코드팩토리의 플러터 프로그래밍 2판

인프런 베스트셀러 강사와 함께 현업 수준으로 실력 끌어올리기

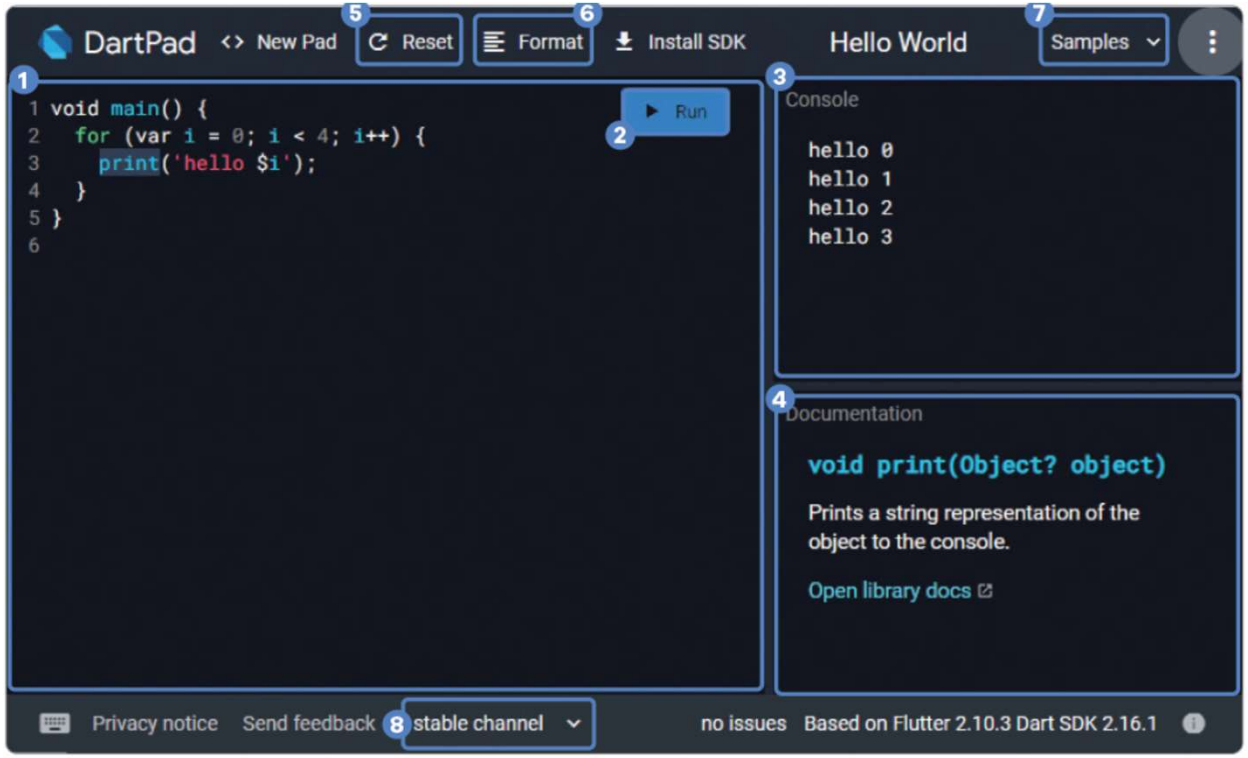
01 다트 입문하기

1.1 다트 소개

- 구글이 2011년 10월에 공개한 프로그래밍 언어(Dart programming language)
- 플러터의 인기에 힘입어 모바일 영역에서 큰 각광을 받고 있음
- UI(User Interface)를 제작하는 데 최적화
  - 비동기 언어, 이벤트 기반
  - 아이솔레이트(Isolate)를 이용한 동시성 기능 제공
- 효율적으로 UI를 코딩할 수 있는 기능 제공
  - 널 안정성(Null Safety)
  - 스프레드 기능(Spread Operator)
  - 컬렉션 if문(Collection If)
- 효율적인 개발 환경 제공
  - 멀티 플랫폼에서 로깅 및 디버깅을 하고 실행 가능
  - AOT 컴파일이 가능하기 때문에 어떤 플랫폼에서든 빠른 속도를 자랑
  - 자바스크립트로의 완전한 컴파일 지원
  - 백엔드 프로그래밍 지원

