just like before without the pain of having to write all of that boilerplate. Note: Equatable is designed to only work with immutable objects so all member variables must be final (This is not just a feature of Equatable - overriding a `hashCode` with a mutable value can break hash-based collections).



Run 버튼을 누르면 참조 비교가 실행되어 `false` 가 반환 됩니다.

```

1  class A {
2      final int value;
3
4      const A(this.value);
5  }
6
7  Run | Debug
8  void main() {
9      print(A(1) == A(1));
10     print(A(1));
11 }

```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```

false
Instance of 'A'
Exited

```

3. `Equatable` 클래스를 상속받도록 A 클래스를 수정해 주세요.

▼ 코드스니펫 - `equatable.dart` : Equatable 상속

```
class A extends Equatable {
```

```

1  import 'package:equatable/equatable.dart';
2
3  class A extends Equatable {
4      int value;
5
6      A(this.value);
7

```



자동 완성을 이용하면 `import 'package:equatable/equatable.dart';` 를 자동으로 추가할 수 있습니다.

4. 빨간줄이 그어져 있는 `class A` 를 클릭한 뒤 상단에 전구 아이콘을 누르고 `Create 1 missing override` 를 선택해 주세요.

```
1 import 'package:equatable/equatable.dart';
2
3 class A extends Equatable {
4
5
6
7
8
9
10
11
```

Quick Fix...

- 🔦 Create 1 missing override
- 🔦 Make class 'A' abstract
- 🔦 Create 'noSuchMethod' method
- 🔦 Ignore 'must\_be\_immutable' for this line
- 🔦 Ignore 'must\_be\_immutable' for this file

다음과 같이 `props` 가 생성됩니다.

```
1 import 'package:equatable/equatable.dart';
2
3 class A extends Equatable {
4   final int value;
5
6   const A(this.value);
7
8   @override
9   // TODO: implement props
10  List<Object?> get props => throw UnimplementedError();
11 }
```



특정 클래스가 `Equatable` 을 상속받으면 `props` 를 필수로 구현해야 합니다.

5. `TODO` 주석을 삭제하고, 다음과 같이 `value` 속성을 이용하여 클래스를 비교하도록 다음과 같이 `props` 를 구현해 주세요.

▼ 코드스니펫 - `equatable.dart` : props 구현

```
@override
List<Object?> get props => [value];
```

```
1  import 'package:equatable/equatable.dart';
2
3  class A extends Equatable {
4      final int value;
5
6      const A(this.value);
7
8      @override
9      List<Object?> get props => [value];
10 }
11
```



`props` 는 비교시 사용하고 싶은 속성을 배열로 반환해 주면 됩니다.

- 다시 `run` 텍스트 버튼을 눌러 실행하면 **값 비교(Value Equality)**가 실행되어 `true`가 반환되고, `print()` 시 속성까지 함께 출력되는 것을 확인할 수 있습니다.

```
1  import 'package:equatable/equatable.dart';
2
3  class A extends Equatable {
4    final int value;
5
6    const A(this.value);
7
8    @override
9    List<Object?> get props => [value];
10 }
11
12 void main() {
13   print(A(1) == A(1));
14   print(A(1));
15 }
16
```

Run | Debug

PROBLEMS   DEBUG CONSOLE   ...   Filter (e.g. text, !exclude)

true  
A(1)  
Exited



Equatable 패키지를 이용하면 **값 비교(Value Equality)**를 손쉽게 구현할 수 있습니다.

7. 최종 소스코드는 다음과 같습니다.

▼ 코드스니펫 - `equatable.dart` : 최종

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

class A extends Equatable {
  final int value;

  const A(this.value);
```

```

    @override
    List<Object?> get props => [value];
  }

void main() {
  print(A(1) == A(1));
  print(A(1));
}

```

```

1  import 'package:equatable/equatable.dart';
2
3  class A extends Equatable {
4    final int value;
5
6    const A(this.value);
7
8    @override
9    List<Object?> get props => [value];
10 }
11
12 Run | Debug
13 void main() {
14   print(A(1) == A(1));
15   print(A(1));
16 }

