# 범석의 안드로이드 메모장



Flutter

## Dart 다트언어에 대하여 알아보기

by 봄석 2019. 11. 21.

# 다트 Dart 다트 언어에 대하여 알아보기


다트 언어에 대하여 알아보도록 하겠습니다.

아래의 다트 Document 원문에 대한 내용입니다.

https://dart.dev/guides/language/language-tour#class-variables-and-methods

## 목차

Table of Contents






최근글 인기글

[개발상식&언어] Runtime(..
2020.02.02

[개발상식&언어] 모아보기
2020.02.02

Flutter(플러터)가 Dart(다..
2020.02.02

Flutter - WidgetTest(위젯..
2019.12.17

Dart - Test 작성하기(tes..
2019.12.13


최근댓글

수정하겠습니다 ! 감사합니다

아조시... 3가지 방법이라면…

안녕하세요 궁금해요님. Rx…

안녕하세요 잘 읽어주셔서 …

태그

debounce, jenkins, joinToString, scheduler, rxandroid, zip, Observable, reduce, Workmanager, RxJava, Jetpack, JUnit 5, 리액티브연산자, Java, CI, AWS, Git, Firebase, When, List, map, Docker, kotlin, buildString, DART, 도커, flatMap, dart language,

## 1. 왜 다트를 해야할까 ?

다트는 구글이 자바스크립트를 대체하기 위해 2011년 개발한 웹 프로그래밍용 언어이지만 2018년이 될 때까지 배울 필요가 없는 프로그래밍 언어 중 하나로 여겨졌다. 하지만 다트는 새로운 프로그래밍 언어로, 나날이 큰 인기를 얻고 있습니다

현재 Dart는 다양한 VM과 컴파일러 지원을 통해 웹 프런트엔드부터 데스크톱, 서버, 모바일 애플리케이션까지 개발할 수 있다. Dart로 구축한 애플리케이션은 VM 위에서 동작합니다. Dart 프로젝트의 목적은 구조적이지만 유연하고, 프로그래머들에게 자연스럽고, 다양한 종류의 기기에서 돌아가게 하는 것입니다. 현재 구글의 크로스 플랫폼 앱 프레임워크인 Flutter가 이 언어를 사용하고 있습니다.

## 2. 다트의 중요한 개념

- 변수에 할당할 수 있는 모든 것은 객체 이며 모든 객체는 클래스 의 인스턴스입니다. 함수 및 null 도 모두 객체입니다. 모든 객체는 Object 클래스 에서 상속됩니다.

- Dart는 강력하게 타입적이지만. 유형이 예상되지 않는다고 할 때는 dynamic을 사용하여 타입에 lenient 하게 사용할 수도 있습니다.

- Dart는 List<int>(정수 목록) 또는 List<dynamic>(모든 유형의 개체 목록)과 같은 일반 유형을 지원합니다.

- Dart는 main() 함수 같은 탑 레벨 함수, 클래스 또는 객체에 연결된 함수, 함수 내의 함수 작성 등을 지원합니다.

- 마찬가지로 Dart는 최상위 변수 뿐만 아니라 클래스 또는 객체에 연결된 변수를 지원합니다. 인스턴스 변수는 필드 또는 속성이라고도 합니다.

- 자바와는 달리, 다트 키워드가 없습니다 public, protected하고 private. 식별자가 밑줄 (_)로 시작하면 라이브러리 전용입니다. 자세한 내용은 라이브러리 및 가시성을 참조하십시오.

- 식별자는 문자 나 밑줄 (_)로 시작하고 그 뒤에 문자와 숫자를 조합하여 사용할 수 있습니다.

- 다트에는 표현식 (런타임 값이 있음)과 명령문 (없음)이 있습니다. 예를 들어 조건식 condition ? expr1 : expr2의 값은 expr1또는 expr2입니다. 값이없는 if-else 문과 비교하십시오 . 명령문은 종종 하나 이상의 표현식을 포함하지만 표현식은 명령문을 직접 포함할 수 없습니다.

-다트 도구는 경고 와 오류 의 두 가지 문제를 보고 할 수 있습니다 . 경고는 코드가 작동하지 않을 수 있음을 나타내지 만 프로그램 실행을 방해하지는 않습니다. 오류는 컴파일 타임 또는 런타임 일 수 있습니다. 컴파일 타임 오류는 코드가 전혀 실행되지 못하게 합니다.런타임 오류로 인해 코드가 실행되는 동안 예외 가 발생합니다.

## 3. 다트 언어의 키워드

| keyword | description |
| --- | --- |
| abstract | 추상클래스 정의 키워드 |
| Implicit interfaces | 암시적 인터페이스<br>interface 키워드가 없음으로 일반 class로 정의된 내용을 암시적으로 구현하게 할 수 있습니다. |
| dynamic | 특정 타입을 지정하지 않고 변수를 선언할 수있는 키워드입니다. |
| show | 라이브러리의 일부만 선택적으로 가져올 수 있는 키워드입니다. |
| hide | 라이브러리의 일부분을 제외하고 가져올수 있는 키워드입니다. |
| as | Typecase에 사용됩니다. |
| is | 객체유형을 비교하고 bool을 반환합니다. |
| is! | 객체유형을 비교하고 bool을 반환합니다. |
| static | 전역 변수 및 전역 메서드 등을 선언할 수 있습니다.<br>상수 선언은 camelCase를 따릅니다. |
| assert | assert의 bool조건을 확인하여 false라면 에러를 발생시킵니다.<br>- flutter에서는 디버그 모드에서 assertion을 활성화합니다.<br>- dartdevc와 같은 개발 전용 도구는 기본적으로 assetion을 활성화합니다.<br>- 프로덕션 코드에서는 assertion이 무시됩니다. |
| enum | 열거형으로 고정된 상수의 값을 나타냅니다.<br>- 각 열거형 값에는 index getter가 있습니다. |