1.2 문법 공부 환경 안내(다트패드)

- https://dartpad.dev 접속



- ❶ 코드 편집 영역
- ❷ [Run] 버튼
- ❸ 콘솔 영역에 실행 결과 출력
- ❹ 함수나 변수 관련 설명
- ❺ [Reset] 버튼
- ❻ [Format] 버튼
- ❼ [Samples] 버튼
- ❽ [stable channel] 버튼

1.2 문법 공부 환경 안내(안드로이드 스튜디오)

- main.dart 파일에 기본 생성되는 코드 모두 삭제 후 main( ) 함수만 남김

lib/main.dart

```
void main() {
 print('hello world');
}
```

- [Terminal] 탭에서 'dart lib/main.dart' 명령어 실행

```
Terminal:  Local  +
(base) jihochoi@Jiui-MacBookPro test_proj_3 % dart lib/main.dart
hello world
(base) jihochoi@Jiui-MacBookPro test_proj_3 %
```

- main.dart 파일에 코드 작성 후 'dart lib/main.dart' 명령으로 코드 실행

1.3 기초 문법

- 메인 함수

```
void main() {

}
```

  - 중괄호 사이에 원하는 코드를 입력
  - ( ) 안에 입력받을 매개변수 지정

- print( ) 함수

```
void main(){
 // 콘솔에 출력
 print('Hello World');
}
```

  - 문자열을 콘솔에 출력하는 함수

- 주석

```
void main(){
 // 주석을 작성하는 첫 번째 방법은
 // 한 줄 주석입니다.

 /*
  * 여러 줄 주석 방법입니다.
  * 시작 기호는 /*이고 끝나는 기호는 */입니다.
  * 필수는 아니지만 관행상 중간 줄의 시작으로 *를 사용합니다.
  * */

 /// 슬래시 세 개를 사용하면
 /// 문서 주석을 작성할 수 있습니다.
 /// DartDoc이나 안드로이드 스튜디오 같은
 /// IDE에서 문서(Documentation)로 인식합니다.
}
```

  - 프로그램에서 코드로 인식하지 않는 부분

1.3 기초 문법

- var를 사용한 변수 선언
  - 변수 선언: **var 변수명 = 값;**

```
void main(){
 var name = '코드팩토리';
 print(name);

 // 변숫값 변경 가능
 name = '골든래빗';
 print(name);

 // 변수명 중복은 불가능
 // 그래서 다음 코드에서 주석을 제거하면 코드에서 에러 발생
 // var name = '김고은';
}
```

▼실행 결과

```
코드팩토리
골든래빗
```

1.3 기초 문법

- dynamic을 사용한 변수 선언
  - dynamic 키워드를 사용하면 변수의 타입이 고정되지 않아 다른 타입의 값을 저장 가능

```
void main() {
 dynamic name = '코드팩토리';
 name = 1;
}
```

1.3 기초 문법

- final/const를 사용한 변수 선언
  - final과 const 키워드는 변수의 값을 처음 선언 후 변경할 수 없음

```
void main() {
 final String name = '블랙핑크';
 name = 'BTS';   // 에러 발생. final로 선언한 변수는 선언 후 값을 변경할 수 없음

 const String name2 = 'BTS';
 name2 = '블랙핑크';  // 에러 발생. const로 선언한 변수는 선언 후 값을 변경할 수 없음
}
```

