자바에서 다트로 넘어가기

플러터(Flutter)로 개발을 하면서 다트를 처음 써보게 되었는데, 하루도 안되어 적응을 하였다.

그만큼 자바와 다트는 많은 공통점이 있다.

다트 공식 웹사이트에선 **familiar to many existing developers** 라고 말하는 데 과장된 이야기가 아니다.

대부분의 문법이 큰 차이가 없으며, 객체 지향 프로그래밍을 접해봤다면 빠르게 다트를 배울 수 있다.

다트와 자바엔 어떤 공통점과 차이가 있는지 알아보도록 하자.

변수 - 타입 지정, 타입 추론

다트의 변수 선언은 자바랑 무척 유사하지만 차이가 있다. 자바는 타입을 항상 지정해야된다.

다트는 타입을 정할 수도 있고, 안 정할 수도 있다. 타입을 정하지 않으면 컴파일할때 타입이 정해진다.(타입 추론)

다만 한번 정한 타입을 나중에 바꿀 수는 없다.(타입 세이프함 type-safe)

아래 코드를 보자. 기본적인 숫자와 문자에 관한 자료형들이다.

JAVA - 타입 지정

```
int num = 10;
float height = 175.3;
long area = 2500L;
String name = "Mike";
```

전반적으로 비슷하지만 동적 타입이 있단 점에서 다르다.

DART - 타입 지정 or 타입 추론

```
int num = 10;
double height = 175.3;
String name = "Mike";
var world = "Hello world"; // 동적 타입, 타입 추론
var age = 78;
```

밑에 처럼 타입을 중간에 바꾸는 건 안된다.

~~var abc = "Hello";~~

~~abc = 10;~~

~~abc = true;~~

배열 대신에 리스트

자바에선 배열과 리스트를 구분하지만 다트는 리스트만 사용하기에 좀 더 편하게 데이터를 다룰 수 있다.

JAVA

```
int[] numbers = [10, 20, 30];
String[] countries = {"USA", "JAPAN", "KOREA"};
ArrayList<Integer> lists = new ArrayList<>();
lists.add(100);
lists.add(200);
lists.add(300);
```

리스트에 서로 다른 자료형도 섞을 수 있다.

DART

```
var numbers = [10, 20, 30];
var countries = ["USA", "JAPAN", "KOREA", 10];
print(countries[0]);
```

함수

웹을 위해 만들어진 언어라 그런지, 자바스크립트랑 비슷한 측면이 많다. 타입을 명시해도 되고, 안 해도 된다.

JAVA

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

DART - 타입 추론.

```
combine(a,b) {
  return a + b;
}

var result = combine(10, 20);
print(result);

var word = combine("hello", "world");
print(word);
```

DART - 타입 지정.

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

int sum = add(15, 25);
print(sum);
```

Effective Dart에서는 타입을 지정하는 걸 권장하고 있다. 사실 플러터로 다트를 배우다 보니, 타입 지정을 안해도 되는지 몰랐었다.

If, For, While, Case

자바와 다트의 차이가 없는거나 마찬가지라 생략한다.

Class - 이름 있는 생성자(Named Constructor)

자바와 다트의 생성자를 같이 보도록 하자. 크게 다르지 않다.

JAVA

```
class Point {
  int x;
  int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

DART

```
class Point {
  num x;
  num y;

  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }

  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

1가지를 제외하고는 큰 차이가 없다. **Point.origin()** 가 있냐 없냐의 차이다. **Point.origin()** 은 대체 어떤 역할을 하는 걸까? **Point.origin()** 은 이름 있는 생성자(Named Constructor)로 x,y를 0으로 초기화 하는 역할을 한다.

