▼ week 03

▼ 기본 위젯

> ‘Everything is a Widget’

위젯은 현재 주어진 상태를 기반으로 어떤 UI를 구현할지를 정의

위젯의 상태가 변경되면 변경 사항에 알맞게 변경된 UI를 화면에 다시 그려줌

이때 플러터 프레임워크는 기존 상태의 위젯과 새로운 상태의 위젯을 비교해서 UI 변화를 반영할 때 필요한 최소한의 변경 사항을 산출해서 화면을 그려냄

위젯은 자식을 하나만 갖는 위젯과 자식을 여럿 갖는 위젯으로 나뉨

자식을 하나만 갖는 위젯들은 대체로 child 매개변수를 입력받음

- Container 위젯 : 자식을 담는 컨테이너 역할

  단순하게 자식을 담는 역할을 하는 게 아니라 배경색, 너비와 높이, 테두리 등의 디자인 지정

- GestureDetector 위젯 : 플러터에서 제공하는 제스처 기능을 자식 위젯에서 인식하는 위젯

  탭이나 드래그 그리고 더블 클릭 같은 제스처 기능이 자식 위젯에 인식됐을 때 함수 실행 가능

- SizedBox 위젯 : 높이와 너비를 지정하는 위젯

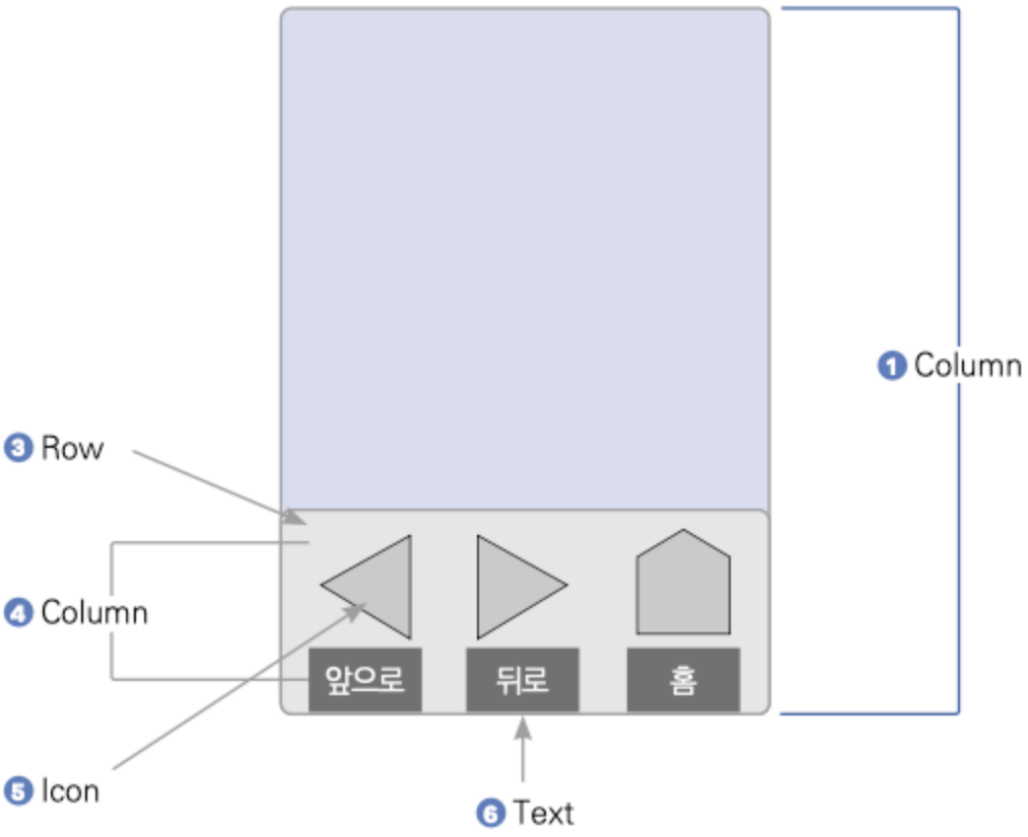
  Container 위젯과 다르게 디자인적 요소는 적용할 수 없고 const 생성자로 선언할 수 있어서 퍼포먼스 측면에서 더 효율적