  - final은 런타임(실행될 때 값이 확정), const는 빌드 타임 상수(실행하지 않은 상태에서 값이 확정)

```
void main() {
 final DateTime now = DateTime.now();

 print(now);
}
```

```
void main() {
 // 에러
 const DateTime now = DateTime.now();

 print(now);
}
```

1.3 기초 문법

- 변수 타입

```
void main(){
 // String - 문자열
 String name = '코드팩토리';

 // int - 정수
 int isInt = 10;

 // double - 실수
 double isDouble = 2.5;

 // bool - 불리언 (true/false)
 bool isTrue = true;
 print(name);
 print(isInt);
 print(isDouble);
 print(isTrue);
}
```

▼실행 결과

```
코드팩토리
10
2.5
true
```

1.4 컬렉션(List 타입)

- 여러 값을 순서대로 나열한 변수에 저장할 수 있는 타입
- **리스트명[인덱스]** 형식으로 특정 원소에 접근
- 마지막 원소는 '리스트 길이 -1'로 지정해야 함

```
void main() {
 // 리스트에 넣을 타입을 <> 사이에 명시할 수 있습니다.
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 print(blackPinkList);
 print(blackPinkList[0]);  // 첫 원소 지정
 print(blackPinkList[3]);  // 마지막 원소 지정

 print(blackPinkList.length);  // ❶ 길이 반환

 blackPinkList[3] = '코드팩토리';  // 3번 인덱스값 변경
 print(blackPinkList);
}
```

```
[리사, 지수, 제니, 로제]
리사
로제
4
[리사, 지수, 제니, 코드팩토리]
```

1.4 컬렉션(List 타입)

- add( ) 함수
  - List에 값을 추가할 때 사용
  - 추가하고 싶은 값을 매개변수에 입력

```
void main() {
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 blackPinkList.add('코드팩토리');  // 리스트의 끝에 추가

 print(blackPinkList);
}
```

▼실행 결과

```
[리사, 지수, 제니, 로제, 코드팩토리]
```

1.4 컬렉션(List 타입)

- where( ) 함수
  - List에 있는 값들을 순서대로 순회(looping)하면서 특정 조건에 맞는 값만 필터링
  - 매개변수에 함수 입력, 입력된 함수는 기존 값을 하나씩 매개변수로 입력받음
  - 각 값별로 true를 반환하면 값을 유지, false를 반환하면 값을 버림

```
void main() {
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 final newList = blackPinkList.where(
     (name) => name == '리사' || name == '지수',  // '리사' 또는 '지수'만 유지
 );

 print(newList);
 print(newList.toList()); // Iterable을 List로 다시 변환할 때  .toList() 사용
}
```

```
(리사, 지수)
[리사, 지수]
```

1.4 컬렉션(List 타입)

- map( ) 함수
  - List에 있는 값들을 순서대로 순회하면서 값을 변경
  - 매개변수에 함수 입력, 입력된 함수는 기존 값을 하나씩 매개변수로 입력받음
  - 반환하는 값이 현잿값을 대체하며 순회가 끝나면 Iterable 반환

```
void main() {
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 final newBlackPink = blackPinkList.map(
     (name) => '블랙핑크 $name',  // 리스트의 모든 값 앞에 '블랙핑크' 추가
 );
 print(newBlackPink);

 // Iterable을 List로 다시 변환하고 싶을 때 .toList() 사용
 print(newBlackPink.toList());
}
```

```
(블랙핑크 리사, 블랙핑크 지수, 블랙핑크 제니, 블랙핑크 로제)
[블랙핑크 리사, 블랙핑크 지수, 블랙핑크 제니, 블랙핑크 로제]
```

1.4 컬렉션(List 타입)

- reduce( ) 함수
  - List에 있는 값들을 순서대로 순회하면서 매개변수에 입력된 함수(매개변수 2개)를 실행
  - 단, 순회할 때마다 값을 쌓아가는 특징이 있음
  - List 멤버의 타입과 같은 타입을 반환

```
void main() {
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 final allMembers = blackPinkList.reduce((value, element) => value + ', ' +
  element);  // ❶ 리스트를 순회하며 값들을 더합니다.

 print(allMembers);
}
```

```
리사, 지수, 제니, 로제
```