```
Point p = Point(10, 30);
print(p.x); // 10

var origin = Point.origin();
print(origin.x); // 0
```

이처럼 이름 있는 생성자(Name Constructor)는 여러 생성자를 만들거나 생성자 내에서 값 체크, 파싱 등 각종 작업을 할 때 쓰인다.

(다트 공식 사이트의 Named Constructor 정의 - Use a named constructor to implement multiple constructors for a class or to provide extra clarity)

이제 json을 파싱하는 생성자를 만들어보겠다.

DART

```
class Point {
  num x;
  num y;

  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }

  Point.origin() {
    x = 0;
    y = 0;
  }

  Point.fromJson(Map<String, num> json)
    : x = json['x'],
      y = json['y'] {
    print('In Point.fromJson(): ($x, $y)');
  }
}
```

이름 있는 생성자를 만들고 실행을 해보자. 객체를 만들때 json 파싱을 이어서 한다.

```
void main() {
  Map<String, int> json = Map();
  json['x'] = 15;
  json['y'] = 20;

  var point = Point.fromJson(json);
  print("point x : ${point.x}, y: ${point.y}"); // point x : 15, y: 20 출력
}
```

Factory

팩토리 생성자는 팩토리 패턴을 쉽게 쓰기 위해 만들어졌다.

자바에서 팩토리 패턴을 쓰려면 팩토리 클래스를 따로 만들어줘야한다. 다트에서는 기본 생성자와 같이 쓸 수 있다.

자바로 팩토리 패턴을 구현해보고, 다트로도 구현을 해보도록 하겠다.

여기서는 로봇 회사의 예를 들어보겠다. 청소로봇, 전투용 로봇 등 다양한 로봇을 만드는 회사이다.

공장에서 로봇의 종류에 따라 생산을 한다.

JAVA - 로봇 클래스 선언, 각 로봇을 담당.

```
public abstract class Robot {
    abstract String getName();
    abstract String command();
}

public class CleanRobot extends Robot {
    @Override
    String getName() {
        return "Clean";
    }

    @Override
    String command() {
        System.out.println("clean a room");
        return "clean a room";
    }
}

public class WarRobot extends Robot {
    @Override
    String getName() {
        return "War";
    }

    @Override
    String command() {
        System.out.println("declare war");
        return "declare war";
    }
}
```

로봇 클래스를 만들었으면 팩토리 클래스를 만들어보자.로봇 이름에 따라 다른 클래스를 만들어준다.

JAVA - 팩토리 클래스, 모든 로봇의 생산을 담당

```
public class RobotFactory {

    Robot createRobot(String name) {
        switch(name) {
            case "Clean":
                return new CleanRobot();
            case "War":
                return new WarRobot();
        }

        return null;
    }
}
```

JAVA - 실행

```
RobotFactory factory = new RobotFactory();
Robot r1 = factory.createRobot("Android");
r1.command();
Robot r2 = factory.createRobot("iOS");
r2.command();
```

다트의 경우 기본 클래스 안에 팩토리 생성자를 만든다. 팩토리 생성자는 자바에서 팩토리 클래스랑 동일한 일을 한다. 따라서 추가적으로 팩토리 클래스를 만들 필요가 없다.

DART - 로봇 클래스, 팩토리 생성자 사용

```
abstract class Robot {

  Robot.create();

  factory Robot(RobotType type) { // 팩토리 생성자
    switch(type) {
      case RobotType.Clean:
        return CleanRobot();
      case RobotType.War:
        return WarRobot();
    }
  }

  String getName();
  String command();
}

enum RobotType{
  Clean,War
}
```

DART - 자식 클래스들

```
class CleanRobot extends Robot {

  CleanRobot(): super.create();

  @override
  String getName() {
    return "Clean";
  }

  @override
  String command() {
    print("clean a room");
    return "clean a room";
  }
}

class WarRobot extends Robot {

  WarRobot(): super.create();

  @override
  String getName() {
    return "War";
  }

  @override
  String command() {
    print("declare war");
    return "declare war";
  }
}
```

DART - 실행

```
Robot r1 = Robot(RobotType.Clean);
r1.command();
Robot r2 = Robot(RobotType.War);
r2.command();
```

따로 팩토리 클래스를 만들지 않았지만, 서로 다른 클래스를 만들어낸다.

Map

자바와 다트의 맵은 조금 다르다. 자바의 맵은 자유롭게 수정이 가능하지만, 다트는 수정 가능하게 쓸 수 도, 불가능하도록 쓸 수 도 있다.

자바는 맵에 키가 있어도 값을 추가한다.

JAVA

