




`map()` 은 배열을 순환하며 값을 변경할 수 있는 함수입니다.

```
List<int> a = [1, 2, 3];  
List<String> b = a.map((i) => "$i명").toList();  
print(b); // [1명, 2명, 3명]
```

▼ `map()` 은 **Iterable**을 반환하며 `toList()` 를 통해 **List**로 변경할 수 있습니다.

DartPad

 <https://dartpad.dev/?id=a22ea2fe31a856e4cba882aae4b8bd30>

```
void main() {  
  List<int> a = [1, 2, 3];  
  
  Iterable<String> b = a.map((i) {  
    print("Map $i");  
    return "$i명";  
  });  
  
  List<String> c = b.toList();  
  print(c); // [1명, 2명, 3명]  
}
```

Console


```
Map 1  
Map 2  
Map 3  
[1명, 2명, 3명]
```

`map()` 에 대한 보다 자세한 내용은 **공식 문서**를 참고해 주세요.

▼ List & Iterable

- **List** : **Iterable**의 하위 클래스로 배열의 모든 원소를 메모리에 올려두고 사용합니다.
- **Iterable** : 순차적으로 접근 가능한 요소의 모음으로, 접근하는 요소만 메모리에 올립니다.

DartPad

 <https://dartpad.dev/?id=c76917fda3f6ef3bac019dfbbab598e3>

```
import 'dart:math';

void main() {
  /// 2의 1000승
  int n = pow(2, 1000).toInt();

  /// List
  try {
    List<String> a = List.generate(n, (i) => "$i");
    print(a.length);
  } catch (e) {
    print("실행 실패 : $e"); // 실행 실패 : Invalid argument(s): Invalid array length
  }

  /// Iterable
  Iterable<String> b = Iterable.generate(n, (i) => "$i");
  print(b.length); // 1.0715086071862673e+301
  Iterator iterator = b.iterator;
  if(iterator.moveNext()) {
    print(iterator.current); // 0
  }
  if(iterator.moveNext()) {
    print(iterator.current); // 1
  }
}
```


2의 1000승 크기의 배열은 너무 커서 메모리에 한 번에 생성할 수 없습니다.

List는 메모리 부족으로 실행할 수 없지만, **Iterable**은 에러가 나지 않습니다. Iterable에 대한 보다 상세한 설명은 [블로그](#)를 참고해 주세요.



DartPad에서 예제를 실행해 보세요.

DartPad

 <https://dartpad.dev/?id=947e50bd57442d7d71a36ee5fd012916>


```
void main() {  
  List<List<int>> a = [[1]];  
  List<List<int>> b = a.map((i) => i.toList()).toList(); // 방법1  
  // List<List<int>> b = a.map((i) => [...i]).toList(); // 방법2  
  // List<List<int>> b = [...a.map((i) => i.toList())]; // 방법3  
  // List<List<int>> b = [...a.map((i) => [...i])]; // 방법4  
  
  a.add([2]);  
  print(a); // [[1], [2]]  
  print(b); // [[1]]  
  
  a[0].add(2);  
  print(a); // [[1, 2], [2]]  
  print(b); // [[1]]  
}
```

▼ 깊은 복사 - Custom Class



커스텀 클래스를 얹은 복사하면 다음과 같이 동작합니다.

DartPad

 <https://dartpad.dev/?id=5c0bdb095fb7ef07a5cb26831dda7187>




```
void main() {  
  A a = A(1, 2, 3);  
  A b = a; // 얹은 복사  
  
  print(a == b); // true (메모리 주소가 같음)  
  
  a.value1 = 4;  
  print(a.value1); // 4  
  print(b.value1); // 4  
}  
  
class A {  
  int value1;  
  int value2;  
  int value3;  
  
  A(  
    this.value1,  
    this.value2,  
    this.value3,  
  );  
}
```



커스텀 클래스는 신규 인스턴스를 생성하여 깊은 복사를 할 수 있습니다.

DartPad

 <https://dartpad.dev/?id=2e076b1bdff2ab6b48aeae0abb6c85eb>




```
void main() {  
  A a = A(1, 2, 3);  
  A b = A(a.value1, a.value2, a.value3); // 깊은 복사  
  
  print(a == b); // false (메모리 주소가 다름)  
  
  a.value1 = 4;  
  print(a.value1); // 4  
  print(b.value1); // 1  
}  
  
class A {  
  int value1;  
  int value2;  
  int value3;  
  
  A(  
    this.value1,  
    this.value2,  
    this.value3,  
  );  
}
```



신규 인스턴스를 만들 때 기존 클래스의 속성을 생성자에 매번 전달하는 것은 상당히 번거롭기 때문에 일반적으로 `copyWith()` 메소드를 만들어 사용합니다.