| | |
|---|---|
| | - 모든 값의 목록을 얻으려면 values 상수를 사용할 수 있습니다. |
| in | List와 Set과 같은 반복가능한 Collection에서 이 키워드를 사용하여 반복할 수 있습니다. |
| super | extents를 통해 상속받은 클래스에서 부모클래스를 참조할때 사용합니다. |
| async | 비동기 함수를 선언할 때 사용합니다.<br>- Future <String> lookUpVersion() async =>'1.0.0';<br>- Future checkVersion() async {<br>var version = await lookUpVersion( );<br>}<br>- async 함수는 시간이 오래 걸리는 작업을 수행할 수 있지만 해당 작업을 기다리지 않습니다. await를 만나면 실행하며 , await가 완료된 후에 실행을 재개하여 Future 객체를 반환합니다.<br>- async 함수의 에러 핸들은 try.catch 및 finally를 사용합니다. |
| await | 비동기 함수의 결과를 기다리는데 사용합니다. |
| sync | 동기 Generator함수를 구현할 때 sync* 로 선언하고,<br>yield를 이용하여 값을 전달합니다.<br><br>비동기 Generator함수를 구현할 때는 async* 를 사용하여 선언하고<br>yield를 이용하여 값을 전달합니다. |
| Iterable | 동기 Generator |
| Stream | 비동기 Generator |

| switch_case | switch_case 문 키워드 |
| --- | --- |
| factory | 팩토리 생성자 |
| mixin | - 믹스인은 다른 클래스 코드를 재사용하는 방법입니다.<br>- mixin으로 선언한 클래스는 다른 클래스에서 with 키워드를 이용하여 재사용할 수 있습니다.<br>- 일반적으로 abstract class 대신 사용합니다<br>- instantiated(인스턴트화) 할 수 없습니다 |
| with | mixin으로 선언한 클래스는 다른 클래스에서 with 키워드를 이용하여 재사용할 수 있습니다. |
| on | - mixin으로 선언한 클래스에서 사용할 상위 클래스 on키워드를 사용하여 선언합니다<br>- mixin A on B {}이라고 선언된 클래스가 있다면<br>- A를 with으로 사용하는 클래스는 반드시 B를 구현하거나 상속받아야 합니다.<br>- on 키워드를 사용하려면 mixin 키워드를 사용해서 믹스인을 선언해야 합니다. |
| throw | 이 키워드를 이용하여 예외를 던질 수 있습니다. |
| | - try { } 블록에서 에러가 발생할 수도 있는 구문을 작성합니다.<br>- on [SomeException]{ }블럭에서 SomeException 타입의 에러를 지정하여 처리할 수 있습니다.<br>- catch(e){ } 블록에서는 여러 유형의 예외를 처리하거나 on에서 지정한 타입의 excption을 매개변수로 받을 수 있습니다. |

| | |
|---|---|
| try, on , catch , finally | - catch(e, s){}로 선언하면 e로 excption을 , s는 stacktrace입니다.<br>- catch나 on블록 안에서 rethrow;를 사용하면 하위 블록으로 에러를 다시 던질 수 있습니다.<br>- finally : 예외 발생 여부와 상관없이 일부 코드가 반드시 실행되도록 합니다. |
| final , const | final은 최종 변수로 값을 변경할 수 없습니다<br>const는 상수 변수입니다. |
| null | 초기화되지 않은 변수의 초기 값은 null입니다. 숫자 형식의 변수(int)조차도 처음에는 null입니다. 숫자는 다트의 다른 모든 것과 마찬가지로 개체이기 때문입니다. |
| typedefs | 타입 정의 또는 기능 형 별칭 , 함수 유형을 필드 및 반환 형식을 선언할 때 사용할 수 있는 이름을 제공합니다. |
| covariant | 재정의 된 메서드에서 매개 변수는 슈퍼 클래스에있는 해당 매개 변수와 동일한 형식 또는 슈퍼 유형입니다.<br>- convariant를 사용하여 유형을 원래 매개 변수의 하위 유형으로 바꿀 수 있습니다.<br>- 일반적으로 수퍼 클래스 메서드에 적용하는 것이 가장 적합합니다.<br>- covariant키워드는 단일 매개 변수에 적용되며 또한 세터가 필드에서 지원됩니다. |

## 4. 다트 언어의 연산자

| Operator | Description |
|---|---|
| expr++ , expr-- , () , [] , ?. | 단항 접미사 |
| -expr ,!expr , ~expr , ++expr , --expr ,await, expr | 단항 접두사 |
| * , / , % , ~/ | 곱셈 |
| + , - | 덧 뺄샘 |
| << , >> , >>> | 시프트 |
| & | 비트 AND |
| ^ | 비트 XOR |
| \| | 비트 OR |
| >= > <= < as is is! | 부호 , 유형 |
| == , != | 부등호 |
| && | 논리 AND |
| \|\| | 논리 OR |
| ?? | null의 경우 |
| expr1 ? expr2 : expr3 | 가정 ? true : false |
| = *= /= += -= &= ^= | 할당 |

## 5. UseCase -keyword

### 5.1. abstract

abstract 키워드는 추상 클래스를 정의합니다.

```
abstract class AbstractContainer{
  void updateChildren();
}
```

```
class Container extends AbstractContainer{
  @override
  void updateChildren() {

  }
}
```

## 5.2. as, is

is : 타입 추론 연산자로 키워드 왼쪽 객체가 오른쪽의 타입인지 bool값으로 반환합니다.

as : 캐스팅 연산자로 오른쪽의 타입으로 캐스팅합니다.

```
main() {
  final emp = 1;
  print(emp is String);
  print(emp is! String);
  print(emp as int);
}
```

## 5.3. assert

assert의 bool조건을 확인하여 false라면 에러를 발생시킵니다.- flutter에서는 디버그 모드에서 assertion을 활성화합니다.