1.4 컬렉션(List 타입)

- fold( ) 함수
  - reduce( ) 함수와 실행되는 논리는 동일하지만 어떠한 타입이든 변환 가능

```
void main() {
 List<String> blackPinkList = ['리사', '지수', '제니', '로제'];

 // ❶ reduce() 함수와 마찬가지로 각 요소를 순회하며 실행됩니다.
 final allMembers =
     blackPinkList.fold<int>(0, (value, element) => value + element.length);

 print(allMembers);
}
```

```
8
```

1.4 컬렉션(Map 타입)

- 키(key)와 값(value)의 짝을 저장
- **Map<키 타입, 값 타입> 맵이름** 형식으로 생성

```
void main() {
 Map<String, String> dictionary = {
   'Harry Potter': '해리 포터',        // 키 : 값
   'Ron Weasley': '론 위즐리',
   'Hermione Granger': '헤르미온느 그레인저',
 };
 print(dictionary['Harry Potter']);
 print(dictionary['Hermione Granger']);
}
```

▼실행 결과

```
해리 포터
헤르미온느 그레인저
```

1.4 컬렉션(Map 타입)

- 키와 값 반환받기
  - 값을 반환받고 싶은 Map 타입의 변수에 key와 value 게터 실행

```
void main() {
 Map<String, String> dictionary = {
   'Harry Potter': '해리 포터',
   'Ron Weasley': '론 위즐리',
   'Hermione Granger': '헤르미온느 그레인저',
 };

 print(dictionary.keys);
 // Iterable이 반환되기 때문에 .toList()를 실행해서 List를 반환받을 수도 있음
 print(dictionary.values);
}
```

```
(Harry Potter, Ron Weasley, Hermione Granger)
(해리 포터, 론 위즐리, 헤르미온느 그레인저)
```

1.4 컬렉션(Set 타입)

- 중복 없는 값들의 집합
- **Set<타입> 세트이름** 형식으로 생성
- 각 값의 유일(unique)함을 보장받을 수 있음

```
void main() {
 Set<String> blackPink = {'로제', '지수', '리사', '제니', '제니'}; // ❶ 제니 중복

 print(blackPink);
 print(blackPink.contains('로제')); // ❷ 값이 있는지 확인하기
 print(blackPink.toList());         // ❸ 리스트로 변환하기

 List<String> blackPink2 = ['로제', '지수', '지수'];
 print(Set.from(blackPink2));  // ❹ List 타입을 Set 타입으로 변환
}
```

```
{로제, 지수, 리사, 제니}
true
[로제, 지수, 리사, 제니]
{로제, 지수}
```

1.4 컬렉션(enum)

- 한 변수의 값을 몇 가지 옵션으로 제한하는 기능
- 선택지가 제한적일 때 사용
- String으로 대체할 수 있지만 enum은 기본적으로 자동 완성이 지원되고 정확히 어떤 선택지가
  존재하는지 정의해둘 수 있어 유용

```
enum Status {
 approved,
 pending,
 rejected,
}

void main() {
 Status status = Status.approved;
 print(status);  // Status.approved
}
```

▼실행 결과

```
Status.approved
```

1.5 연산자(기본 수치 연산자)

- 기본 산수 기능 제공
  - 다트패드로 실습하면 소수점이 없는 정수로 출력

```dart
void main() {
 double number = 2;

 print(number + 2); // 4 출력
 print(number - 2); // 0 출력
 print(number * 2); // 4 출력
 print(number / 2); // 1 출력. 나눈 몫
 print(number % 3); // 2 출력. 나눈 나머지

 // 단항 연산도 됩니다.
 number++; // 3
 number--; // 2
 number += 2; // 4
 number -= 2; // 0
 number *= 2; // 4
 number /= 2; // 1
}
```