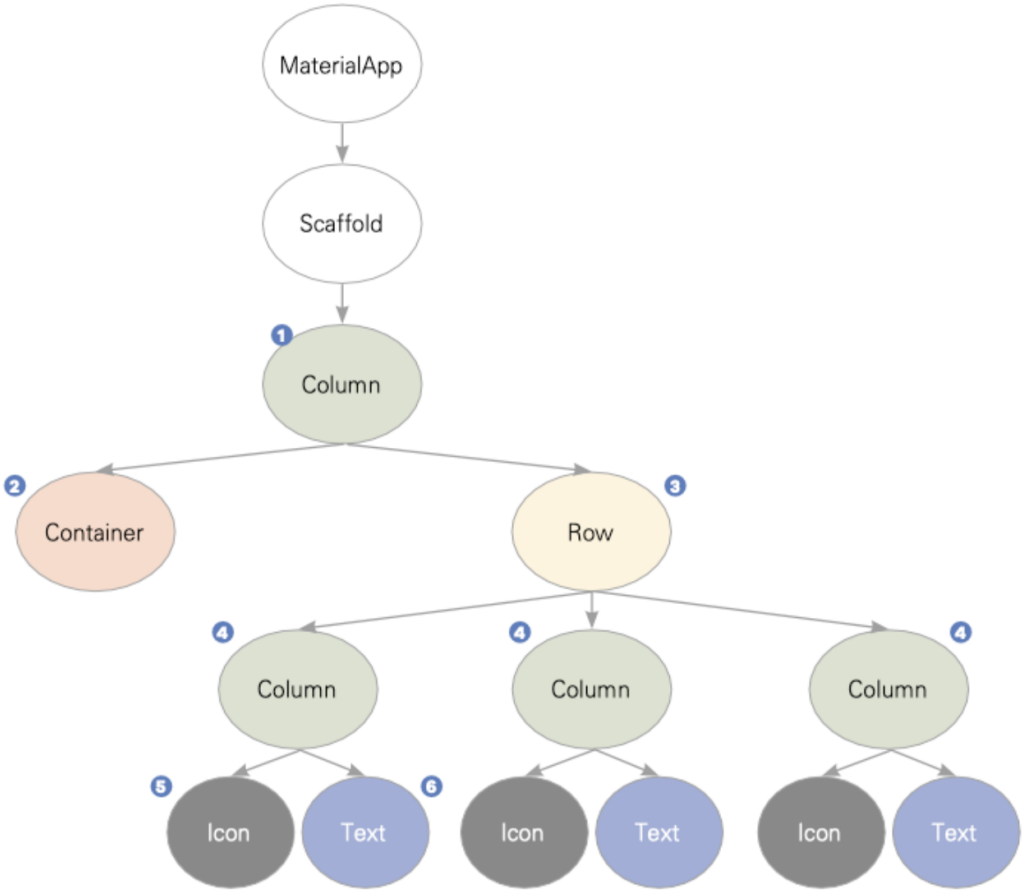
다수의 자식을 입력할 수 있는 위젯은 children 매개변수를 입력받으며 리스트로 여러 위젯 입력 가능

- Column 위젯 : children 매개변수에 입력된 모든 위젯들을 세로로 배치
- Row 위젯 : children 매개변수에 입력된 모든 위젯들을 가로로 배치
- ListView 위젯 : 리스트를 구현할 때 사용

  마찬가지로 children 매개변수에 다수의 위젯을 입력할 수 있으며 입력된 위젯이 화면을 벗어나게 되면 스크롤 가능

  child와 children 매개변수에 지속적으로 하위 위젯을 입력하면 크리스마스 트리처럼 위젯 계층이 정리


1 Column
3 Row
4 Column
앞으로
뒤로
홈
5 Icon
6 Text


MaterialApp
Scaffold
1
Column
2
Container
3
Row
4
Column
4
Column
4
Column
5
Icon
6
Text
Icon
Text
Icon
Text

Child와 Children의 차이점

플러터는 위젯 아래에 계속 위젯이 입력되는 형태로 '위젯 트리'를 구성하여 UI를 제작

child 매개변수와 children 매개변수는 위젯에 하위 위젯을 추가할 때 사용

child는 위젯 하나만 추가 가능,

children은 위젯 여러개 추가 가능

대부분 위젯은 child 또는 children 매개변수를 하나만 제공하며 child와 children 매개변수를 동시에 입력받는 위젯은 존재하지 않음

- 텍스트 관련 위젯
  - Text 위젯

```
Text(
// 작성하고 싶은 글
'seonga',
// 글자에 스타일 적용
 style: TextStyle(
 // 글자 크기
 fontSize: 16.0,
 // 글자 굵기
 fontWeight: FontWeight.w700,
 // 글자 색상
 color: Colors.blue,
 ),
),
```

- 제스처 관련 위젯

TextButton	OutlinedButton	ElevatedButton
텍스트만 있는 버튼	테두리가 있는 버튼	입체적으로 튀어나온 느낌의 배경이 들어간 버튼
텍스트 버튼	아웃라인드 버튼	엘리베이티드 버튼