- dartdevc와 같은 개발 전용 도구는 기본적으로 assetion을 활성화합니다.

- 프로덕션 코드에서는 assertion이 무시됩니다.

```
main() {
  int text;
  text ??= 0;

  final number = 99;

  final urlString = "http";

  // Make sure the variable has a non-null value.
  assert(text != null);

// Make sure the value is less than 100.
  assert(number < 100);

// Make sure this is an https URL.
  assert(urlString.startsWith('https'),
  'URL ($urlString) should start with "https".');
}
```

## 5.4. async, await

**async 키워드는 비동기 함수를 선언할 때 사용합니다.**

- async 함수는 시간이 오래 걸리는 작업을 수행할 수 있지만 해당 작업을 기다리지 않습니다. await를 만나면 실행하며 , await가 완료된 후에 실행을 재개하여 Future 객체를 반환합니다.

- async 함수의 에러 핸들은 try.catch 및 finally를 사용합니다.

**await 키워드는 비동기 함수의 결과를 기다리는 데 사용합니다.**

```
//String lookUpVersion() => "1.0.0";

Future<String> lookUpVersion() async => "1.0.0";

Future checkVersion() async {
  var version = await lookUpVersion();
}

Future checkVersionErrorHandle() async {
  try {
    var version = await lookUpVersion();
  } catch (e) {
    // React to inability to look up the version
  }
}

Future checkVersionManyOfAsyncFunction() async {
  var version = await lookUpVersion();
  await lookUpVersion();
  await lookUpVersion();
}

Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  await for (var value in stream) {
    sum += value;
  }
  return sum;
}

Future handle(Stream<String> requestServer) async {
  await for (var request in requestServer) {
    handleRequest(request);
  }
}

void handleRequest(String request) {
  print(request);
}

main() {
  checkVersion();
```

```
  checkVersionErrorHandle();
  checkVersionManyOfAsyncFunction();

  final elements = [1, 2, 3];
  sumStream(Stream.fromIterable(elements));

  handle(Stream.fromIterable(["r1", "r2", "r3"]));
}
```

## 5.5. constructor(this)

클래스와 이름이 같은 함수를 생성하여 생성자를 선언합니다 (선택적으로 명명된 생성자에 설명된 추가 식별자 )

가장 일반적인 형태의 생성자 인 생성자 생성자는 클래스의 새 인스턴스를 만듭니다.

**this 키워드는 현재 인스턴스를 나타냅니다.**

```
class Point {
  num x, y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

## 5.6. convariant

convariant키워드는 매개 변수 유형을 다른 하위 유형으로 대체하여 사용할 수 있습니다.

```
class Animal {
  void chase(Animal x) {
        //     .....
  }
}

class Mouse extends Animal {}

class Cat extends Animal {
  void chase(covariant Mouse x) {
        //...
  }
}
```

위 예에서는 covariant하위 유형에서의 사용을 보여 주지만 covariant키워드는 슈

퍼 클래스 또는 하위 클래스 메서드에 배치 될 수 있습니다. 일반적으로 수퍼 클래스 메서드가 가장 적합합니다. covariant키워드는 단일 매개 변수에 적용되며 또한 세터과 필드에서 지원됩니다.

## 5.7. dynamic

명시적으로 타입이 예상되지 않도록 선언할 수 있습니다.

```
void log(dynamic object){
  print(object.toString());
}

bool convertToBool(dynamic arg){
  if(arg is bool) return arg;
  if(arg is String )return arg =='true';
  throw ArgumentError('Cannnot convert $arg to a bool');
}

main(){
  log(true);
  log(0.123124);
  log(1);
  log("sss");
}
```

## 5.8. enum

**Enum클래스는 은 고정된 수의 상수 값을 나타내는 데 사용되는 특수한 종류의 클래스입니다.**

열거 형을 사용할 수 있으며 모든 열거 형 값을 처리하지 않으면 경고가 표시됩니다.

열거된 유형에는 다음과 같은 제한이 있습니다.

- 열거 형을 하위 클래스로 만들거나 혼합하거나 구현할 수 없습니다.
- 열거 형을 명시 적으로 인스턴스화 할 수 없습니다.

```
enum Color { red, green, blue }


main() {
  assert(Color.red.index == 0);
  assert(Color.green.index == 1);
  assert(Color.blue.index == 2);

  // -------------------------
```

```
  List<Color> colors = Color.values;
  assert(colors[2] == Color.blue);

  // ----------------------

  var aColor = Color.blue;

  switch (aColor) {
    case Color.red:
      print('Red as roses!');
      break;
    case Color.green:
      print('Green as grass!');
      break;
    default: // Without this, you see a WARNING.
      print(aColor); // 'Color.blue'
  }
}
```

## 5.9. factory

### factory키워드는

항상 클래스의 새 인스턴스를 생성하지 않는 생성자를 구현할 때 키워드를 사용합니다.

예를 들어 팩토리 생성자는 캐시에서 인스턴스를 반환하거나 하위 유형의 인스턴스를 반환할 수 있습니다.

다음 예제는 캐시에서 객체를 반환하는 팩토리 생성자를 보여줍니다.

캐시에 이미 로거가 있다면 생성하지 않고, 없다면 생성합니다.