DartPad

 <https://dartpad.dev/?id=bb18576b92e27bf3072ad61953a66776>

```
void main() {  
  A a = A(1, 2, 3);  
  A b = a.copyWith(); // 깊은 복사  
  
  print(a == b); // false (메모리 주소가 다름)  
  
  a.value1 = 4;  
  print(a.value1); // 4  
  print(b.value1); // 1  
}  
  
class A {  
  int value1;  
  int value2;  
  int value3;  
  
  A(  
    this.value1,  
    this.value2,  
    this.value3,  
  );  
  
  A copyWith({  
    int? value1,  
    int? value2,  
    int? value3,  
  }) {  
    return A(  
      value1 ?? this.value1, // value1이 null이면 현재 인스턴스의 value1 전달  
      value2 ?? this.value2, // value2가 null이면 현재 인스턴스의 value2 전달  
      value3 ?? this.value3, // value3가 null이면 현재 인스턴스의 value3 전달  
    );  
  }  
}
```

`copyWith()` 는 현재 인스턴스의 속성들을 복사하여 새로운 인스턴스를 반환하는 메소드입니다. 만약 특정 값만 다르게 복사하고 싶은 경우, 해당 값을 별도로 이름 지정 개변수로 전달하면 됩니다.


```
A a = A(1, 2, 3);  
A b = a.copyWith(value1: 5); // a를 복사하되 value1만 5로
```

▼ 깊은 복사 - Custom Class 중첩



Custom Class에 다른 Class가 중첩된 경우, 중첩된 내부 객체를 별도로 복사해야 합니다. 그렇지 않으면 다음과 같이 내부 객체는 얕은 복사가 됩니다.

DartPad


 <https://dartpad.dev/?id=ebbf5fb88654352d11aebb9cdabb48ea>

```
1▼ void main() {
2    A a1 = A(1, B(1));
3    A a2 = a1.copyWith();
4
5    print(a1 == a2); // false (메모리 주소가 다름)
6    a1.value = 2;
7    print(a1.value); // 2
8    print(a2.value); // 1
9
10   print(a1.b == a2.b); // true (메모리 주소가 같음)
11   a1.b.value = 2;
12   print(a1.b.value); // 2
13   print(a2.b.value); // 2
14 }
15
16▼ class A {
17   int value;
18   B b;
19
20   A(this.value, this.b);
21
22▼ A copyWith({int? value, B? b}) {
23   return A(
24     value ?? this.value,
25     b ?? this.b,
26   );
27 }
28 }
29
30▼ class B {
31   int value;
32
33   B(this.value);
34
35▼ B copyWith({int? value}) {
36   return B(value ?? this.value);
37 }
38 }
```




A 클래스를 `copyWith()` 할 때, 내부 B 클래스도 `copyWith()` 를 이용해 깊은 복사를 하면 됩니다.

DartPad

 <https://dartpad.dev/?id=18ce3c45c95a1ab2feb0262fc718f815>


```
1 void main() {
2   A a1 = A(1, B(1));
3   A a2 = a1.copyWith();
4
5   print(a1 == a2); // false (메모리 주소가 다름)
6   a1.value = 2;
7   print(a1.value); // 2
8   print(a2.value); // 1
9
10  print(a1.b == a2.b); // false (메모리 주소가 다름)
11  a1.b.value = 2;
12  print(a1.b.value); // 2
13  print(a2.b.value); // 1
14 }
15
16 class A {
17   int value;
18   B b;
19
20   A(this.value, this.b);
21
22   A copyWith({int? value, B? b}) {
23     return A(
24       value ?? this.value,
25       (b ?? this.b).copyWith(), // b 깊은 복사
26     );
27   }
28 }
29
30 class B {
31   int value;
32
33   B(this.value);
34
35   B copyWith({int? value}) {
36     return B(value ?? this.value);
37   }
38 }
39
```

▼ 깊은 복사 - List & Custom Class



배열에 커스텀 클래스가 있는 경우도 중첩된 내부 객체를 별도로 복사해야 전체를 깊은 복사를 할 수 있습니다.

DartPad

 <https://dartpad.dev/?id=96fb669b030d16a5020b79edd615138b>

```
void main() {
  List<A> a = [A(1, 2), A(3, 4)];
  List<A> b = a.toList(); // 배열만 깊은 복사, 커스텀 클래스는 얇은 복사

  print(a == b); // false (메모리 주소가 다름)
  a.add(A(5, 6));
  print(a.length); // 3
  print(b.length); // 2

  print(a[0] == b[0]); // true (메모리 주소가 같음)
  a[0].value1 = 10;
  print(a[0].value1); // 10
  print(b[0].value1); // 10
}

class A {
  int value1;
  int value2;


  A(
    this.value1,
    this.value2,
  );

  A copyWith({
    int? value1,
    int? value2,
  }) {
    return A(
      value1 ?? this.value1,
      value2 ?? this.value2,
    );
  }
}
```



`map()` 과 `copyWith()` 를 이용해 커스텀 클래스를 새롭게 생성하면 깊은 복사를 구현할 수 있습니다.