```
Map<String, Integer> map = new HashMap<>();
map.put("height", 175);
map.put("height", 180);
System.out.println(map.get("height")); // 180 출력
```

다트는 2가지 방법으로 맵을 관리한다.

- 키가 있어도 값을 추가하거나(수정 불가),
  - map[key] = value
- 키가 없을때만 값을 추가한다(수정 가능).
  - putIfAbsent 사용

DART

```
Map<String, int> map = Map<String, int>();
map.putIfAbsent("height", () => 175); // 키가 없을 때만 값을 추가
map.putIfAbsent("height", () => 180);

print(map["height"]); // 175 출력

map["height"] = 190; // 이미 키가 있어도 값을 추가
print(map["height"]); // 190 출력
```

Concurrency(동시성) - Isolate, Thread

다트에서는 어떻게 병렬 작업을 할까?

다트는 싱글 스레드이다. 여러 스레드를 만들 수 없다. 다트에선 Isolate를 만들어 병렬 처리를 한다. 스레드 안에 Isolate가 여럿 있는 걸 상상하면 된다.

Isolate는 분리된 작업 단위이다. Isolate는 각각의 메모리 힙(Each isolate has its own memory heap)가 있다. 락(lock)을 걸 수 없기에 경쟁 상태(race condition)나 데드락이 발생하지 않는다!

가장 기본이 되는 Isolate는 **main isolate** 이다. 다트 런타임에 의해 만들어진다.

main isolate는 필요에 따라 Isolate를 만들어 쓴다(spawn).

isolate를 만드는 걸 스폰(spawn)이라고 한다. 게임을 하면 몬스터를 스폰한다는 말을 종종 하는데 만든다는 의미다.

간단한 예제를 보자.

DART - Isolate 만들기(스폰하기)

```
void main() {
  Isolate.spawn(sendMessage, 'Hello');
  Isolate.spawn(sendMessage, 'Greetings');
  Isolate.spawn(sendMessage, 'Welcome');
}

void sendMessage(var message) {
  print('This is a ${message}');
}
```

main을 실행하면 3개의 Isolate가 만들어진다.

각각의 Isolate는 순서대로 실행되지는 않는다.

```
- 출력 결과 1
This is a 2. Greetings
This is a 1. Hello
This is a 3. Welcome

- 출력 결과 2
This is a 3. Welcome
This is a 2. Greetings
This is a 1. Hello
```

비동기처리 - Future, Async, Await

자바는 오래된 언어다 보니 비동기처리가 기본적으로 지원되지 않는다. 외부 라이브러리(RxJava)를 쓰거나, 최신 자바 버전을 써야한다. RxJava는 난이도가 좀 있는 편이라 처음엔 쓰기 어렵다.

다트의 비동기 처리는 간편하다. 퓨처(Future, 미래)란 객체를 쓰는데, 자세한 건 밑에서 알아보도록 하자.

JAVA

```
// Java 8 - async 함수를 사용해 비동기 처리를 했다.
CompletableFuture.runAsync(() -> {
            System.out.println("Run async in CompletableFuture");
        });

// RXJava - 새 스레드를 생성해서 비동기 처리를 했다.
Observable.just(1,2,3,4,5)
          .subscribeOn(Schedulers.io())
          .subscribe(
              numbers -> {
                  System.out.println(numbers);
          });
```

다트의 비동기는 Future, await, async로 구성되어 있다.

간단히 어떤 의미인지 알아보도록 하자.

- Future : 바로 끝나지 않는 작업을 할때 쓰인다. 일반적인 함수는 결과를 리턴하지만, 비동기 함수는 Future를 리턴한다.
- await : 비동기 처리가 끝날 때까지 기다린다는 의미다. 다른 작업을 진행하지 않는다. Future의 작업을 완료하고 다음 작업을 할때 쓰인다.
- async : 비동기 처리를 하겠단 선언이다. await는 async와 항상 함께 쓰인다.

아직 어떤 뜻인지 와닿지 않을 거다. 예제를 따라해보면서 이해해보자.

DART - 1. Future만 써보기 - second() 함수만 2초 뒤에 실행된다.