```
class Logger {
  final String name;
  bool mute = false;

  // _cache is library-private, thanks to
  // the _ in front of its name.
  static final Map<String, Logger> _cache = <String, Logger>{};

  factory Logger(String name) {
    return _cache.putIfAbsent(name, () => Logger._internal(name));
  }

  Logger._internal(this.name);

  void log(String msg) {
    if (!mute) print(msg);
  }
}

main() {
  var logger = Logger('UI');
```

```
  logger.log('Button cxlicked');

}
```

## 5.10. final

immutable 한 변수를 선언하는 키워드입니다.

```
main() {
  final name = 'Bob'; // Without a type annotation
  final String nickname = 'Bobby';

  ///컴파일 타임 상수const 가 될 변수에 사용하십시오 .
  const bar = 1000000; // Unit of pressure (dynes/cm2)
  const double atm = 1.01325 * bar; // Standard atmosphere

  ///const키워드는 상수 변수를 선언하기위한 것이 아닙니다.
  ///상수 값 을 생성하고 상수 값 을 생성하는 생성자를 선언하는 데 에도 사용할 수 있습니다
  ///. 모든 변수는 상수 값을 가질 수 있습니다.
  var foo = const [];
  final bars = const [];
  const baz = []; // Equivalent to `const []`

  //  name = 'Alice'; // Error: a final variable can only be set once.


  // Valid compile-time constants as of Dart 2.5.
  const Object i = 3; // Where i is a const Object with an int value...
  const list = [i as int]; // Use a typecast.
  const map = {if (i is int) i: "int"}; // Use is and collection if.
  const set = {if (list is List<int>) ...list}; // ...and a spread.
}
```

## 5.11. for


```
main() {
  var message = StringBuffer('Dart is fun');
  for (var i = 0; i < 5; i++) {
    message.write('!');
  }
  print(message);

  //------------------

  var callbacks = [];
  for (var i = 0; i < 2; i++) {
    callbacks.add(() => print(i));
  }
  callbacks.forEach((c) => c());

  //------------------
```

```
  var collection = [0, 1, 2];
  for (var x in collection) {
    print(x);
  }
}
```

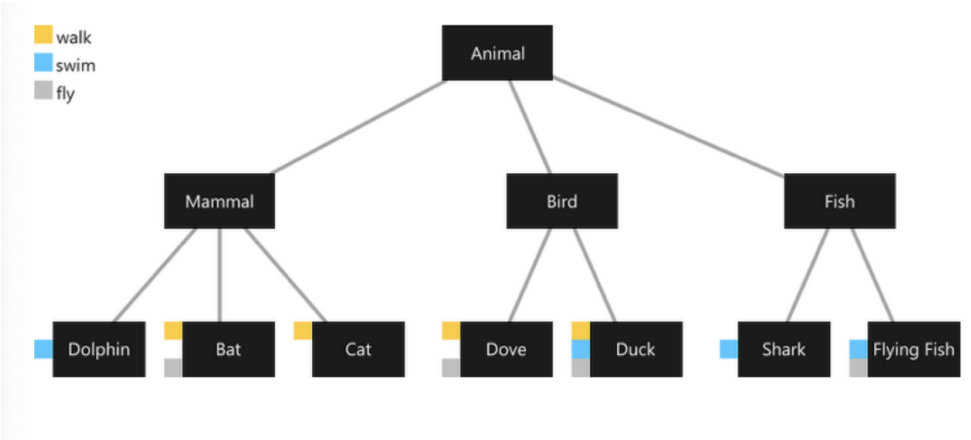
## 5.12. if, else

```
main() {
  if (isRaining())
    print("bringRainCoat");
  else if (isSnowing())
    print("wearJacket");
  else
    print("putToDown");
}

bool isRaining() => true;

bool isSnowing() => true;
```

## 5.13. mixin , on , with




Animal이라는 슈퍼클래스를 상속하는 Mammal , Brid , Fish가 있습니다.

그리고 Mammal , Brid , Fish를 상속하는 하위 클래스들이 있습니다.

그 하위 클래스들은 각각의 행위 fly , swim , walk 등의 행동을 할 수 있습니다.

하위 클래스들이 fly , swim , walk 등의 행동을 가질 때

하위 클래스들이 하나의 슈퍼 클래스만 가진다면 구현을 쉽게 할 수 있습니다.

위 클래스들을 적절히 상속만 시켜주면은 가능합니다.

하지만 Dart에서는 모든 클래스에는 ( Object 제외) 하나의 슈퍼클래스만 가질 수 있습니다.

Walker , Swim , Fly 행동 클래스를 상속하는 대신 인터페이스처럼 구현할 수 도 있지만
여러 클래스에서 동작을 구현해하므로 좋은 해결책이 아닙니다.
여러 클래스 계층에서 클래스코 드를 재사용할 방법으로 mixin을 사용합니다,

## mixin usecase 1

```
class A {}

mixin X on A {}

//class Y on A{}//에러

class p extends A with X {}

//class Q extends X {} //에러
```

on 키워드는 믹스인의 사용을 선언된 클래스를 확장하거나 구현하는 클래스로만 제한하는 데 사용됩니다.
on 키워드를 사용하려면 mixin 키워드를 사용해서 믹스인을 선언해야 합니다

## mixin usecase 2

```
class A {
  String getMessage() => 'A';
}

class B {
  String getMessage() => 'B';
}

class P {
  String getMessage() => 'P';
}

class AB extends P with A, B {}

class BA extends P with B, A {}
```

```
main(List<String> args) {
  callSequnce();

  typeCheck();
}

void typeCheck() {
  AB ab = AB();

  print(ab is P);
  print(ab is A);
  print(ab is B);

  BA ba = BA();

  print(ba is P);
  print(ba is A);
  print(ba is B);
}

void callSequnce() {
   String result = '';

  AB ab = AB();
  result += ab.getMessage();

  BA ba = BA();
  result += ba.getMessage();

  print(result);