DartPad

 <https://dartpad.dev/?id=562bd3dd5aae8f0bf228f4113634efad>

```
void main() {
  List<A> a = [A(1, 2), A(3, 4)];
  List<A> b = a.map((i) => i.copyWith()).toList(); // 모두 깊은 복사

  print(a == b); // false (메모리 주소가 다름)
  a.add(A(5, 6));
  print(a.length); // 3
  print(b.length); // 2

  print(a[0] == b[0]); // false (메모리 주소가 다름)
  a[0].value1 = 10;
  print(a[0].value1); // 10
  print(b[0].value1); // 1
}

class A {
  int value1;
  int value2;

  A(
    this.value1,
    this.value2,
  );

  A copyWith({
    int? value1,
    int? value2,
  }) {
    return A(
      value1 ?? this.value1,
      value2 ?? this.value2,
    );
  }
}
```

▼ 요약



얕은 복사(Shallow Copy)를 수행하면 메모리를 절약할 수 있지만, 데이터가 의도치 않게 변경될 수 있습니다.

```
1 ▼ void main() {  
2     List<int> a = [1];  
3     List<int> b = a; // 얕은 복사  
4  
5     a.add(2);  
6     print(a); // [1, 2]  
7     print(b); // [1, 2]  
8 }  
9
```

<https://dartpad.dev/?id=7d569d77e9d286d610ebcc14054561d3>

깊은 복사(Deep Copy)를 수행하면 데이터의 의도치 않은 변경을 막을 수 있지만, 메모리 사용량이 증가합니다.

```
1 ▼ void main() {  
2     List<int> a = [1];  
3     List<int> b = a.toList(); // 깊은 복사  
4  
5     a.add(2);  
6     print(a); // [1, 2]  
7     print(b); // [1]  
8 }  
9
```

<https://dartpad.dev/?id=32ca4849f0b3f73be4c6feea9c119efd>

얕은 복사와 **깊은 복사**를 다음과 같이 상황에 맞게 사용하는게 좋습니다.

- 객체 수정이 발생하지 않는다면, **얕은 복사**를 사용하여 **메모리를 절약**합니다.
- 객체 수정이 발생한다면, **깊은 복사**를 이용하여 **의도치 않은 변경을 방지**합니다.

03. 불변 객체

▼ 불변 객체 사용 이유



불변 객체(Immutable Object)를 활용하면,

1. 개발자의 지식 수준과 관련 없이 상황에 적절한 코드를 작성하도록 강제하고, **얕은 복사**와 **깊은 복사**의 장점을 모두 누릴 수 있습니다.

```
1 void main() {  
2   List<int> a = const [1];  
3   List<int> b = a; // 얕은 복사  
4  
5   // a.add(2);      // 불변 객체이므로 수정 불가능(에러 발생)  
6   a = [...a, 2];    // 불변 객체이므로 새로운 객체 할당만 가능  
7   print(a == b);    // false (메모리 주소 다름)  
8   print(a);         // [1, 2]  
9   print(b);         // [1]  
10 }  
11
```

<https://dartpad.dev/?id=4d35fb52fb86d3e2b3af84995ed449e3>

- 객체 수정이 필요한 시점에, **깊은 복사**를 사용하도록 강제합니다.

Uncaught Error: Unsupported operation: add

5번째 라인의 `a.add(2);`를 사용하면 의도치 않은 데이터 변경이 발생할 수 있으나 불변 객체이므로 5번째 라인 실행시 에러가 발생합니다.

- 의도치 않은 데이터 변경을 신경쓰지 않고, **얕은 복사**를 통해 메모리를 절약할 수 있습니다.
2. **값 비교(Value Equality)** 구현시 `hashCode` 변경을 방지할 수 있습니다.

```

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 됨
  print(map[a]); // null (hashCode 기반 자료형이 정상 작동하지 않음)
}

```

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

`hashCode` 가 변경되면 Map과 같은 `hashCode` 기반 자료형이 의도와 다르게 동작할 수 있습니다.

▼ 불변 객체 특징



불변 객체(Immutable Object) 특징

- 동일한 메모리에서 값 수정 불가능
 - 데이터의 예기치 않은 변경으로부터 안전
 - 얇은 복사 활용 가능(메모리 절약)
 - 멀티 스레드 프로그래밍에 유용
- 수정 대신 새 인스턴스 생성
 - 수정 코드 실행시 에러 발생
 - `const` 로 생성된 객체들은 값이 동일한 경우 얇은 복사 활용(메모리 절약)

/// 4개 객체 생성 후 개별 참조

```
List a1 = [];
List a2 = [];
List a3 = [];
List a4 = [];
```

/// 1개 객체 생성 후 메모리 참조

```
const List b1 = [];
const List b2 = [];
const List b3 = [];
const List b4 = [];
```

- 가변 객체에 비해 코드가 길고, 연산량이 많음

```
/// 가변 객체
List<int> a = [1];
a.add(2);           // 추가
a[0] = 0;           // 수정
a.remove(0);        // 삭제

/// 불변 객체
List<int> b = const [1];
b = [...b, 2];       // 추가
b = b.map((i) => i == 1 ? 0 : i).toList(); // 수정
b = b.where((i) => i != 0).toList();       // 삭제
```

<https://dartpad.dev/?id=586f939aa79054bc9ad754aba56f9467>



상황에 따라 가변 객체를 사용하는게 나은 경우도 있으며, 이 때문에 많은 프로그래밍 언어에서 불변이나 가변 중 하나를 선택할 수 있도록 구현되어 있습니다.

▼ 불변 객체 구현




불변 객체를 생성하고 다루는 방법을 배워 봅시다.

▼ 불변 객체 - 배열



불변 배열에 값 추가, 수정 및 삭제하는 방법을 배워봅시다. DartPad 링크를 열어주세요.