- Button 위젯
  - TextButton 위젯

```
TextButton(
// 클릭 시 실행
onPressed: () {},
// 스타일 지정
style: TextButton.styleFrom(
 // 주 색상 지정
 foregroundColor: Colors.red,
 ),
 // 버튼에 넣을 위젯
 child: Text('텍스트 버튼'),
),
```

  - OutlinedButton 위젯

```
OutlinedButton(
 // 클릭 시 실행할 함수
 onPressed: () {},
 // 버튼 스타일
 style: OutlinedButton.styleFrom(
   foregroundColor: Colors.red,
 ),
 // 버튼에 들어갈 위젯
 child: Text('아웃라인드 버튼'),
),
```

  - ElevatedButton 위젯

```
ElevatedButton(
 // 클릭 시 실행할 함수
 onPressed: (){},
 // 버튼 스타일링
 style: ElevatedButton.styleFrom(
   backgroundColor: Colors.red,
 ),
 // 버튼에 들어갈 위젯
```

```
 child: Text('엘레베이티드 버튼'),
),
```

- IconButton 위젯

아이콘을 버튼으로 생성하는 위젯으로 icon 매개변수에 보여주고 싶은 아이콘을 넣을 수 있음

onPressed 매개변수에 IconButton을 누르면 실행할 콜백 함수를 제공할 수 있음

아이콘은 글리프(Glyph) 기반의 아이콘을 사용할 수 있으며 Icons 클래스를 통해 플러터에서 제공하는 기본 아이콘들을 사용할 수 있음

```
IconButton(
 onPressed: () {},
 icon: Icon(
   // 플러터에서 기본으로 제공하는 아이콘
   // 제공되는 아이콘 목록은 아래 링크에서 확인
   // https://fonts.google.com/icons
   Icons.home,
 ),
)
```

- GestureDetector 위젯

앱은 모든 입력을 손가락으로 하므로 여러 가지 입력을 인지하는 위젯이 필요

```
GestureDetector(
 // 한 번 탭했을 때 실행할 함수
 onTap: (){
   // 출력 결과는 안드로이드 스튜디오의 [Run] 탭에서 확인 가능
   print('on tap');
 },
 // 두 번 탭했을 때 실행할 함수
 onDoubleTap: ( ){
   print('on double tap');
 },
 // 길게 눌렀을 때 실행할 함수
```

```
  onLongPress: (){
    print('on long press');
  },
  // Gesture를 적용할 위젯
  child: Container(
    decoration: BoxDecoration(
      color: Colors.red,
    ),
    width: 100.0,
    height: 100.0,
  ),
)
```

매개변수	설명
onTap	한 번 탭했을 때 실행되는 함수를 입력할 수 있습니다.
onDoubleTap	두 번 연속으로 탭했을 때 실행되는 함수를 입력할 수 있습니다.
onLongPress	길게 누르기가 인식됐을 때 실행되는 함수를 입력할 수 있습니다.
onPanStart	수평 또는 수직으로 드래그가 시작됐을 때 실행되는 함수를 입력할 수 있습니다.
onPanUpdate	수평 또는 수직으로 드래그를 하는 동안 드래그 위치가 업데이트될 때마다 실행되는 함수를 입력할 수 있습니다.
onPanEnd	수평 또는 수직으로 드래그가 끝났을 때 실행되는 함수를 입력할 수 있습니다.
onHorizontalDragStart	수평으로 드래그를 시작했을 때 실행되는 함수를 입력할 수 있습니다.
onHorizontalDragUpdate	수평으로 드래그를 하는 동안 드래그 위치가 업데이트될 때마다 실행되는 함수를 입력할 수 있습니다.
onHorizontalDragEnd	수평으로 드래그가 끝났을 때 실행되는 함수를 입력할 수 있습니다.
onVerticalDragStart	수직으로 드래그를 시작했을 때 실행되는 함수를 입력할 수 있습니다.
onVerticalDragUpdate	수직으로 드래그를 하는 동안 드래그 위치가 업데이트될 때마다 실행되는 함수를 입력할 수 있습니다.
onVerticalDragEnd	수직으로 드래그가 끝났을 때 실행되는 함수를 입력할 수 있습니다.
onScaleStart	확대가 시작됐을 때 실행되는 함수를 입력할 수 있습니다.
onScaleUpdate	확대가 진행되는 동안 확대가 업데이트될 때마다 실행되는 함수를 입력할 수 있습니다.
onScaleEnd	확대가 끝났을 때 실행되는 함수를 입력할 수 있습니다.