  // 가장 마지막에 선언된 것이 우선됨
}
```

슈퍼클래스와 with으로 선언한 클래스들이 모두 같은 메소드를 가진다면

가장 마지막에 선언한 클래스의 메소드를 우선적으로 처리합니다.

## mixin usecase 3

```
abstract class Super {
  void method() {
    print("Super");
  }
}

class MySuper implements Super {
  void method() {
    print("MySuper");
  }
}

mixin Mixin on Super {
  void method() {
    super.method();
    print("Sub");
  }
}

class Client extends MySuper with Mixin {}
```

```
void main() {
  Client().method();
}
```

13 행에서 18 행까지의 mixin 선언은 Super에 대한 수퍼 클래스 제한을 나타냅니다

.

믹스 인이 슈퍼에서 제공하는 기능을 사용하기 때문에 이 믹스 인을 클래스에 적용하려면이 클래스가 슈퍼 클래스를 확장하거나 구현해야 합니다..

## mixin usecase animal

```
abstract class Animal {}

abstract class Mammal extends Animal {}

abstract class Bird extends Animal {}

abstract class Fish extends Animal {}

abstract class Walker {
  // This class is intended to be used as a mixin, and should not be
  // extended directly.
  factory Walker._() => null;

  void walk() {
    print("I'm walking");
  }
}

abstract class Swimmer {
  // This class is intended to be used as a mixin, and should not be
  // extended directly.
  factory Swimmer._() => null;

  void swim() {
    print("I'm swimming");
  }
}

abstract class Flyer {
  // This class is intended to be used as a mixin, and should not be
  // extended directly.
  factory Flyer._() => null;

  void fly() {
    print("I'm flying");
  }
}

class Dolphin extends Mammal with Swimmer {}

class Bat extends Mammal with Walker, Flyer {}

class Cat extends Mammal with Walker {}

class Dove extends Bird with Walker, Flyer {}

class Duck extends Bird with Walker, Swimmer, Flyer {}
```

```
class Shark extends Fish with Swimmer {}

class FlyingFish extends Fish with Swimmer, Flyer {}
```

원문 - https://medium.com/flutter-community/dart-what-are-mixins-3a72344011f3

## 5.14. show , hide

show 키워드는 show 키워드로 선언한 클래스만 import 합니다.

hide 키워드는 hide 키워드로 선언한 클래스를 제외한 나머지 클래스를 import 합니다.

```
// Import only foo.
import 'lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'lib2.dart' hide foo;

main() {
  foo();
}
```

## 5.15. static

```
import 'dart:math';

class Queue {
  static const initialCapacity = 16;
}

class Point {
  num x, y;
  Point(this.x, this.y);

  static num distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  assert(Queue.initialCapacity == 16);


  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
```

```
  print(distance);
}
```

## 5.16. super

서브클래스에서 슈퍼클래스를 참조하는 키워드입니다.

```
class Television {
  void turnOn() {
    print('is super turnOn Method');
  }
}

class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    print('is sub turnOn Method');
  }
}

main() {
  final smartTelevision = SmartTelevision();
  smartTelevision.turnOn();
}
```

## 5.17. switch-case

```
void case1() {
  var command = 'OPEN';
  switch (command) {
    case 'CLOSED':
      print('CLOSED');
      break;
    case 'PENDING':
      print('PENDING');
      break;
    case 'APPROVED':
      print('APPROVED');
      break;
    case 'DENIED':
      print('DENIED');
      break;
    case 'OPEN':
      print('OPEN');
      break;
    default:
      print('DEFAULT');
  }
}
```

## 5.18. sync* , async* , yield

**동기 Generator(ex Iterator) 함수를 구현할 때 sync*로 선언하고,**

**yield를 이용하여 값을 전달합니다.**

**비동기 Generator함수(ex Stream)를 구현할 때는 async* 를 사용하여 선언하고**

**yield를 이용하여 값을 전달합니다.**

```
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}

Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}

Iterable<int> naturalsDownFrom(int n) sync* {
  //생성기가 재귀 적 인 경우 다음을 사용하여 성능을 향상시킬 수 있습니다 yield*.
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n - 1);
  }
}

main() {
  print(naturalsTo(5));
  asynchronousNaturalsTo(10).forEach(print);
  print(naturalsDownFrom(5));
}
```

함수가 호출되면 generator함수를 통해 스트림이 만들어집니다. 그리고 스트림이 청취되기 시작하면 함수의 본문이 시작됩니다. 함수가 리턴되면 스트림이 닫힙니다. 스트림이 닫힐 때까지 yeild나 yeild*를 통해서 스트림으로 이벤트가 내보내집니다. yield*는  다른 비동기생성기로 위임하고 다른 생성기가 값 생성을 중지 한 후에 자체 값 생성을 다시 시작합니다.

## 5.19. throw

```
import 'dart:html';

//예외를 던지는 것은 표현식이므로 => 문과 표현식을 허용하는 다른 곳에서 예외를 throw 할 수 있습니다.
void distanceTo(Point other) => throw UnimplementedError();

main(){
  throw FormatException('Expected at least 1 section');
```

```
  throw 'Out of llamas!';
}
```

## 5.20. try_catch_on_finally

- try { } 블록에서 에러가 발생할 수도 있는 구문을 작성합니다.

- on [SomeException]{ }블럭에서 SomeException 타입의 에러를 지정하여 처리할 수 있습니다.

- catch(e){ } 블록에서는 여러 유형의 예외를 처리하거나 on에서 지정한 타입의 exception을 매개변수로 받을 수 있습니다.

- catch(e, s){}로 선언하면 e로 excption을 , s는 stacktrace입니다.

- catch나 on블록 안에서 rethrow;를 사용하면 하위 블록으로 에러를 다시 던질 수 있습니다.

- finally : 예외 발생 여부와 상관없이 일부 코드가 반드시 실행되도록 합니다.