DartPad

 <https://dartpad.dev/?id=5eeeceda2fdbe9241cb48fb1713187be>

```
1 ▼ void main() {  
2     /// 추가  
3     /// spread operator  
4     List<int> a1 = const [1];  
5     a1 = [...a1, 2];  
6     print(a1); // [1, 2]  
7  
8  
9  
10    /// 수정  
11    /// map  
12    List<int> b1 = const [1];  
13    b1 = b1.map((v) => v == 1 ? 2 : v).toList();  
14    print(b1); // [2]  
15  
16  
17  
18    /// 삭제  
19    /// where  
20    List<int> c1 = const [1];  
21    c1 = c1.where((v) => v != 1).toList();  
22    print(c1); // [];  
23 }  
24
```



`const` 는 컴파일 타임에 고정 값인 객체 앞에만 선언할 수 있기 때문에, 런타임에 생성된 객체 앞에는 붙일 수 없습니다.

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


따라서 아래 이미지에 표시한 배열들은 앞에 `const` 키워드를 붙일 수 없어 수정 가능한 상태입니다.

```
1 ▼ void main() {  
2     /// 추가  
3     /// spread operator  
4     List<int> a1 = const [1];  
5     a1 = [...a1, 2];  
6     print(a1); // [1, 2]  
7  
8  
9  
10    /// 수정  
11    /// map  
12    List<int> b1 = const [1];  
13    b1 = b1.map((v) => v == 1 ? 2 : v).toList();  
14    print(b1); // [2]  
15  
16  
17  
18    /// 삭제  
19    /// where  
20    List<int> c1 = const [1];  
21    c1 = c1.where((v) => v != 1).toList();  
22    print(c1); // [];  
23 }  
24
```



`List.unmodifiable()` 을 이용하면 런타임 배열의 수정을 막을 수 있습니다.

DartPad

 <https://dartpad.dev/?id=52883d0d16b3564c0a782a237b72b733>

```
1 ▼ void main() {  
2   /// 추가  
3   /// spread operator  
4   List<int> a1 = const [1];  
5   a1 = List.unmodifiable([...a1, 2]);  
6   print(a1); // [1, 2]  
7  
8  
9  
10  /// 수정  
11  /// map  
12  List<int> b1 = const [1];  
13  b1 = List.unmodifiable(b1.map((v) => v == 1 ? 2 : v));  
14  print(b1); // [2]  
15  
16  
17  
18  /// 삭제  
19  /// where  
20  List<int> c1 = const [1];  
21  c1 = List.unmodifiable(c1.where((v) => v != 1));  
22  print(c1); // [];  
23 }  
24
```

불변 객체를 다룰 때 `List.unmodifiable()` 까지 구현할지 여부는 선호에 따라 선택하면 됩니다.



참고로 `const` 를 붙인 경우에만, 값이 같은 경우 동일한 메모리 주소를 반환합니다.

```
void main() {
  print(const [1] == const [1]); // true (메모리 주소 같음)
  print(List.unmodifiable([1]) == List.unmodifiable([1])); // false (메모리 주소 다름)
  print(Map.unmodifiable({1: 1}) == Map.unmodifiable({1: 1})); // false (메모리 주소 다름)
}
```

<https://dartpad.dev/?id=d23238c8b31b4a897b5bd75b0c45c3a5>

▼ 불변 객체 - 커스텀 클래스



VSCode에서 커스텀 클래스를 불변 객체로 생성하는 방법을 배워봅시다.

1. VSCode에 `custom_class` 폴더에 `immutable.dart` 파일을 만들어 주세요.

▼ 코드스니펫 - `immutable.dart` : 시작 코드

```
class A {
  int value1;
  int value2;

  A({
    required this.value1,
    required this.value2,
  });

  @override
  String toString() {
    return "A(value1:$value1, value2:$value2)";
  }
}

void main() {
  A a = A(value1: 1, value2: 1);
  A b = a; // 얇은 복사
```

```

    a.value1 = 2;
    print(a);
    print(b);
  }

```

```

bin > custom_class > immutable.dart > ...
1  class A {
2    int value1;
3    int value2;
4
5    A({
6      required this.value1,
7      required this.value2,
8    });
9
10   @override
11   String toString() {
12     return "A(value1:$value1, value2:$value2)";
13   }
14 }
15
16 void main() {
17   A a = A(value1: 1, value2: 1);
18   A b = a; // 얀 복사
19
20   a.value1 = 2;
21   print(a);
22   print(b);
23 }
24
Run | Debug
A(value1:2, value2:1)
A(value1:2, value2:1)
Exited

```



얀은 복사를 한 뒤 객체의 속성을 수정했기 때문에 실행시 `a` 와 `b` 의 `value1` 이 동일하게 변경됨을 확인할 수 있습니다.



불변 객체로 만들어 의도치 않은 데이터 변경을 막아봅시다.

2. A 클래스의 모든 속성에 `final` 을 붙이고, 생성자 앞에 `const` 키워드를 붙여 불변 객체로 만들어 주세요. 그리고 `main()` 함수에서 객체 수정 대신, 새로운 인스턴스를 생성하도록 변경해 주세요.