GestureDetector 위젯에서 제공하는 제스처 매개변수

- FloatingActionButton 위젯

Material Design에서 추구하는 버튼 형태

안드로이드 앱들을 사용하다 보면 화면의 오른쪽 아래에 동그란 플로팅 작업 버튼을 쉽게 볼 수 있으며FloatingActionButton과 Scaffold를 같이 사용하면 특별한 어려움 없이 해당 형태의 디자인을 구현할 수 있음

```
import 'package:flutter/material.dart';

void main() {
 runApp(FloatingActionButtonExample());
}

class FloatingActionButtonExample extends Statele
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        floatingActionButton: FloatingActionButto
          // 클릭했을 때 실행할 함수
          onPressed: () {},
          child: Text('클릭'),
        ),
        body: Container(),
      ),
    );
  }
}
```

디자인 관련 위젯

- Container 위젯

다른 위젯을 담는 데 사용

위젯의 너비와 높이를 지정하거나, 배경이나 테두리를 추가할 때 주로 사용

```
Container(
 // 스타일 작용
 decoration: BoxDecoration(
```

```
    // 배경 색깔 적용
    color: Colors.red,
    // 테두리 적용
    border: Border.all(
      // 테두리 굵기
      width: 16.0,
                // 테두리 색상
                color: Colors.black,
              ),
              // 모서리 둥글게 만들기
              borderRadius: BorderRadius.circular(
                16.0,
              ),
            ),
            // 높이
            height: 200.0,
            // 너비
            width: 100.0,
          )
```

- SizedBox 위젯

일반적으로 일정 크기의 공간을 공백으로 두고 싶을 때 사용

Container 위젯을 사용해도 공백을 만들 수 있지만 SizedBox는 const 생성자를 사용했을 때 퍼포먼스에서 이점

```
SizedBox(
 // 높이 지정
 height: 200.0,

 // 너비 지정
 width: 200.0,

 // SizedBox는 색상이 없으므로 크기를 확인하는
 // 용도로 Container 추가
 child: Container(
   color: Colors.red,
```

```
  ),
 )
```

- Padding 위젯

child 위젯에 여백을 제공할 때 사용

Padding 위젯의 상위 위젯과 하위 위젯 사이의 여백 가능

adding 위젯의 padding 매개변수는 EdgeInsets라는 값을 입력

child 매개변수에 Padding을 적용하고 싶은 위젯을 입력

```
Container(
 color: Colors.blue,
 child: Padding(

   // 상하, 좌우로 모두 16픽셀만큼 패딩을 적용합니다.
   padding: EdgeInsets.all(
     16.0,
   ),
   child: Container(
     color: Colors.red,
     width: 50.0,
     height: 50.0,
   ),
 ),
)
```

+) 자주 사용하지는 않지만 위젯의 바깥에 간격을 추가해주는 마진(Magin)이라는 기능

다만 마진은 따로 위젯이 존재하지 않고 Container 위젯에 추가

```
// 최상위 검정 컨테이너 (margin이 적용되는 대상)
Container(
 color: Colors.black,

 // 중간 파란 컨테이너
```

```
  child: Container(
    color: Colors.blue,

    // 마진 적용 위치
    margin: EdgeInsets.all(16.0),

    // 패딩 적용
    child: Padding(
      padding: EdgeInsets.all(16.0),

      // 패딩이 적용된 빨간 컨테이너
      child: Container(
        color: Colors.red,
        width: 50,
        height: 50,
      ),
    ),
  ),
)
```

생성자	설명
EdgeInsets.all(16.0)	상하좌우로 매개변수에 입력된 패딩을 균등하게 적용합니다.
EdgeInsets.symmetric(horizontal: 16.0, vertical: 16.0)	가로와 세로 패딩을 따로 적용합니다. Horizontal 매개변수에 가로로 적용할 패딩을 입력하고 Vertical 매개변수에 세로로 적용할 패딩을 입력합니다.
EdgeInsets.only(top: 16.0, bottom: 16.0, left: 16.0, right: 16.0)	위아래, 좌우 패딩을 따로 적용합니다. top, bottom, left, right 매개변수에 각각 위아래, 좌우 패딩을 입력할 수 있습니다.
EdgeInsets.fromLTRB(16.0, 16.0, 16.0, 16.0)	위아래, 좌우 패딩을 따로 적용합니다. 포지셔널 파라미터를 좌, 위, 우, 아래 순서로 입력해주면 됩니다.