```
void someErrorOccur() {
  print("excute...");
//  throw OutOfMemoryError();
  throw Exception();
//  throw 'Exception';
}

main() {
  try {
    someErrorOccur();
  } on OutOfMemoryError {
    // A specific exception
    print("is OutOfMemory Exception");
  } on Exception catch (e) {
// Anything else that is an exception
    print('Unknown exception: $e');
    rethrow;
  } catch (e, s) {
// No specified type, handles all
    print('Something really unknown.. Details: $e');
    print('Stack trace : $s');
  } finally {
    print('end excute..');
  }
}
```

## 5.21. typedefs

타입 정의 또는 기능 형 별칭 , 함수 유형을 필드 및 반환 형식을 선언할 때 사용할 수 있는 이름을 제공합니다.

```
typedef Compare = int Function(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);
  assert(coll.compare is Function);
  assert(coll.compare is Compare);
}
```

# 6. 다트 Detail Example

## 6.1. anonymous function

다트의 익명 함수는 아래와 같이 생성할 수 있습니다.

```
([[Type] param1[, …]]) {
  codeBlock;
  };
```

예

```
main() {
  var list = ['apples', 'bananas', 'oranges'];
  list.forEach((item) {
    print('${list.indexOf(item)}: $item');
  });

  //화살표 표기법을 이용하여 단축 가능
  list.forEach((item) => print('${list.indexOf(item)}: $item'));
}
```

## 6.2. collection

# Set

```
main() {
  //Set을 아래와 같이 정의 할 수 있습니다.
  var halogens = {
    'fluorine',
    'chlorine',
    'bromine',
    'iodine',
    'astatine',
    'fluorine'
  };
  assert(halogens.length == 5);

  // String 유형의 Set 정의
 // Set<String> names = {}; // This works, too.
  // var names = {}; // Creates a map, not a set.
  var names = <String>{};


  //Set에 데이터 추가
  names.add("ss");
  names.addAll(halogens);


  //상수  Set
  // constantSet.add('helium'); // Uncommenting this causes an error.
  final constantSet = const {
    'fluorine',
    'chlorine',
    'bromine',
    'iodine',
    'astatine',
  };
}
```

# Map

```
main(){
  //선언하며 데이터 추가
  var gifts = {
    // Key:    Value
    'first': 'partridge',
    'second': 'turtledoves',
    'fifth': 'golden rings'
  };

  var nobleGases = {
    2: 'helium',
    10: 'neon',
    18: 'argon',
  };



  //선언 후 데이터 추가
  var gifts2 = Map();
  gifts['first'] = 'partridge';
  gifts['second'] = 'turtledoves';
  gifts['fifth'] = 'golden rings';
```

```
  var nobleGases2 = Map();
  nobleGases[2] = 'helium';
  nobleGases[10] = 'neon';
  nobleGases[18] = 'argon';


  //상수 맵 선언
  // constantMap[2] = 'Helium'; // Uncommenting this causes an error.
  final constantMap = const {
    2: 'helium',
    10: 'neon',
    18: 'argon',
  };

}
```

## List

```
void main() {
  List<int> list = [2, 1, 3];
  list.add(4);
  print(list);
  print(list[0]);
}
```

## 6.3. 일급 객체

다트는 언어는 매개변수로 함수를 전달할 수 있고

변수에 함수를 할당할 수 있는 일급 객체를 지원합니다.

```
void printElement(int element) {
  print(element);
}

var list = [1, 2, 3];

main() {
  //매개변수로 함수 전달
  list.forEach(printElement);


  //변수에 함수 할당
  var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
  assert(loudify('hello') == '!!! HELLO !!!');
}
```

## 6.4. 암시적 인터페이스(implic_interface)

다트에서는 interface 키워드가 없습니다

일반 class로 정의 한 후 구현하고자  하는 클래스에서  implements를 사용하여

정의된 함수나 변수를 모두 구현하게 할 수 있습니다.

```
class Person {
  final _name;

  Person(this._name);

  String greet(String who) => "Hello, $who. I am $_name";
}

class Impostor implements Person{

  @override
  get _name => '';

  @override
  String greet(String who)=> "Hi, $who. Do you know who I am ?";
}

String greetBob(Person person)=>person.greet("Bob");

main(){
  print(greetBob(Person("kathy")));
  print(greetBob(Impostor()));
}
```

## 6.5. 어휘 폐쇄(lexcial_closures)

폐쇄 기능은 원래의 범위 밖에서 사용되는 경우에도, 그 어휘 범위의 변수에 액세스하는 기능 개체입니다.

함수는 주변 범위에 정의된 변수를 닫을 수 있습니다. 다음 예제에서는 makeAdder( ) 변수를 캡처합니다

addBy. 반환된 함수가 어디를 가든 기억 addBy 합니다.

```
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

void main() {
  // Create a function that adds 2.
  var add2 = makeAdder(2);

  // Create a function that adds 4.
  var add4 = makeAdder(4);

  assert(add2(3) == 5);
  assert(add4(3) == 7);
}
```

## 6.6. 중첩 함수

함수 안에 중첩하여  함수를 선언할 수 있습니다.

```
bool topLevel = true;

void main() {
  var insideMain = true;

  void myFunction() {
    var insideFunction = true;

    void nestedFunction() {
      var insideNestedFunction = true;

      assert(topLevel);
      assert(insideMain);
      assert(insideFunction);
      assert(insideNestedFunction);
    }
  }
}
```

## 6.7. 스프레드 연산자(spread operator)와 컬렉션

스프레드 연산자 (...)와 널 인식 스프레드 연산자 (...?)를 도입하여 여러 요소를 컬렉션에 간결하게 삽입할 수 있습니다.

예를 들어, 스프레드 연산자 (...)를 사용하여 목록의 모든 요소를 다른 목록에 삽입할 수 있습니다.
스프레드 연산자의 오른쪽에 있는 표현식이 벌인 경우 널 인식 스프레드 연산자 (...?)를 사용하여 예외를 피할 수 있습니다.

또한 컬렉션에 if 및 for가 도입되어 조건부 ( if) 및 반복 ( for)을 사용하여 컬렉션을 작성하는 데 사용할 수 있습니다.

