CLASS

```
class Player{
        String name = 'nico';
        int xp = 1500;

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
         var player = Player();
         player.sayHello(); // 출력된다.
}
```

Class 는 반드시 자료형을 선언해줘야 한다.

1. var 사용 불가
2. this 사용 자제
3. final 사용 가능

- Constructors

```
class Player{
        late final String name;
        late int xp;

        Player(String name, int xp){
                this.name = name;
                this.xp = xp;
        }

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
         var player = Player('nico', 1500);
         player.sayHello(); // 출력된다.
```

```
        var player2 = Player('홍웅', 2000);
}
```

위와 같이 작성해도 되지만 더 짧게 아래와 같이 작성할 수 있다.

```
class Player{
        final String name;
        int xp;

        Player(this.name, this.xp);

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
        var player = Player('nico', 1500);
        player.sayHello(); // 출력된다.
        var player2 = Player('홍웅', 2000);
}
```

main 에서 호출할 때 순서가 매우 중요하다.

- Named Constructor Parameters

```
class Player{
        final String name;
        int xp;

        Player(required this.name,
                                        required this.xp,
                                        required this.team,
                                        required this.age);

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
        var player = Player(
```

```
                        name: 'nico',
                        xp: 1200,
                        team: 'blue',
                        age: 21,
                );
        player.sayHello(); // 출력된다.
        var player2 = Player('홍웅', 2000);
}
```

직관적으로 바꿔 호출할 수 있다. required 를 붙여주지 않으면 null 이 될 수 있어 에러가 남.

- named Constructors

```
class Player{
        final String name;
        int xp, age;
        String team;

        Player(required this.name,
                                        required this.xp,
                                        required this.team,
                                        required this.age);

Player.createBluePlayer({required String name, required int age}) :
this.age = age, // parameter 로 받는 age
this.name = name, // parameter 로 받는 name
this.team = 'blue', // default value
this.xp = 0; // default value

Player.createRedPlayer(String name, int age) :
this.age = age,
this.name = name,
this.team = 'red',
this.xp = 0;

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
```

```
        var player = Player.createBluePlayer(
                name: 'nico',
                age: 20;
        )

        var player2 = Player.createRedPlayer('nico', 20);
}
```

Player.createBluePlayer 이런 식으로 named constructor 를 만들 수 있다.

즉, 이건 Player 를 초기화하는 method 다.

콜론을 이용하여 Player 클래스를 초기화한다. 콜론(:)을 넣음으로써 dart 에게 여기서 Player 객체를 초기화하겠다고 한 것.

위에서 만든 두 함수는 같은 동작을 하지만 직관성이 다르다.

- Recap

```
class Player{
        final String name;
        int xp, age;
        String team;

        Player.fromJson(Map<String, dynamic> playerJson) :
                name = playerJson['name'],
                xp = playerJson['xp'],
                team = playerJson['team'];

        void sayHello(){
                print("Hi my name is $name");
        }
}

void main(){
        var apiData = [
                {
                        "name": "nico",
                        "team": "red",
                        "xp": 0,
                },
                {
                        "name": "홍웅",
                        "team": "red",
```

```
                        "xp": 0,
                },
                {
                        "name": "경희",
                        "team": "red",
                        "xp": 0,
                },
        ];

        apiData.forEach((playerJson) {
                var player = Player.fromJson(playerJson);
                player.sayHello();
}
```

- Cascade Notation

```
class Player{
        String name;
        int xp;
        String team;

        Player({required this.name, required this.xp, required this.team});

        void sayHello(){
                print("hi my name is $name");
        }
}

void main(){
        var nico = Player(name: 'nico', xp:1200, team:'red');
        nico.name = 'las';
        nico.xp = 120000;
        nico.team = 'blue';
        ..sayHello();

        var nico = Player(name: 'nico', xp:1200, team:'red')
        ..name = 'las'
        ..xp = 120000
        ..team = 'blue';
        ..sayHello();
}
```

main 의 위와 아래는 같은 코드이다.

..은 바로 위의 class 를 가르키기 때문에 nico 를 가르키고 있다.

- Enums

```
enum Team { red, blue }
enum XPLevel {pro, medium, beginner}
class Player{
        String name;
        int xp;
        Team team;

        Player({required this.name, required this.xp, required this.team});

        void sayHello(){
                print("hi my name is $name");
        }
}

void main(){
        var nico = Player(name: 'nico', xp:1200, team:'Team.red');
        nico.name = 'las';
        nico.xp = 120000;
        nico.team = 'Team.blue';
        ..sayHello();

        var nico = Player(name: 'nico', xp:1200, team:'Team.red')
        ..name = 'las'
        ..xp = 120000
        ..team = 'Team.blue';
        ..sayHello();
}
```

Enum 은 오타를 방지할 수 있다. enum Team 으로 만들고 Player class 안의 team 의 자료형을 Team 으로 만들면 red, blue 두 가지 선택지 중에서 고를 수 있다.

- Abstract Classes

```
abstract class Human{
```

```
        void walk();
}

cl|ass Player extends Human{
        void walk(){
                print('im walk');
        }
}
```

1. 추상화 클래스로는 객체를 생성할 수 없다.

추상화 클래스는 다른 클래스들이 구현해야하는 청사진 같은 것.

상속 받으면 메서드를 무조건 구현해줘야 함. → 구현의 강제화

- Inheritance

```
class Human{
        final String name;
        Human({required this.name});
        void sayHello(){
                print("Hi my name is $name");
        }
}

enum Team = {blue, red};
class Player extends Human{
        final Team team;

        Player({
                        required this.team,
                        required String name,
        }) : super(name: name); // 부모의 생성자

        @override
        void sayHello(){
                super.sayHello();
                print('and i play for ${team}');
        }
}

void main(){
```

```
        var player = Player(team: Team.red, name:'nico');
}
```

super()를 이용하여 부모와 소통가능.

- Mixin : 생성자가 없는 클래스

```
class Strong{
        final double streghtLevel = 1500.99;
}

class QuickRunner {
        void runQuick(){
                print('run');
        }
}

enum Team { blue, red }

class Player with Strong, QuickRunner {
        final Team team;
        Player({
                required this.team,
        });
}
```

with 을 사용하여 Quick 과 Strong 클래스에 있는 프로퍼티와 메소드를 Player 에 담아줄 수 있다.

단순히 Mixin 내부의 프로퍼티와 메소드들을 가져오는 것

- Conclusions

Flutter(프레임 워크)를 배우자 !