EdgeInsets 클래스 생성자 종류 정리

- SafeArea 위젯

플러터는 가용되는 화면을 모두 사용하기 때문에 노치가 있는 핸드폰에서 노치에 위젯들이 가릴 수 있음

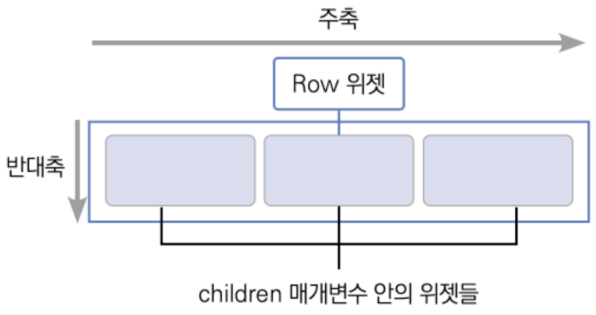
이를 사용하면 따로 기기별로 예외 처리를 하지 않고도 안전한(Safe) 화면에서만 위젯을 그릴 수 있음

```
SafeArea(
 // 원하는 부위만 따로 적용할 수도 있습니다.
 // true는 적용 false는 미적용
 top: true,
 bottom: true,
 left: true,
 right: true,
 child: Container(
   color: Colors.red,
   height: 300.0,
   width: 300.0,
 ),
),
```

