Classes

- 플러터는 모든게 class라고 할 수 있을 만큼 중요

```
// 클래스 활용 기본 예제
class Player {
  // 클래스 안 변수는 타입을 꼭 명시해야 한다
  // var 사용 ㄴㄴ
  final String name = 'nico';
  int xp = 1500;

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


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

Constructors

```
class Player {
  late final String name;
  late int xp;
  // name과 xp의 정보를 나중에 받아오기 때문에 late를 사용한다.

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

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

// argument로 name과 xp를 전달하여 새 player를 생성
void main() {
  var player = Player('nico', 1500);
  player.sayHello();
  var player2 = Player('lynn', 300);
  player2.sayHello();
}
```

- 위의 코드에서 name과 xp의 타입을 선언을 해뒀는데 다시 타입을 선언하는 부분이 반복적이다.
- 아래에 더 좋은 방법이 있다.

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

  Player(this.name, this.xp);
  // 클래스 안에 선언해뒀던 타입을 그대로 가져다 쓰는 것

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

// argument로 name과 xp를 전달하여 새 player를 생성
void main() {
  var player = Player('nico', 1500);
  player.sayHello();
  var player2 = Player('lynn', 300);
  player2.sayHello();
}
```

Named Constructor Parameters

- 함수를 쓸 때와 마찬가지로 argument가 많아지면 혼란스러워진다.
- named argument를 가진 constructor로 바꿔서 해결 해보자

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

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

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

// argument로 name과 xp를 전달하여 새 player를 생성
void main() {
  var player = Player(
    name: 'nico',
```

```
    xp: 1500,
    team: 'blue',
    age: 32,
  );
  player.sayHello();
  var player2 = Player(
    name: 'lynn',
    xp: 1230,
    team: 'red',
    age: 25,);
  player2.sayHello();
}
```

- parameter에 중괄호를 씌워준다
- dart가 null을 우려하여 오류가 뜨는데
- required를 적어서 해결

Named Constructors

- 두 개의 constructor를 만드는데
  - xp의 기본값을 0 team의 값을 blue로 가진 player와
  - xp의 기본값을 0 team의 값을 red로 가진 player
- 사용자는 name과 age를 보내도록 하고 싶다.

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

  Player({required this.name,
          required this.xp,
          required this.team,
          required this.age,
         }); // 내가 클래스를 호출할 때 마다 기본으로 호출되는 constructor

 //////////////////////////Named Constructors/////////////////////////////////////
  Player.createBluePlayer({
    required String name,
    required int age
    // named parameter는 기본적으로 required 속성이 없어서 적어줘야 한다.
    }) :  this.age = age,
          this.name = name,
          this.team = 'blue',
          this.xp = 0;

  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");
  }
}

// argument로 name과 xp를 전달하여 새 player를 생성
void main() {
  var bluePlayer = Player.createBluePlayer(
    name: 'nico',
    age: 21,
  );
  var redPlayer = Player.createRedPlayer('lynn',30);
}
```

- dart만의 특별한 syntax인 콜론 (:) 을 사용해 property를 초기화 시킨다.

- createBluePlayer 는 named parameter를 사용하였고
- createRedPlayer는 positional parameter를 사용했다.
  - 상황에 맞게 사용하면 될 듯

recap

- api를 통해 json 형태의 데이터를 받아왔다고 가정하고
- named constructor와 forEach 함수를 통해 결과를 냈다.

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

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

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

}



void main() {
  var apiData = [
    {
```

```
      "name": "nico",
      "team": "red",
      "xp": 0,
    },
    {
      "name": "lynn",
      "team": "red",
      "xp": 0,
    },
    {
      "name": "dal",
      "team": "red",
      "xp": 0,
    },
  ];

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

// Hello my name is nico
// Hello my name is lynn
// Hello my name is dal
```

Cascade Notation

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

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

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



void main() {
  var nico = Player(name: 'nico', xp: 12000, team: 'red')
////Cascade Notation//////////////////////////////////////////
    ..name = 'las'
    ..xp = 1200000
    ..team = 'blue';
////////////////////////////////////////////////////////////
}
```

- ..을 통해 앞에 선언했던 클래스를 가리키게 함

Enum

- 오타 같은 실수를 하지 않게끔 도와준다.

```
enum Team { red, blue }
// enum을 통해 Team의 선택지를 red, blue로 제한함

class Player {
  String name;
  int xp;
  Team team; // team은 이제 String 타입이 아니라 Team 중 하나가 들어가야 하는 것

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

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



void main() {
  var nico = Player(name: 'nico', xp: 12000, team: Team.red) // team은 Team에서 골라야 함
    ..name = 'las'
    ..xp = 1200000
    ..team = Team.blue
    ..sayHello();
}
```

Abstract Classes

- 추상화 클래스는 다른 클래스들이 직접 구현해야 하는 메소드들을
- 모아 놓은 일종의 청사진

```
////////////////////////////////
abstract class Human {
  void walk();
} // 해당 class의 구조를 사용하면 walk에 대해 구현해야 한다
////////////////////////////////

enum Team { red, blue }
```

```
class Player extends Human{ // extends 키워드를 통해 abstract class를 상속
  String name;
  int xp;
  Team team;

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

  void walk(){
    print("im walking");
  }

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

class Coach extends Human{
  void walk(){
    print("im not walking");
  }
}



void main() {
  var nico = Player(name: 'nico', xp: 12000, team: Team.red)
    ..name = 'las'
    ..xp = 1200000
    ..team = Team.blue
    ..sayHello();
}
```

Inheritance

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

enum Team{ red, blue }

class Player extends Human {
  final Team team;

  Player({
    required this.team,
    required String name // Human이 name을 받기 때문에 작성
```

```
  })  : super(name); // 받은 name을 Human에게 전달
// super를 통해 human에 필요한 name의 값을 player 선언할 때 받아서 올려준다.

}

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

Mixins

- 생성자(constructor)가 없는 클래스여야 하는 것이 조건 !!
- extends 대신 with을 사용함
- 부모 클래스로 두는 것이 아니고 mixin 내부의 프로퍼티와 메소드를 가져오는 것

```
mixin Strong {
  final double strengthLevel = 1500.99;
}

mixin QuickRunner {
  void runQuick() {
    print("run!!!!!!!!!!!!!");
  }
}

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

enum Team{ red, blue }

class Player with Strong, QuickRunner {
  final Team team;

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

void main() {
  var player = Player(team: Team.red,);
}
```

- 버전이 올라가면서 mixin 클래스는 mixin 이나 mixin class로 선언해야 함