```

PROBLEMS   DEBUG CONSOLE   ...   Filter (e.g. text, !exclude)

```

true
A(1)
Exited

```

## 02. 객체 복사

### ▼ 학습 목표



아래 코드에서 `a`를 `b`에 복사한 뒤, `a`를 수정했을 때 `b`의 값도 함께 변경됩니다.

```
void main() {  
  List<int> a = [1];  
  List<int> b = a;  
  
  a.add(2);  
  print(a); // [1, 2]  
  print(b); // [1, 2]  
}
```

<https://dartpad.dev/?id=9675bb570ae91f76729b563b984b02be>

위와 같이 동작하는 이유와 이를 방지하는 방법을 배워봅시다.

#### ▼ 얽은 복사





**얕은 복사(Shallow Copy)**란, 참조(메모리 주소)만 전달하는 것을 의미합니다.

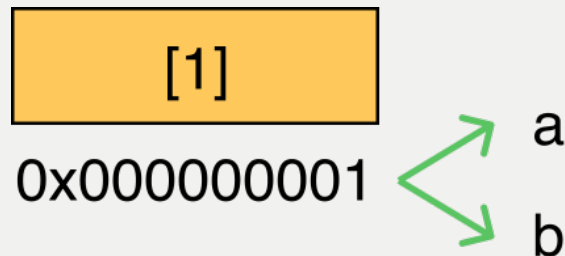
```
void main() {
  List<int> a = [1];
  List<int> b = a; // 얕은 복사 (메모리 주소만 전달)

  print(a == b); // true (a와 b가 동일한 메모리 주소를 가짐)

  a.add(2); // a와 b가 바라보는 메모리의 값을 수정
  print(a); // [1, 2]
  print(b); // [1, 2]
}
```

<https://dartpad.dev/?id=ac80aab09b92662f9351728762245c3f>

참조만 복사하기 때문에, 두 변수가 동일한 메모리를 바라보고 있습니다.



따라서 **a**의 참조에 있는 값을 변경하면 **b**의 참조에 있는 값도 동일하게 변경됩니다.



변수는 **참조(메모리 주소)**를 가지기 때문에, 변수를 다른 변수에 할당하면 **얕은 복사(Shallow Copy)**를 할 수 있습니다.

```
List<int> a = [1];
List<int> b = a; // 얕은 복사
print(a == b); // true (동일한 메모리 주소)
```

#### ▼ 깊은 복사

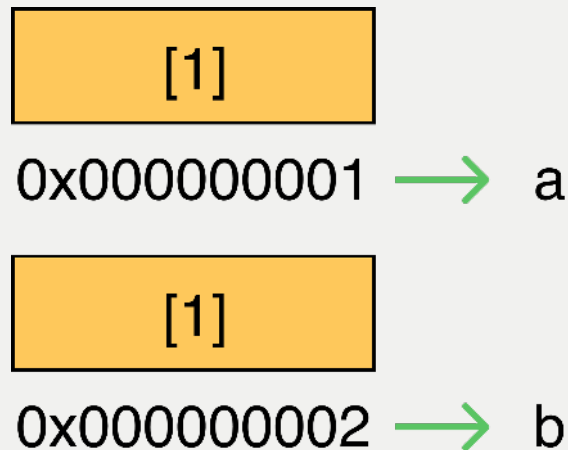


**깊은 복사(Deep Copy)**란, 값이 동일한 객체를 새롭게 생성하는 것을 의미합니다.

```
void main() {  
  List<int> a = [1];  
  List<int> b = a.toList(); // 깊은 복사 (새로운 배열 생성)  
  
  print(a == b); // false (a와 b가 다른 메모리 주소를 가짐)  
  
  a.add(2);  
  print(a); // [1, 2]  
  print(b); // [1]  
}
```

<https://dartpad.dev/?id=14b5a96f67f90aa4e6d9b658642795a0>

객체를 새롭게 생성하기 때문에, 두 변수가 다른 메모리를 바라보고 있습니다.



따라서 **a**의 참조에 있는 값을 변경해도 **b**의 참조에 있는 값이 변경되지 않습니다.




전체 값을 새로운 메모리에 할당하는 **깊은 복사(Deep Copy)**는 데이터 구조에 따라 구현 방법이 다릅니다.