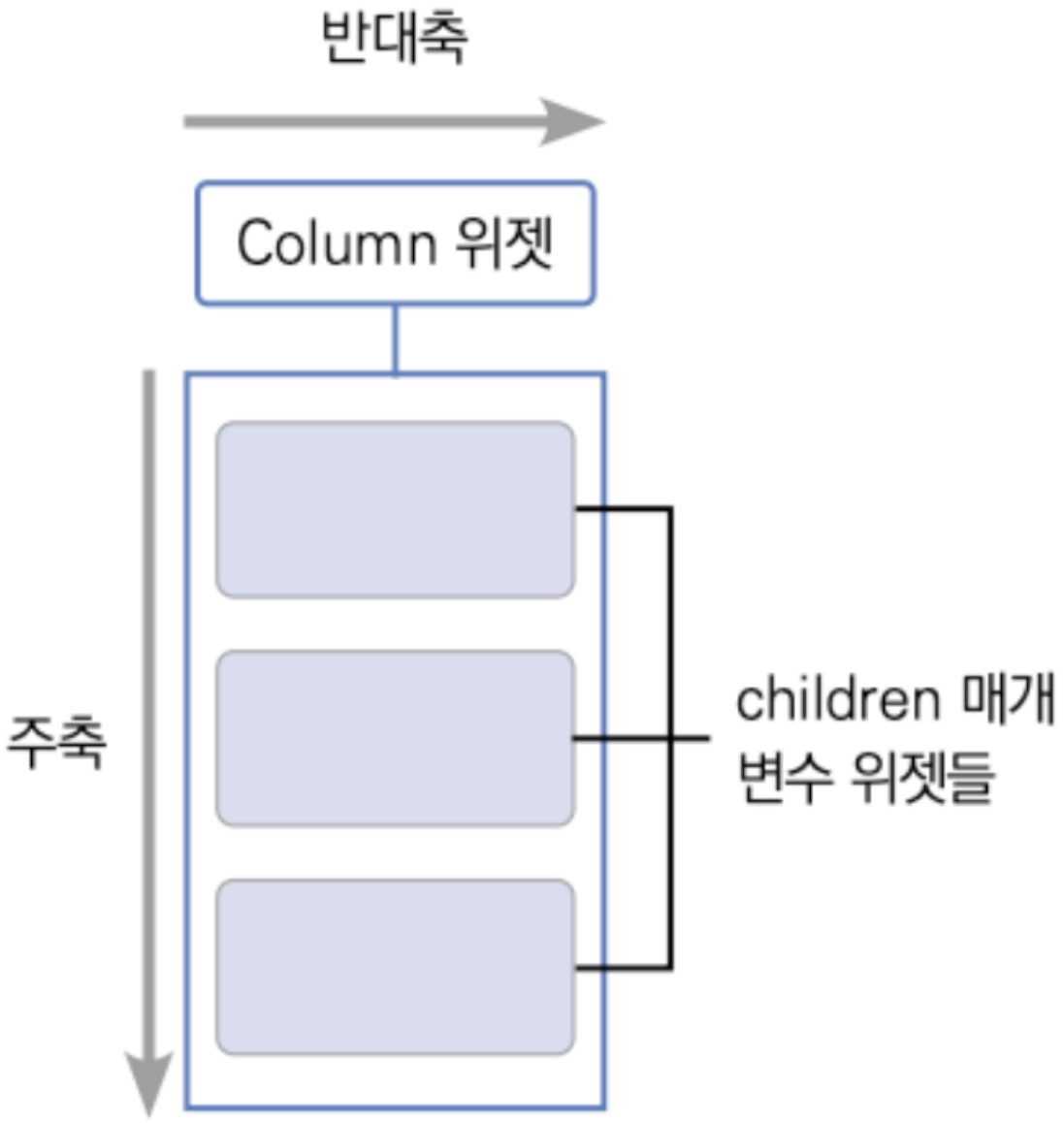
배치 관련 위젯

Row와 Column에는 주축(Main Axis)과 반대축(Cross Axis)이라는 개념이 존재하는데 Row는 가로가 주축, 세로가 반대축이 되고 Column의 경우 반대

주축과 반대축을 어떻게 조합하냐에 따라 Row와 Column 위젯을 이용해서 다양하게 배치 가능



Row 위젯의 주축과 반대축



Column 위젯의 주축과 반대축

- Row 위젯

Row는 가로로 위젯을 배치하는 데 사용

하나의 child 위젯을 입력받는 위젯들과 다르게 여러 개의 child 위젯을 입력받을 수 있는 children 매개변수를 노출

```
import 'package:flutter/material.dart';

void main() {
 runApp(RowWidgetExample());
}
```

```
class RowWidgetExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: SizedBox(
          height: double.infinity,
          child: Row(
            // 주축 정렬 지정
            mainAxisAlignment: MainAxisAlignment.start,
            // 반대축 정렬 지정
            crossAxisAlignment: CrossAxisAlignment.cent
            // 넣고 싶은 위젯 입력
            children: [
              Container(
                height: 50.0,
                width: 50.0,
                color: Colors.red,
              ),
              // SizedBox는 일반적으로 공백을
              // 생성할 때 사용
              const SizedBox(width: 12.0),
              Container(
                height: 50.0,
                width: 50.0,
                color: Colors.green,
              ),
              const SizedBox(width: 12.0),
              Container(
                height: 50.0,
                width: 50.0,
                color: Colors.blue,
              ),
            ],
          ),
        ),
      ),
    );
```

```
    }
  }
```

옵션	설명	예제
MainAxisAlignment.start	시작에 정렬	
MainAxisAlignment.center	중앙에 정렬	
MainAxisAlignment.end	끝에 정렬	
MainAxisAlignment.spaceBetween	자식 위젯의 간격을 균등하게 정렬	
MainAxisAlignment.spaceAround	자식 위젯의 간격을 균등하게 배정하고 왼쪽 끝과 오른쪽 끝을 위젯 사이 거리의 반만큼 배정해 정렬	
MainAxisAlignment.spaceEvenly	자식 위젯의 간격을 균등하게 배치하고 왼쪽 끝과 오른쪽 끝도 균등하게 배치	

MainAxisAlignment 옵션(예제는 모두 CrossAxisAlignment.center 기준)

옵션	설명	예제
CrossAxisAlignment.start	시작에 정렬	
CrossAxlisAlignment.center	중앙에 정렬	
CrossAxisAlignment.end	끝에 정렬	
CrossAxisAlignment.stretch	반대축 최대한으로 늘려서 정렬	

CrossAxisAlignment 옵션(예제는 모두 MainAxisAlignment.center 기준)

- Column 위젯

Row 위젯과 완전히 같은 매개변수들을 노출하나, 주축과 반대축이 Row와 반대

```
import 'package:flutter/material.dart';

void main() {
 runApp(ColumnWidgetExample());
}

class ColumnWidgetExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: SizedBox(
```

```
            width: double.infinity,
            child: Column(
              // 주축 정렬 지정
              mainAxisAlignment: MainAxisAlignment.start,
              // 반대축 정렬 지정
              crossAxisAlignment: CrossAxisAlignment.star
              // 넣고 싶은 위젯 입력
              children: [
                Container(
                  height: 50.0,
                  width: 50.0,
                  color: Colors.red,
                ),
                // SizedBox는 일반적으로 공백을 생성할 때 사용
                const SizedBox(width: 12.0),
                Container(
                  height: 50.0,
                  width: 50.0,
                  color: Colors.green,
                ),
                const SizedBox(width: 12.0),
                Container(
                  height: 50.0,
                  width: 50.0,
                  color: Colors.blue,
                ),
              ],
            ),
          ),
        ),
      );
  }
}
```