▼실행 결과

```
4.0
0.0
4.0
1.0
2.0
```

1.5 연산자(null 관련 연산자)

- null 아무 값도 없음을 뜻하며, 0과는 다름
- 다트 언어에서는 변수 타입이 null값을 가지는지 여부를 직접 지정해줘야 함
- 타입 뒤에 ?를 추가해줘야 null값 저장 가능

```
void main() {
 // 타입 뒤에 ?를 명시해서 null값을 가질 수 있습니다.
 double? number1 = 1;

 // 타입 뒤에 ?를 명시하지 않아 에러가 납니다.
 double number2 = null;
}
```

1.5 연산자(null 관련 연산자)

- null을 가질 수 있는 변수에 새로운 값을 추가할 때 ??를 사용하면 기존에 null일 때만 값이 저장되도록 할 수 있음
  - 다트패드로 실습하면 소수점이 없는 정수로 출력

```
void main() {
 double? number; // 자동으로 null값 지정
 print(number);

 number ??= 3;   // ??를 사용하면 기존 값이 null일 때만 저장됩니다.
 print(number);

 number ??= 4;   // null이 아니므로 3이 유지됩니다.
 print(number);
}
```

▼실행 결과

```
null
3.0
3.0
```

1.5 연산자(값 비교 연산자)

- 정수 크기를 비교하는 연산자

```
void main() {
 int number1 = 1;
 int number2 = 2;

 print(number1 > number2); // false
 print(number1 < number2); // true
 print(number1 >= number2); // false
 print(number1 <= number2); // true
 print(number1 == number2); // false
 print(number1 != number2); // true
}
```

1.5 연산자(타입 비교 연산자)

- is 키워드를 사용해 변수의 타입을 비교

```
void main() {
 int number1 = 1;

 print(number1 is int);    // true
 print(number1 is String); // false
 print(number1 is! int);   // false. !는 반대를 의미합니다(int 타입이 아닌 경우 true).
 print(number1 is! String);// true
}
```

1.5 연산자(논리 연산자)

- and와 or을 의미하는 연산자

```
void main() {
 bool result = 12 > 10 && 1 > 0; // 12가 10보다 크고 1이 0보다 클 때
 print(result); // true

 bool result2 = 12 > 10 && 0 > 1; // 12가 10보다 크고 0이 1보다 클 때
 print(result2); // false

 bool result3 = 12 > 10 || 1 > 0; // 12가 10보다 크거나 1이 0보다 클 때
 print(result3); // true

 bool result4 = 12 > 10 || 0 > 1; // 12가 10보다 크거나 0이 1보다 클 때
 print(result4); // true

 bool result5 = 12 < 10 || 0 > 1; // 12가 10보다 작거나 0이 1보다 클 때
 print(result5); // false
}
```

1.6 제어문(if문)

- if문은 원하는 조건을 기준으로 다른 코드를 실행하고 싶을 때 사용
- if문 → else if문 → else문의 순서대로 괄호 안에 작성한 조건이 true이면 해당 조건의 코드 블록 실행

```
void main() {
 int number = 2;

 if (number % 3 == 0) {
   print('3의 배수입니다.');
 } else if (number % 3 == 1) {
   print('나머지가 1입니다.');
 } else {
   // 조건에 맞지 않기 때문에 다음 코드 실행
   print('맞는 조건이 없습니다.');
 }
}
```

▼실행 결과

```
맞는 조건이 없습니다.
```

1.6 제어문(switch문)