#### ▼ 깊은 복사 - Map



Map 얀 복사를 하면 아래와 같습니다.

DartPad

 <https://dartpad.dev/?id=f1c82cb2e36bc778798806f6d91894c6>




```
1 ▼ void main() {  
2   Map<int, int> a = {1: 1};  
3   Map<int, int> b = a; // 얀 복사  
4  
5   a[1] = 2;  
6   print(a == b); // true (메모리 주소 같음)  
7   print(a);      // {1: 2}  
8   print(b);      // {1: 2}  
9 }  
10
```



Map은 `...` (전개 연산자, Spread Operator)를 이용해 깊은 복사를 할 수 있습니다.

DartPad

 <https://dartpad.dev/?id=d1157a1e9cb84f700682ee402569f65c>

```
1 ▼ void main() {  
2   Map<int, int> a = {1: 1};  
3   Map<int, int> b = {...a}; // 깊은 복사  
4  
5   a[1] = 2;  
6   print(a == b); // false (메모리 주소 다름)  
7   print(a);      // {1: 2}  
8   print(b);      // {1: 1}  
9 }  
10
```

#### ▼ 깊은 복사 - List



배열은 다음과 같은 방법으로 깊은 복사를 할 수 있습니다.

```
List<int> a = [1];

/// 방법1) 배열.toList() 활용
List<int> b = a.toList();

/// 방법2) 전개 연산자(Spread Operator) 활용
List<int> b = [...a];


/// 방법3) 반복문 활용
List<int> b = [for (var i in a) i];

/// 방법4) JSON 직렬화 & 역직렬화 활용
import 'dart:convert';
List<int> b = jsonDecode(jsonEncode(a)).cast<int>();
```



DartPad에서 예제를 실행해 보세요.

DartPad

 <https://dartpad.dev/?id=6d511a9836e229d852dac52f99784d71>

```
import 'dart:convert';

void main() {

  List<int> a = [1];
  List<int> b = a.toList(); // 방법1) 배열.toList() 활용
  // List<int> b = [...a]; // 방법2) 전개구문(Spread Operator) 활용
  // List<int> b = [for (var i in a) i]; // 방법3) 반복문 활용
  // List<int> b = jsonDecode(jsonEncode(a)).cast<int>(); // 방법4) JSON 직렬화와 역직렬화를 이용


  a.add(2); // a가 바라보는 메모리의 값을 수정
  print(a); // [1, 2]
  print(b); // [1]
}
```

## ▼ 깊은 복사 - List 중첩



중첩된 객체를 **깊은 복사(Deep Copy)**하려는 경우, 중첩된 내부 객체를 별도로 복사해야 합니다.

DartPad

 <https://dartpad.dev/?id=071730a7d3efc13c3149292f1955a634>

```
void main() {  
  List<List<int>> a = [[1]];  
  List<List<int>> b = a.toList(); // 첫 번째 배열 신규 생성  
  
  print(a == b);           // false (메모리 주소 다름)  
  a.add([2]);  
  print(a);                 // [[1], [2]]  
  print(b);                 // [[1]]  
  
  print(a[0] == b[0]);     // true (메모리 주소 동일)  
  a[0].add(2);  
  print(a);                 // [[1, 2], [2]]  
  print(b);                 // [[1, 2]]  
}
```

첫 번째 배열은 `toList()` 로 배열을 새로 생성 하였기 때문에 메모리 주소가 다르지만, 내부 배열은 메모리 주소가 동일하여 함께 변경 됩니다. 전체 객체를 깊은 복사하려면 내부 배열도 새로 생성해야 합니다.



2차원 배열(배열속 배열)은 다음과 같은 방법으로 깊은 복사를 할 수 있습니다.

```
List<List<int>> a = [[1]];
```

```
/// map() & toList() & Spread Operator
```

```
List<List<int>> b = a.map((i) => i.toList()).toList();
```

```
List<List<int>> b = a.map((i) => [...i]).toList();
```

```
List<List<int>> b = [...a.map((i) => i.toList())];
```

```
List<List<int>> b = [...a.map((i) => [...i])];
```