▼ 코드스니펫 - `immutable.dart` : 불변 객체

```
class A {
  final int value1;
  final int value2;

  const A({
    required this.value1,
    required this.value2,
  });

  @override
  String toString() {
    return "A(value1:$value1, value2:$value2)";
  }
}

void main() {
  A a = A(value1: 1, value2: 1);
  A b = a; // 얕은 복사

  // a.value1 = 2; // 에러 발생
  a = A(value1: 2, value2: a.value2); // 깊은 복사
  print(a);
  print(b);
}
```

```

1  class A {
2+  | final int value1;
3+  | final int value2;
4
5+  | const A({
6    |   required this.value1,
7    |   required this.value2,
8    | });
9
10  | @override
11  | String toString() {
12  |   | return "A(value1:$value1, value2:$value2)";
13  |   }
14  | }
15
16  void main() {
17  |   A a = A(value1: 1, value2: 1);
18  |   A b = a; // 얕은 복사
19
20+  | // a.value1 = 2; // 에러 발생
21+  | a = A(value1: 2, value2: a.value2); // 깊은 복사
22  | print(a);
23  | print(b);
24  | }
25

```



A 클래스의 모든 속성은 `final` 로 선언 되었기 때문에 속성에 새 값을 재할당 할 수 없고, 값이 변경된 인스턴스를 새롭게 생성해야 합니다.



Run 버튼을 눌러 실행하면, `a`와 `b`가 다른 참조를 가지고 있기 때문에 값이 다를 수 있습니다.

```

Run | Debug
16 void main() {
17   A a = A(value1: 1, value2: 1);
18   A b = a; // 얕은 복사
19
20   // a.value1 = 2; // 에러 발생
21   a = A(value1: 2, value2: a.value2); // 깊은 복사
22   print(a);
23   print(b);
24 }
25
PROBLEMS OUTPUT DEBUG CONSOLE ... Filter (e.g. text, !exclu...
A(value1:2, value2:1)
A(value1:1, value2:1)
Exited

```



불변 객체에는 값을 수정하고 싶을 때, 값이 변경된 객체를 생성하는데 `copyWith()` 함수를 만들어 사용하면 더욱 편리합니다.

3. A 클래스에 `copyWith()` 메소드를 추가해 주세요.

▼ 코드스니펫 - `immutable.dart` : `copyWith()` 추가

```

class A {
  final int value1;
  final int value2;

  const A({
    required this.value1,
    required this.value2,
  });

  @override
  String toString() {
    return "A(value1:$value1, value2:$value2)";
  }
}

```

```

    }

    A copyWith({
      int? value1,
      int? value2,
    }) {
      return A(
        value1: value1 ?? this.value1,
        value2: value2 ?? this.value2,
      );
    }
  }

void main() {
  A a = A(value1: 1, value2: 1);
  A b = a; // 얕은 복사

  // a.value1 = 2; // 런타임 에러 발생
  // a = A(value1: 2, value2: a.value2); // 깊은 복사
  a = a.copyWith(value1: 2); // 깊은 복사
  print(a);
  print(b);
}

```

```

10  @override
11  String toString() {
12      return "A(value1:$value1, value2:$value2)";
13  }
14+
15+  A copyWith({
16+      int? value1,
17+      int? value2,
18+  }) {
19+      return A(
20+          value1: value1 ?? this.value1,
21+          value2: value2 ?? this.value2,
22+      );
23+  }
24  }
25
26  void main() {
27      A a = A(value1: 1, value2: 1);
28      A b = a; // 얕은 복사
29
30      // a.value1 = 2; // 에러 발생
31+     // a = A(value1: 2, value2: a.value2); // 깊은 복사
32+     a = a.copyWith(value1: 2); // 깊은 복사
33     print(a);
34     print(b);
35 }
36

```



`copyWith()` 는 수정하고 싶은 속성만 이름 지정 매개변수(Named Parameter)로 전달 받는데, 만약 해당 매개변수가 `null` 인 경우 기존 인스턴스가 가진 속성을 전달해 새 인스턴스를 생성합니다.

04. JSON & 직렬화

▼ JSON

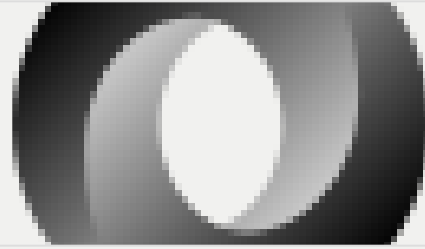


JSON(JavaScript Object Notation)은 데이터를 표현하는 규칙(포맷)입니다.

JSON - 위키백과, 우리 모두의 백과사전

JSON(제이슨, JavaScript Object Notation)은 속성-값 쌍 (attribute-value pairs), 배열 자료형(array data types) 또는 기타 모든 시리얼화 가능한 값(serializable value) 또는 "키-

🌐 <https://ko.wikipedia.org/wiki/JSON>



자료형	설명	예시
Number	모든 숫자를 나타냅니다.	1 -10 2.5
String	항상 쌍따옴표(")로 묶어야합니다.	"Hello" "반가워"
Boolean	참거짓	true false
Array	대괄호[]로 나타냅니다. Dart의 List와 동일합니다.	[1, "안녕"]
Object	중괄호{}로 나타냅니다. Dart의 Map과 동일합니다.	{"name": "철수"}


```
{
  "name": "철수",
  "age": 10
}
```



문자열에서 단따옴표('문자열')가 아닌 쌍따옴표("문자열")를 사용해야 합니다.



네트워크를 통해 다른 컴퓨터로 데이터 전송할 때 일련의 바이트(문자열)로 전달하는데, 이때 **JSON 포맷을 따르는 문자열**로 보내면 수신측에선 데이터를 다루기 편리합니다.