옵션	설명	예제
MainAxisAlignment.start	시작에 정렬	
MainAxisAlignment.center	중앙에 정렬	
MainAxisAlignment.end	끝에 정렬	
MainAxisAlignment. spaceBetween	자식 위젯의 간격을 균등하게 정렬	
MainAxisAlignment.spaceAround	자식 위젯의 간격을 균등하게 배정하고 왼쪽 끝과 오른쪽 끝을 위젯 사이 거리의 반만큼 배정해 정렬	
MainAxisAlignment.spaceEvenly	자식 위젯의 간격을 균등하게 배치하고 왼쪽 끝과 오른쪽 끝도 동일하게 균등하게 배치	

MainAxisAlignment 옵션(예제는 모두 CrossAxisAlignment.center 기준)

옵션		예제
CrossAxisAlignment.start	시작에 정렬	
CrossAxlisAlignment.center	중앙에 정렬	
CrossAxisAlignment.end	끝에 정렬	
CrossAxisAlignment.stretch	반대축 최대한으로 늘려서 정렬	

CrossAxisAlignment 옵션(예제는 모두 MainAxisAlignment.center 기준)

- Flexible 위젯

Flexible 위젯은 Row나 Column에서 사용하는 위젯

Flexible 위젯을 Column과 Row에서 사용하면 Flexible에 제공된 child가 크기를 최소한으로 차지

추가적으로 flex 매개변수를 이용해 각 Flexible 위젯이 얼만큼의 비율로 공간을 차지할지 지정 가능

```
Column(
 children: [
   Flexible(
     // flex는 남은 공간을 차지할 비율을 의미합니다.
     // flex값을 값을 제공하지 않으면 기본값은 1입니다.
     flex: 1,
```

```
      // 파란색 Container
      child: Container(
        color: Colors.blue,
      ),
    ),
    Flexible(
      flex: 1,

      // 빨간색 Container
      child: Container(
        color: Colors.red,
      ),
    )
  ],
),
```

- Expanded 위젯

Flexible 위젯을 상속하는 위젯

Column과 Row에서 Expanded를 사용하면 위젯이 남아 있는 공간을 최대한으로 차지

- 원리 : fit 매개변수에 FlexFit.tight 또는 FlexFit.loose를 입력

  FlexFit.loose는 자식 위젯이 필요한 만큼의 공간만 차지

  FlexFit.tight는 자식 위젯이 차지하는 공간과 관계없이 남은 공간을 모두 차지

Expanded 위젯은 Flexible 위젯의 fit 매개변수에 FlexFit.tight를 기본으로 제공해준 위젯으로 Flexible 위젯과 다르게 남는 공간을 최대한으로 차지

```
Column(
 children: [
   // 파란색 Container
   Expanded(
     child: Container(
       color: Colors.blue,
     ),
   ),
   // 빨간색 Container
   Expanded(
```

```
      child: Container(
        color: Colors.red,
      ),
    )
  ],
)
```

- Stack 위젯

Stack 위젯은 위젯을 겹치는 기능을 제공

플러터의 그래픽 엔진인 스키아 엔진은 2D 엔진이기 때문에 겹친 두께를 표현하지 못하지만 Stack을 사용하면 위젯 위에 위젯을 올린 듯한 효과 연출 가능

```
Stack(
 children: [
   // 빨간색 Container
   Container(
     height: 300.0,
     width: 300.0,
     color: Colors.red,
   ),

   // 노란색 Container
   Container(
     height: 250.0,
     width: 250.0,
     color: Colors.yellow,
   ),

   // 파란색 Container
   Container(
     height: 200.0,
     width: 200.0,
     color: Colors.blue,
   ),
 ],
),
```

▼ 상태관리

> 위젯의 상태 = 데이터

하지만 위젯의 모든 데이터를 상태라고 하지는 않음

상태란 위젯에서 다양한 이유로 변경되는 데이터를 의미함

따라서 위젯의 상태 관리란 이런 상태 데이터를 관리하는 방법

위젯의 상태 관리 기본 개념

- 위젯 자체의 상태를 이용
- 상위 위젯의 상태를 이용
- 위젯 자체의 상태와 상위 위젯의 상태를 함께 이용

1. 위젯 자체의 상태를 이용

   상태 데이터를 해당 위젯에서만 사용하는 경우

   상태 데이터가 변경되면 화면을 변경해야 하지만 해당 위젯에만 필요하며, 화면을 함께 구성하는 다른 위젯에서는 이용하지 않는 데이터 → 해당 위젯에서 선언하고 관리(StatefulWidget)