```
main() {
  var list = [1, 2, 3];
  var list2 = [0, ...list];
  assert(list2.length == 4);

  var list3;
  var list4 = [0, ...?list3];
  assert(list4.length == 1);

  //collection if
  bool promoActive = true;
  var nav = ['Home', 'Furniture', 'Plants', if (promoActive) 'Outlet'];
  assert(nav.length == 4);

  //collection for
  var listOfInts = [1, 2, 3];
  var listOfStrings = ['#0', for (var i in listOfInts) '#$i'];
  assert(listOfStrings[1] == '#1');
```

```
  assert(listOfStrings.length == 4);
}
```

## 6.8. 널 인식 연산자 (null_aware_operators)

??= 을 사용하여 해당 값이  null이라면 초기화를 진행합니다.

nul이 아니라면 값이 변경되지 않습니다.

```
//널 인식 연산자
 int a; // The initial value of a is null.
 a ??= 3;
 print(a); // <-- Prints 3.

 a ??= 5;
 print(a); // <-- Still prins 3.

 print(1 ?? 3); // <-- Prints 1.
 print(null ?? 12); // <-- Prints 12.
```

## 6.9. 문자열 보간(string Interpolation)

"" 안에  $또는 ${}으로  값을 부여할 수 있습니다.

```
print('${3 + 2}');
print('${"word".toUpperCase()}');
```

## 6.10. 조건부 프로퍼티 접근(conditionalPropertyAccess)

null 일 수 있는 객체의 속성이나 메서드에 대한 접근을 보호하려면 점 (.) 앞에 물음표 (?)를 넣으십시오

null을 체크하면서 프로퍼티나 함수 호출이 가능합니다.

```
var myObject = new MyObject();

myObject
    ?.myProperty; //null 일 수있는 객체의 속성이나 메서드에 대한 액세스를 보호하려면 ?점 ( .) 앞에 물음표 ( )를 넣으십시오 .

myObject?.myProperty?.someMethod(); //null 체크를 하며 , 연속 호출 가능
```

## 6.11. 화살표 문법(arrow syntax)

=> 다트 코드에서 기호를 보았을 것입니다. 이 화살표 구문은 표현식을 오른쪽으로 실행하고 값을 반환하는 함수를 정의하는 방법입니다.

```
final aListOfStrings = ['one', 'two', 'three'];
```

```
  bool hasEmpty = aListOfStrings.any((s) {
    return s.isEmpty;
  });

  bool hasEmpty2 = aListOfStrings.any((s) => s.isEmpty);

  //위의 두 방법은 모두 같은 방법입니다.
```

## 6.12. cascade

동일한 객체에서 일련의 작업을 수행하려면 cascade (..)을 사용하십시오.

우리는 모두 다음과 같은 표현을 보았습니다.

자바의 Builder Pattern처럼 사용 가능합니다.

알아야 할 것은 리턴 타입이 자기 자신이 아니며 참조라는 것입니다.

```
void cascade(){
  final myObject = new MyObject();
  myObject..someMethod()..someMethod()..someMethod();


  var bigObject = fillBigObject(new BigObject());
  print(bigObject.anInt);
  print(bigObject.aString);
  print(bigObject.aList);
}

BigObject fillBigObject(BigObject obj) {
  // Create a single statement that will update and return obj:
  return obj
    ..anInt = 1
    ..aString = 'String!'
    ..aList = [3, 0]
    ..allDone();
}

//cascade not used
var button = querySelector('#confirm');
 button.text = 'Confirm';
 button.classes.add('important');
 button.onClick.listen((e) => window.alert('Confirmed!'));


//cascade used
querySelector('#confirm')
 ..text = 'Confirm'
 ..classes.add('important')
 ..onClick.listen((e) => window.alert('Confirmed!'));
```

## 6.13. Getter , Setter

```
class MyClass {
  int _aProperty = 0;

  int get aProperty => _aProperty;

  set aProperty(int value) {
    if (value >= 0) {
```

```
    _aProperty = value;
  }
 }

 List<int> _values = [];

 void addValue(int value) {
  _values.add(value);
 }

 // A computed property.
 int get count {
  return _values.length;
 }

 int get countSum => _values.fold(0, (acc, v) => acc + v);
}

 final myClass = new MyClass();
 myClass.aProperty = -1;
 print(myClass.aProperty);
 myClass.aProperty = 10;
 print(myClass.aProperty);

 myClass.addValue(1);
 myClass.addValue(2);
 myClass.addValue(3);
 print(myClass.count);
 print(myClass.countSum);
}
```

## 6.14. 선택적 파라미터(OptionalPositionalParameters)

매개변수 선언에 []를  사용하여 매개변수를 선택적으로 사용할 수 있습니다.