 <https://gist.githubusercontent.com/nero-angela/6bcd87a8fb61113da4d20f6d5e8e8656/raw/15c69ceeb0b395330548f2d26d18c99446f742cd/jsonDummy.json>

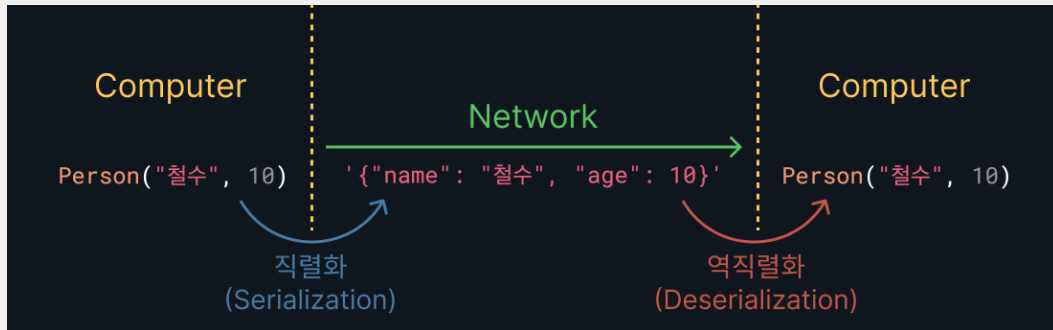
```
[
  {
    "name": "철수",
    "age": 10
  },
  {
    "name": "영희",
    "age": 11
  }
]
```

▼ 직렬화 & 역직렬화



직렬화(Serialization) : Dart 클래스 → 일련의 문자열

역직렬화(Deserialization) : 일련의 문자열 → Dart 클래스



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



직렬화(Serialization)와 역직렬화(Deserialization)를 하는 이유

- 네트워크 전송시, **직렬화**해야 데이터를 보낼 수 있습니다.
- 네트워크 수신시, **역직렬화**를 하지 않으면 다음과 같은 불편함이 있습니다.
 1. 원하는 값을 추출하기 어려움

```
String data = '{"name":"철수", "age":10}';
String name = data.substring(9, 11);
print(name); // "철수"
```


2. 데이터 자료형이 모두 `String` 이므로 다른 타입의 내장 메소드를 쓸 수 없음

```
String data = '{"name":"철수", "age":10}';
String age = data.substring(20, 22);
print(age); // 문자열 10
```



일련의 문자열이 **JSON 포맷을 따른다면** 손쉽게 **직렬화** & **역직렬화**를 구현할 수 있습니다

DartPad

 <https://dartpad.dev/?id=049a68bc329d2c494299ec2b30704450>

```
1 import 'dart:convert';
2
3 void main() {
4   /// JSON 포맷의 문자열
5   String json = '{"name":"철수", "age":10}';
6   print(json.substring(9, 11)); // "철수"
7
8   /// 역직렬화(Deserialization) : String -> Map
9   Map<String, dynamic> jsonMap = jsonDecode(json);
10  print(jsonMap["name"]); // "철수"
11
12  /// 직렬화(Serialization) : Map -> String
13  String jsonString = jsonEncode(jsonMap);
14  print(jsonString.substring(9, 11)); // "철수"
15 }
16
17
18
```

- 직렬화(Serialization)

```
import 'dart:convert';

String jsonString = jsonEncode({"name": "철수"});
```

- 역직렬화(Deserialization)

```
import 'dart:convert';

Map<String, dynamic> map = jsonDecode('{"name": "철수"}');
```

▼ 직렬화 & 역직렬화 구현



VSCode에서 **직렬화(Serialization)** & **역직렬화(Deserialization)**를 구현해 봅시다.



VSCode `custom_class` 폴더 밑에 `json.dart` 파일을 만들고 다음과 같이 작성해 주세요.

▼ 코드스니펫 - `json.dart` : 시작

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

  const Person({
    required this.name,
    required this.age,
  });
}

void main() {
  /// 네트워크 응답 문자열
  String jsonString = '{"name": "철수", "age": 10}';
}
```

The screenshot shows the VS Code interface. On the left, the file explorer displays the project structure under 'DART_PRACTICE', with 'bin > custom_class' expanded and 'json.dart' selected. The editor on the right shows the content of 'json.dart', which matches the code block above. The code is line-numbered from 1 to 15. The status bar at the bottom indicates 'Run | Debug'.

서버에서 13번째 라인과 같은 JSON 포맷을 따르는 문자열을 응답 받았다고 가정해 봅시다.

▼ 역직렬화 구현



JSON 포맷 문자열을 Person 클래스로 역직렬화(Deserialization)해 봅시다.



역직렬화(Deserialization) 진행 순서

1. JSON 포맷 String → Map<String, dynamic>
2. Map<String, dynamic> → Person 클래스

1. JSON 포맷 문자열 이기 때문에 jsonDecode() 함수를 이용하면 Map<String, dynamic> 으로 손쉽게 변경할 수 있습니다.

▼ 코드스니펫 - json.dart : jsonDecode()

```
/// JSON 포맷 String -> Map<String, dynamic>
Map<String, dynamic> jsonMap = jsonDecode(jsonString);
print(jsonMap);
```

```
1  import 'dart:convert';
2
3  class Person {
4    final String name;
5    final int age;
6
7    const Person({
8      required this.name,
9      required this.age,
10   });
11 }
12
13 void main() {
14   /// 네트워크 응답 문자열
15   String jsonString = '{"name": "철수", "age": 10}';
16
17   /// JSON 포맷 String -> Map<String, dynamic>
18   Map<String, dynamic> jsonMap = jsonDecode(jsonString);
19   print(jsonMap);
20 }
21
```



`import 'dart:convert';` 는 자동 완성을 이용해 추가해 주세요.



Run 버튼을 눌러 실행하면 다음과 같은 결과가 출력됩니다.