2. 상위 위젯의 상태를 이용

   상태 데이터를 한 위젯에서만 이용하지 않고 위젯 트리의 다른 위젯이 함께 이용한다는 의미

   상위 위젯에서 상태 관리하는 경우

   예를 들어 아이콘을 누르면 좋아요 수가 늘어난다고 할 때, 좋아요 상태 데이터와 숫자를 상위 위젯에서 관리하여 이를 하위 위젯에게 매개변수로 전달

   조상, 하위 위젯의 상태 얻기

   위젯 트리의 계층이 복잡할 때 위의 방법은 비효율적

   - 조상 위젯의 상태

     findAncestorStateOfType() 함수로 조상 위젯의 상태 객체 얻기 → BulidContext 클래스에서 제공하는 함수

얻고자 하는 조상의 상태 클래스를 제네릭으로 지정하면 해당 타입의 상태를 찾아 전달

```
ParentWidgetState? state = context.findAncestorSt
```

- 하위 위젯의 상태

  때로는 상위 위젯에서 하위 위젯의 상태 객체를 직접 얻어야 할 수도 있음

  하위 위젯을 StatefulWidget으로 선언했다면 상태 객체를 가짐

  > 하위 위젯의 객체는 상위 위젯에서 생성하지만, 하위 위젯의 상태 객체는 상위 위젯에서 생성한 객체가 아니다!

  상위 위젯에서 하위 위젯의 상태 객체를 얻으려면 하위 위젯을 생성할 때 키를 지정하고, 이 키의 currentState 속성 이용

```
class ParentWidgetState extends State<ParentWidge
    GlobalKey<ChildWidgetState> childKey = Global
    int childCount = 0;

    void getChildData() {
        ChildWidgetState? childState = childKey.c
        setState(() {
            childCount = childState?.childCount ?
        });
    }

    @override
    Widget build(BuildContext context) {
        return ChildWidget(key: childKey);
    }
}

class ChildWidget extends StatefulWidget {
    ChildWidget({Key? key}):super(key: key);
    (생략)
}
class ChildWidgetState extends State<ChildWidget>
```

```
    int childCount = 0;
    (생략)
}
```

▼ 내비게이션

라우트 이해하기

> 화면 전환을 제공하려면 Route와 Navigator를 사용한다.

- **Route**는 화면을 지칭하는 객체
- **Navigator**는 **Route** 객체로 화면을 전환

화면 전환하기

- 다음 화면으로 전환: Navigator.push()
- 이전 화면으로 전환: **Navigator.pop()**

라우트 이름으로 화면 전환하기

push함수로 계속 전환하여 화면이 많아지면 비효율적

이런 상황에서 라우트 이름으로 화면을 전환하면 효율적

```
MaterialApp(
      initialRoute: '/one',
      routes: {
        '/one': (context) => const OneScreen(),
        '/two': (context) => const SecondScreen(),
        '/three': (context) => const ThirdScreen(),
        '/four': (context) => const FourthScreen(),
      },
    );
```

```
ElevatedButton(
                onPressed: () {
                  Navigator.pushNamed(context, '/two');
```

```
                },
                child: const Text(
                  'Go Second',
                ),
              ),
```

화면 전환할 때 데이터 전달하기

push(), pushNamed(), pop() 함수를 이용해 화면 전환

- **push( )**: push( ) 함수의 두 번째 매개변수에 라우트를 직접 준비

```
Navigator.push(context, MaterialPageRoute(builder: (con
```

- **pushNamed( )**: **arguments**라는 매개변수 이용

```
ElevatedButton(
    onPressed: () async {
      final result =
          await Navigator.pushNamed(context, '/two', ar
        "arg1": 1,
        "arg2": "hello",
        "arg3": User('kkang', 'seoul'),
      });
      print('result: ${(result as User).name}');
    },
    child: const Text(
      'Go Second',
    ),
  ),
```

- 데이터를 받는 곳에서는 JSON 타입의 데이터를 **Map** 객체로 얻어서 사용

```
 Map<String, Object> args =
        ModalRoute.of(context)?.settings.arguments as
Map<String, Object>;
```

- **pop( )**: 두 번째 매개변수 이용

```
ElevatedButton(
    onPressed: () {
      Navigator.pop(
        context,
        User('kim', 'busan'),
      );
    },
)
```

pop( ) 함수로 전달한 데이터는 화면을 전환할 때 사용했던 push( ) 나 pushNamed( ) 함수의 반환값으로 받을 수 있음

routes와 onGenerateRoute의 차이점

같은 라우트 이름이라도 상황에 따라 다른 화면이 나오게 하고 싶을 때, 데이터를 분석해 동적인 라우트가 필요할 때 onGenerateRoute속성을 사용