```
int sumUp(int a, int b, int c) => a + b + c;

int sumUpToFive(int a, [int b, int c, int d, int e]) {
 int sum = a;
 if (b != null) sum += b;
 if (c != null) sum += c;
 if (d != null) sum += d;
 if (e != null) sum += e;
 return sum;
}

//Dart를 사용하면 매개 변수를 대괄호로 묶어 선택적으로 지정할 수 있습니다.
 int total;
 total ??= 0;
 total = sumUp(1, 2, 3);
 print(total);
 total = sumUpToFive(1, 2);
 print(total);
 total = sumUpToFive(1, 2, 3, 4, 5);
 print(total);
```

## 6.15. 명명 생성자, 생성자에 this 사용하기

### 1.this. 를 사용한 생성자 자

```
class MyColor {
  int red;
  int green;
  int blue;

  MyColor(this.red, this.green, this.blue); // this.를 사용한 생성자 자
}
```

```
final color2 = MyColor(80,80,128);
```

### 2. 명명  매개변수 생성자

```
class MyColor {
  int red;
  int green;
  int blue;

  MyColor({this.red, this.green, this.blue}); // 명명된 매개변수 생성자
}

final color2 = MyColor(red:80,blue:80,green:128);
```

### 3. 명명 매개변수 생성자와 기본값 0

```
class MyColor {
  int red;
  int green;
  int blue;

  MyColor({this.red=0,this.green=0,this.blue=0}) ; //명명 매개변수 생성자와 기본값 0
}

  final color2 = MyColor(red: 80, green: 80, blue: 128);
```

### 4. 선택적 매개변수와 기본값 0

```
class MyColor {
  int red;
  int green;
  int blue;

   MyColor([this.red=0,this.green=0,this.blue=0]) ; //선택적 매개변수와 기본값 0
}
```

## 6.16. 생성자와 assert

생성자를  이용해 객채를  생성할 때 ,

생성자의 입력값을 assert로 확인할 수 있습니다.

```
class NonNegativePoint {
  int x;
  int y;
```

```
  NonNegativePoint(this.x, this.y)
      : assert(x >= 0),
        assert(y >= 0) {
    print('I just made a NonNegativePoint: ($x, $y)');
  }
}
```

## 6.17. 명명 생성자

생성자에  다른 이름을 부여할 수 있습니다.

```
class Points {
  num x, y;

  Points(this.x, this.y);

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

void namedConstructors() {
  final myPoints = Points.origin();
  print("${myPoints.x},${myPoints.y}");
}
```

## 6.18. 팩토리 생성자 ( factory constructor)

팩토리 생성자를 이용하여 여러 타입의 객체를 생성할 수 있습니다.

```
abstract class Shape {
  num get area;

  factory Shape(String type) {
    if (type == 'circle')
      return Circle(2);
    else if (type == 'square')
      return Square(2);
    else
      return null;
  }
}

void factoryConstructor(){
  final circle = Shape('circle');
  final square = Shape('square');
  final nullShape= Shape('aa');
  print(circle.area);
  print(square.area);
  print(nullShape);
}
```

## 6.19. 리디렉션 생성자(redirectingConstructor)

this 키워드를 이용하여 다른  생성자를 호출할 수 있습니다.

```
class Automobile{
  String make;
  String model;
  int mpg;

  // The main constructor for this class.
  Automobile(this.make,this.model,this.mpg);

  // Delegates to the main constructor.
  Automobile.hybrid(String make,String model):this(make,model,60);

  // Delegates to a named constructor
  Automobile.fancyHybrid() : this.hybrid('Futurecar', 'Mark 2');
}
```

## 6.20. 상수 생성자(ConstConstructors)

클래스가 절대 변경되지 않는 객체를 생성하는 경우 이러한 객체를 컴파일 타임 상수로 만들 수 있습니다.

이렇게 하려면 const생성자를 정의하고 모든 인스턴스 변수가 최종적인지 확인하십시오.

```
class ImmutablePoint {
  const ImmutablePoint(this.x, this.y);

  final int x;
  final int y;

  static const ImmutablePoint origin =
  ImmutablePoint(0, 0);
}

void constConstructors(){
  final immutablePoints =ImmutablePoint.origin;
  print("immutablePoints ${immutablePoints.x},${immutablePoints.y}");
}
```

## 6.21. functional dart

```
String scream(int length) => "A${'a' * length}h!";

main() {
  //origin
  final values = [1, 2, 3, 5, 10, 50];
  for (var length in values) {
    print(scream(length));
  }

  //functional
  values.map(scream).forEach(print);

  //complex functional
```

```
  values.skip(1).take(3).map(scream).forEach(print);
}
```

## 7. 전체 샘플 코드 보러 가기

https://github.com/qjatjr1108/Playground_Dart


## 8. Reference

https://dart.dev/guides/language/language-tour#class-variables-and-methods

https://medium.com/flutter-community/dart-what-are-mixins-3a72344011f3

-

1 구독하기

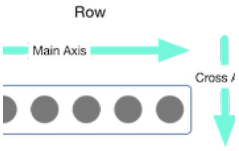
'**Flutter**' 카테고리의 다른 글

| | |
|---|---|
| Flutter - BLoC 패턴 알아보기 (0) | 2019.12.13 |
| Flutter - Row,Column정렬하기 (MainAxisAlignment, CrossAxisAlignment) (0) | 2019.12.02 |
| Dart 언어 Future 알아보기 (0) | 2019.11.25 |
| Dart 언어 Stream 알아보기(Dart 비동기 프로그래밍) (0) | 2019.11.25 |
| Dart 언어 컨벤션 알아보기 (0) | 2019.11.24 |
| Dart 다트언어에 대하여 알아보기 (0) | 2019.11.21 |

## 태그

Async await convariant dart language mixin typedefs yield

## 관련글




Flutter - Row,C… Dart 언어 Futur… Dart 언어 Strea… Dart 언어 컨벤션…

## 댓글 0

| 이름 | 비밀번호 |
| --- | --- |
| 여러분의 소중한 댓글을 입력해주세요. | |

비밀글 등록

1 · · · 7 8 9 10 11 12 13 14 15
· · · 270

TEL. 02.1234.5678 / 경기 성남시 분당구 판교역로