```

12
Run | Debug
13 void main() {
14     /// 네트워크 응답 문자열
15     String jsonString = '{"name": "철수", "age": 10}';
16
17     /// JSON 포맷 String -> Map<String, dynamic>
18     Map<String, dynamic> jsonMap = jsonDecode(jsonString);
19     print(jsonMap);
20 }
21
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
{name: 철수, age: 10}
Exited

```



`Map` 으로 변환하였기 때문에 `jsonMap["name"]` 또는 `jsonMap["age"]` 등의 방법으로 원하는 값을 손쉽게 꺼내어 아래와 같이 `Person` 인스턴스를 만들 수 있습니다.

```

Person person = Person(
    name: jsonMap["name"],
    age: jsonMap["age"],
);

```

- 일반적으로 `Map<String, dynamic>` 으로 부터 `클래스 인스턴스` 를 반환하는 코드는 클래스에 `fromJson()` 이라는 메소드를 생성하여 구현합니다. 아래 이미지와 같이 코드를 작성해 주세요.

▼ 코드스니펫 - `json.dart` : `Person.fromJson()`

```

factory Person.fromJson(Map<String, dynamic> json) {
    return Person(
        name: json['name'],

```

```
        age: json['age'],
      );
    }
  }
```

```
/// Map<String, dynamic> -> Person
Person person = Person.fromJson(jsonMap);
print(person);
```

```
1  import 'dart:convert';
2
3  class Person {
4    final String name;
5    final int age;
6
7    const Person({
8      required this.name,
9      required this.age,
10   });
11
12   factory Person.fromJson(Map<String, dynamic> json) {
13     return Person(
14       name: json['name'],
15       age: json['age'],
16     );
17   }
18 }
19
20 Run | Debug
21 void main() {
22   /// 네트워크 응답 문자열
23   String jsonString = '{"name": "철수", "age": 10}';
24
25   /// JSON 포맷 String -> Map<String, dynamic>
26   Map<String, dynamic> jsonMap = jsonDecode(jsonString);
27   print(jsonMap);
28
29   /// Map<String, dynamic> -> Person
30   Person person = Person.fromJson(jsonMap);
31   print(person);
32 }
```



factory 키워드를 붙여 생성자 메소드로 만들 수 있으며, 다음 규칙을 지켜야 합니다.

- **클래스 인스턴스** 를 반환해야 합니다.
- 메소드명을 **클래스명.메소드명()** 형태로 작성해야 합니다.



Run 버튼을 눌러 실행하면 Person 클래스의 인스턴스가 반환됨을 확인할 수 있습니다.

```

Run | Debug
20 void main() {
21   /// 네트워크 응답 문자열
22   String jsonString = '{"name": "철수", "age": 10}';
23
24   /// JSON 포맷 String -> Map<String, dynamic>
25   Map<String, dynamic> jsonMap = jsonDecode(jsonString);
26   print(jsonMap);
27
28   /// Map<String, dynamic> -> Person
29   Person person = Person.fromJson(jsonMap);
30   print(person);
31 }

```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL

```

{name: 철수, age: 10}
Instance of 'Person'
Exited

```



보통 네트워크에 데이터를 받을 때 위와 같이 **JSON 포맷 String** → **Map<String, dynamic>** → **Person 인스턴스** 로 변환하는 과정이 필요합니다.

▼ 직렬화 구현



Person 클래스 를 **JSON 포맷 문자열** 로 **직렬화(Serialization)**해 봅시다.



직렬화(Serialization) 진행 순서

1. `Person` 클래스 → `Map<String, dynamic>`
2. `Map<String, dynamic>` → JSON 포맷 `String`

1. 클래스의 인스턴스를 `Map<String, dynamic>` 으로 변경하는 함수는 일반적으로 클래스에 `toJson()` 메소드를 만들어 진행합니다. 아래 이미지와 같이 코드를 작성해 주세요.

▼ 코드스니펫 - `json.dart` : `toJson()`

```
Map<String, dynamic> toJson() {  
  return {  
    "name": name,  
    "age": age,  
  };  
}
```

```
/// Person -> Map<String, dynamic>  
Map<String, dynamic> personMap = person.toJson();  
print(personMap);
```

```

12  factory Person.fromJson(Map<String, dynamic> json) {
13      return Person(
14          name: json['name'],
15          age: json['age'],
16      );
17  }
18
19  Map<String, dynamic> toJson() {
20      return {
21          "name": name,
22          "age": age,
23      };
24  }
25  }
26
27  Run | Debug
28  void main() {
29      /// 네트워크 응답 문자열
30      String jsonString = '{"name": "철수", "age": 10}';
31
32      /// JSON 포맷 String -> Map<String, dynamic>
33      Map<String, dynamic> jsonMap = jsonDecode(jsonString);
34      print(jsonMap);
35
36      /// Map<String, dynamic> -> Person
37      Person person = Person.fromJson(jsonMap);
38      print(person);
39
40      /// Person -> Map<String, dynamic>
41      Map<String, dynamic> personMap = person.toJson();
42      print(personMap);
43  }

```



Run 버튼을 눌러 실행하면 `Person` 인스턴스가 `Map<String, dynamic>` 으로 변경된 것을 확인할 수 있습니다.