- 입력된 상수값에 따라 알맞은 case 블록 수행
- break 키워드를 사용하면 switch문 밖으로 나갈 수 있음

```
enum Status {
 approved,
 pending,
 rejected,
}

void main() {
 Status status = Status.approved;

 switch (status) {
   case Status.approved:
     // approved값이기 때문에 다음 코드가 실행됩니다.
     print('승인 상태입니다.');
     break;
   case Status.pending:
     print('대기 상태입니다.');
     break;
   case Status.rejected:
     print('거절 상태입니다.');
     break;
   default:
     print('알 수 없는 상태입니다.');
 }

 // Enum의 모든 수를
 // 리스트로 반환합니다.
 print(Status.values);
}
```

▼실행 결과

```
승인 상태입니다.
[Status.approved, Status.pending, Status.rejected]
```

1.6 제어문(for문)

- 작업을 여러 번 반복해서 실행할 때 사용

```
void main() {
 // 값 선언; 조건 설정; loop 마다 실행할 기능
 for (int i = 0; i < 3; i++) {
   print(i);
 }
}
```

▼실행 결과

```
0
1
2
```

1.6 제어문(for문)

- for...in 패턴의 for문도 제공
- List의 모든 값을 순회하고 싶을 때 사용

```
void main() {

 List<int> numberList = [3, 6, 9];

 for (int number in numberList) {
   print(number);
 }
}
```

▼실행 결과

```
3
6
9
```

1.6 제어문(while문과 do...while문)

- for문과 마찬가지로 반복적인 작업을 실행할 때 사용
- while문은 조건을 기반으로 반복문을 실행
  - 조건이 true이면 계속 실행, false이면 멈춤

```
void main() {
 int total = 0;

 while(total < 10) {  // total값이 10보다 작으면 계속 실행
   total += 1;
 }

 print(total);
}
```

▼실행 결과

```
10
```

1.6 제어문(while문과 do...while문)

- do...while문은 반복문을 실행한 후 조건을 확인

```
void main() {
 int total = 0;

 do {
   total += 1;
 } while(total < 10);

 print(total);
}
```

▼실행 결과

```
10
```

1.7 함수와 람다

- 함수를 사용하면 한 번만 작성하고 여러 곳에서 재활용할 수 있음
- 반환할 값이 없을 때는 void 키워드 사용

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

void main() {
 print(addTwoNumbers(1, 2));
}
```

▼실행 결과

```
3
```

- 매개변수를 지정하는 방법 두 가지
  - 순서가 고정된 매개변수(positional parameter, 위치 매개변수)
  - 이름이 있는 매개변수(named parameter, 명명된 매개변수)

1.7 함수와 람다

- 네임드 파라미터를 지정하려면 중괄호 { }와 required 키워드 사용
  - required 키워드는 매개변수가 null값이 불가능한 타입이면 기본값을 지정해주거나 필수로 입력해야 한다는 의미

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

void main() {
 print(addTwoNumbers(a: 1, b: 2));
}
```

▼실행 결과

```
3
```

1.7 함수와 람다

- 기본값을 갖는 포지셔널 파라미터 지정하기

```
int addTwoNumbers(int a, [int b = 2]) {
 return a + b;
}
void main() {
 print(addTwoNumbers(1));
}
```

▼실행 결과

```
3
```

- 네임드 파라미터에 기본값 적용하기

```
int addTwoNumbers({
 required int a,
 int b = 2,
}) {
 return a + b;
}

void main() {
 print(addTwoNumbers(a: 1));
}
```

▼실행 결과

```
3
```

1.7 함수와 람다

- 포지셔널 파라미터와 네임드 파라미터 섞어서 사용하기

```
int addTwoNumbers(
 int a, {
 required int b,
 int c = 4,
}) {
 return a + b + c;
}

void main() {
 print(addTwoNumbers(1, b: 3, c: 7));
}
```

▼실행 결과

```
11
```

1.7 함수와 람다(익명 함수와 람다 함수)