```
void main() {
    first();
    second();
    third();
}

void first() {
  print("First");
}

void second() async { // async는 비동기 처리하라는 의미.
  Future.delayed(Duration(seconds: 2), () { // 2초 딜레이
    print("Second");
  });
}

void third() {
  print("Third");
}
```

First,

Third,

Second

순으로 출력된다. Seoncd의 출력을 2초 늦췄기 때문이다.

second() 함수를 조금 수정해보겠다.

DART - 2. Future와 따로 실행해보기

```
void main() {
    first();
    second();
    third();
}

void second() async { // async는 비동기 처리하라는 의미.
  Future.delayed(Duration(seconds: 2), () {
    print("Future inside");
  });

  print("Second");
}
```

```
First
Second
Third
Future inside
```

순으로 실행된다.

Future를 함수 내에서 쓰더라도, 멈추지 않고 쭉 실행된다.

그러면 await를 쓰면 어떻게 될까?

DART 3. - await 사용 - 퓨처가 완료될때까지 기다림.

```
void main() {
    first();
    second();
    third();
}

void second() async { // async는 비동기 처리하라는 의미. await를 쓸 때는 꼭 붙여줘야한다.
  await Future.delayed(Duration(seconds: 2), () { // 먼저 실행된다.
    print("Future inside");
  });

  print("Second"); // 가장 마지막에 실행
}
```

```
First
Third
Future inside
Second
```

Future.delay가 print("Second") 보다 먼저 실행된다.

퓨처가 완료할때까지 나머지 부분이 기다린 것이다.

await를 쓰면 Future의 작업이 끝날 때까지 기다렸다 다음으로 넘어간다!

그렇다면 중간에 멈추지 않고 Future 값을 처리하려면 어떻게 해야할까?

then()을 써주면 된다.

비동기 파일 읽기

```
void main() {
  print("BEFORE");
  readFileAsync().then((data) => print(data));
  print("AFTER");
}

Future readFileAsync() async {
  return await File("file.json").readAsString();
}
```

아래는 출력값이다.

```
BEFORE
AFTER
{
  "name": "hochul",
  "age": 33
}
```

생각대로 파일 데이터가 가장 늦게 읽힌다.

반면 await를 쓰면 파일 읽기를 완료할때까지 다른 작업을 하지 않는다.

```
void main() {
  print("BEFORE");
  print(await readFileAsync());
  print("AFTER");
}
```

```
BEFORE
{
  "name": "hochul",
  "age": 33
}
AFTER
```

파일 읽기가 끝난 뒤에 AFTER를 출력한다.

await를 쓰면 간단히 비동기처리를 할 수 있지만, 메인 함수에서 쓰는 건 주의해야겠다.

Isolate vs Future

Isolate와 Future는 어떤 차이가 있을까?

- Isolate: 병렬 작업을 할 수 있게 해준다. 멀티 스레드와 유사하며 여러 Isolate를 만들어서 작업을 할 수 있다.
- Future: 비동기 처리를 하지만, 보통 한 스레드내에서 작업이 이뤄진다.

결론

자바와 다트는 큰 부분에서 비슷하지만, 세부적인 데서 차이가 있다.

자바 개발자라면 1~2일 정도면 다트 문접에 익숙해질 수 있다.

자료형, 생성자, 동시성, 비동기 처리등에서 다르긴 하지만 수월한 편이다.

빠르게 변하는 IT 세상에서 새 언어를 배우는 건 필수적이다.

주말 오후를 잠깐 투자해 다트를 배워보는 건 어떨까?

간결함에 놀라고 안드로이드와 IOS를 한번에 잡을 수 있는 매력 푹 빠져들게 될 것이다.

다음엔 플러터의 기본 레이아웃에 대해 알아보도록 하겠다!

참고 - http://cogitas.net/from-java-to-dart/ 참고 - https://hackernoon.com/are-futures-in-dart-threads-2cdc5bd8063a 참고 - http://jpryan.me/dartbyexample/