```
Run | Debug
27 void main() {
28   /// 네트워크 응답 문자열
29   String jsonString = '{"name": "철수", "age": 10}';
30
31   /// JSON 포맷 String -> Map<String, dynamic>
32   Map<String, dynamic> jsonMap = jsonDecode(jsonString);
33   print(jsonMap);
34
35   /// Map<String, dynamic> -> Person
36   Person person = Person.fromJson(jsonMap);
37   print(person);
38
39   /// Person -> Map<String, dynamic>
40   Map<String, dynamic> personMap = person.toJson();
41   print(personMap);
42 }
43
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
{name: 철수, age: 10}
Instance of 'Person'
{name: 철수, age: 10}
Exited
```

2. `jsonEncode()` 함수를 이용하면 `Map<String, dynamic>` 을 `JSON 포맷 String` 으로 변경할 수 있습니다.

▼ 코드스니펫 - `json.dart` : `jsonEncode()`

```
/// Map<String, dynamic> -> JSON 포맷 String
String personString = jsonEncode(personMap);
print(personString);
```



```
Run | Debug
27 void main() {
28   /// 네트워크 응답 문자열
29   String jsonString = '{"name": "철수", "age": 10}';
30
31   /// JSON 포맷 String -> Map<String, dynamic>
32   Map<String, dynamic> jsonMap = jsonDecode(jsonString);
33   print(jsonMap);
34
35   /// Map<String, dynamic> -> Person
36   Person person = Person.fromJson(jsonMap);
37   print(person);
38
39   /// Person -> Map<String, dynamic>
40   Map<String, dynamic> personMap = person.toJson();
41   print(personMap);
42
43   /// Map<String, dynamic> -> JSON 포맷 String
44   String personString = jsonEncode(personMap);
45   print(personString);
46 }
47
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
{name: 철수, age: 10}
Instance of 'Person'
{name: 철수, age: 10}
{"name":"철수","age":10}
Exited
```



보통 네트워크에 데이터를 보낼 때 위와 같이 **Person 인스턴스** → **Map<String, dynamic>** → **JSON 포맷 String** 으로 변환하는 과정이 필요합니다.

▼ 코드스니펫 - **json.dart** : 최종

```
import 'dart:convert';

class Person {
  final String name;
  final int age;

  const Person({
    required this.name,
    required this.age,
  });
```

```

factory Person.fromJson(Map<String, dynamic> json) {
  return Person(
    name: json['name'],
    age: json['age'],
  );
}

Map<String, dynamic> toJson() {
  return {
    "name": name,
    "age": age,
  };
}

void main() {
  /// 네트워크 응답 문자열
  String jsonString = '{"name": "철수", "age": 10}';

  /// JSON 포맷 String -> Map<String, dynamic>
  Map<String, dynamic> jsonMap = jsonDecode(jsonString);
  print(jsonMap);

  /// Map<String, dynamic> -> Person
  Person person = Person.fromJson(jsonMap);
  print(person);

  /// Person -> Map<String, dynamic>
  Map<String, dynamic> personMap = person.toJson();
  print(personMap);

  /// Map<String, dynamic> -> JSON 포맷 String
  String personString = jsonEncode(personMap);
  print(personString);
}

```

05. 코드 생성기

▼ 등장 배경



커스텀 클래스를 만들 때 매번 반복하여 구현하게 되는 코드들이 있습니다.

```

class Person {
  final String name;
  final int age;

  const Person({
    required this.name,
    required this.age,
  });

  Person copyWith({
    String? name,
    int? age,
  }) {
    return Person(
      name: name ?? this.name,
      age: age ?? this.age,
    );
  }

  factory Person.fromJson(Map<String, dynamic> json) {
    return Person(
      name: json['name'] ?? '',
      age: json['age'] ?? 0,
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'name': name,
      'age': age,
    };
  }

  @override
  bool operator ==(Object other) =>
    identical(this, other) ||
    other is Person &&
    runtimeType == other.runtimeType &&
    name == other.name &&
    age == other.age;

  @override
  int get hashCode => Object.hash(
    name,
    age,
  );

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

```

<https://dartpad.dev/?id=6dbf0a9a1d97674ec72450e13bfe287a>

- 객체 복사 : `copyWith()`

- JSON 변환 : `fromJson()` & `toJson()`
- 값 비교 : `==` & `hashCode`
- 객체 로깅 : `toString()`



다음 패키지들을 이용하면, 반복되는 코드들을 자동으로 생성할 수 있습니다.

- build_runner : 코드 생성
- freezed & freezed_annotation : `==`, `hashCode`, `copyWith()`, `toString()`
- json_serializable & json_annotation : `fromJson()`, `toJson()`