- 둘 다 함수 이름이 없고 일회성으로 사용됨

익명 함수	람다 함수

(매개변수) {
  함수 바디
}
	(매개변수) => 단 하나의 스테이트먼트

  - 익명 함수에서 { }를 빼고 => 기호를 추가한 것이 람다 함수
  - 매개변수는 아예 없거나 하나 이상이어도 됨
  - 람다 함수는 함수 로직을 수행하는 스테이트먼트가 하나만 있어야 하기 때문에 적절히 사용하면 간결하게 코드 작성이 가능해 가독성이 높음

1.7 함수와 람다(익명 함수와 람다 함수)

- reduce( ) 함수를 이용해 리스트의 모든 값을 더하는 익명 함수와 람다 함수 작성하기

익명 함수

```
void main() {
 List<int> numbers = [1, 2, 3, 4, 5];

 // 일반 함수로 모든 값 더하기
 final allMembers = numbers.reduce((value, element) {
   return value + element;
 });

 print(allMembers);
}
```

람다 함수

```
void main() {
 List<int> numbers = [1, 2, 3, 4, 5];

 // 람다 함수로 모든 값 더하기
 final allMembers = numbers.reduce((value, element) => value + element);

 print(allMembers);
}
```

1.7 함수와 람다(typedef와 함수)

- typedef 키워드는 함수의 시그니처를 정의하는 값, 즉 함수 선언부를 정의하는 키워드
  - 시그니처: 반환값 타입, 매개변수 개수와 타입 등
- 함수를 선언하기는 하지만 무슨 동작을 하는지에 대한 정의는 없음

```
typedef Operation = void Function(int x, int y);
```

1.7 함수와 람다(typedef와 함수)

- 시그니처에 맞춘 함수 만들어 사용하기

```
typedef Operation = void Function(int x, int y);

void add(int x, int y) {
 print('결괏값 : ${x + y}');
}

void subtract(int x, int y) {
 print('결괏값 : ${x - y}');
}

void main() {
 // typedef는 일반적인 변수의 type처럼 사용 가능
 Operation oper = add;
 oper(1, 2);

 // subtract() 함수도 Operation에 해당되는
 // 시그니처이므로 oper 변수에 저장 가능
 oper = subtract;
 oper(1, 2);
}
```

▼실행 결과

```
결괏값 : 3
결괏값 : -1
```

1.7 함수와 람다(typedef와 함수)

- 다트에서 함수는 일급 객체(first-class citizen, 일급 시민)이므로 함수를 값처럼 사용할 수 있음
- 플러터에서는 typedef으로 선언한 함수를 매개변수로 넘어 사용

```
typedef Operation = void Function(int x, int y);

void add(int x, int y) {
 print('결괏값 : ${x + y}');
}

void calculate(int x, int y, Operation oper) {
 oper(x, y);
}

void main() {
 calculate(1, 2, add);
}
```

▼실행 결과

```
결괏값 : 3
```

1.8 try...catch

- try...catch문의 목적은 특정 코드의 실행을 시도(try)해보고 문제가 있다면 에러를 잡으라(catch)는 뜻

```
void main() {
 try{

   // 에러가 없을 때 실행할 로직
   final String name = '코드팩토리';

   print(name);  // ❶ 에러가 없으니 출력됨
 }catch(e){      // catch는 첫 번째 매개변수에 에러 정보를 전달해줍니다.

   // 에러가 있을 때 실행할 로직
   print(e);
 }
}
```

▼실행 결과

```
코드팩토리
```

1.8 try...catch

- throw 키워드를 사용해 에러를 발생시킬 수 있음

```
void main() {
 try{
   final String name = '코드팩토리';

   // ❶ throw 키워드로 고의적으로 에러를 발생시킵니다.
   throw Exception('이름이 잘못됐습니다!');

   print(name);
 }catch(e){

   // ❷ try에서 에러가 발생했으니 catch 로직이 실행됩니다.
   print(e);
 }
}
```

▼실행 결과

```
Exception: 이름이 잘못됐습니다!
```