클래스

1. 기본

```
// 클래스 정의
class Person {
  //멤버 변수
  String? name;

  //멤버 함수 => Method    / 클래스 외부에서 하나의 기능을 수행하는 함수 => Function
  getName() {
    return this.name;
  }
}

main() {
  // 객체(인스턴스) 생성
  Person student = new Person(); // 객체를 생성하는 경우 new 연산자를 사용
  var teacher = Person(); // new 연산자를 생략할 수 있음

  // 객체 멤버 변수 접근
  student.name = 'Kim';
  teacher.name = 'Park';

  // 객체 멤버 함수 호출
  print('student name : ${student.getName()}');
  print('teacher name : ${teacher.getName()}');
}
```

2. 생성자

   a. 기본 생성자(Default constructor)

   - 클래스명과 동일하고 매개변수가 존재하지 않으며 생성자를 생략할 경우 자동으로 제공

```
class Person {
  //생성자가 생략된 경우 컴파일 시 자동생성
  //Person() {} // 작성할 경우 클래스명과 동일하며 매개변수가 없음

  //멤버 변수
  String? name;

  //멤버 함수
  getName() {
    return this.name;
  }
}

main() {
  // 객체(인스턴스) 생성
  Person person = new Person();
  print(person.getName());
}
```

   b. 이름이 없는 생성자(Generative constructor**)**

- 클래스명과 동일한 이름으로 선언되는 생성자로 기본 생성자와 달리 매개변수를 가질 수 있음
- 클래스 내에 단 하나만 선언가능

```
class Person {
  // 이름이 없는 생성자
  Person(String name) {
    this.name = name;
  }

  // Person() {}  // 이미 이름이 없는 생성자가 존재하므로 기본 생성자라도 불가

  // 멤버 변수
  String? name;

  // 멤버 함수
  getName() {
    return this.name;
  }
}

main() {
  // 객체(인스턴스) 생성
  Person person = new Person('Hong');
  print(person.getName());
}
```

c. 이름이 있는 생성자(Named constructor)

- 생성자에 이름을 부여한 형태로 한 클래스 내에 많은 생성자를 생성하거나 생성자를 명확히 하기 위해 사용
- 이름이 있는 생성자를 선언하면 기본 생성자는 자동으로 생성되지 않기 때문에
  필요할 경우 기본 생성자를 별도로 작성하거나 반드시 이름이 있는 생성자를 사용

```
class Person {
    // 이름이 있는 생성자 선언
    // 클래스명.생성자명() {}

  Person.init(String name) {
    this.name = name;
  }

  Person.initName() {
    this.name = 'Lee';
  }

  // 멤버 변수
  String? name;

  // 멤버 함수
  getName() {
    return this.name;
  }
}

main() {
  // 이름이 있는 생성자 중 Person.init(String)으로 생성
  Person student = Person.init('Hong');
```

```
  print('student ${student.getName()}');

  //var teacher = new Person(); // 기본 생성자가 클래스 내에 명시되지 않음
  // 이름이 있는 생성자 중 Person.initName()으로 생성
  var teacher = new Person.initName();
  print('teacher ${teacher.getName()}');
}
```

d. 초기화 리스트(Initializer list)

- 생성자의 구현부가 실행되기 전에 인스턴스 변수를 초기화

```
class Person {
  // 초기화 리스트 선언
  // 생성자 옆에 :(콜론)으로 초기화할 매개변수들을 설정
  Person() : name = 'Kim' {
    print('name init : ${this.name}');
    print('age init : ${this.age}');
  }

  // 멤버 변수
  String? name;
  int? age;

  // 멤버 함수
  showInfo() {
    return '$name , $age';
  }
}

main() {
  Person person = Person();
  print(person.showInfo());
}
```

e. 리다이렉팅 생성자(Redirecting constructor)

- 생성자의 목적이 오직 같은 클래스 내의 다른 생성자로의 리다이렉트(redirect)인 경우
- 리다이렉팅 생성자의 바디는 비어있고 콜론 (:) 뒤에 나오며 클래스 이름 대신 `this`를 사용한 생성자 호출로 구성

```
class Person {
  // 이름이 있는 생성자
  Person(this.name, this.age) {}
    // 리다이렉팅 생성자 => 이름이 있는 생성자 호출
  Person.init(String name) : this(name, 20);

  // 멤버 변수
  String? name;
  int? age;

  // 멤버 함수
  showInfo() {
    return '$name , $age';
  }
}
```

```
main() {
  Person person = Person.init('Kim');
  print(person.showInfo());
}
```

f. 상수 생성자(Constant constructor)

- 특정 클래스가 불변성을 가지는 객체를 생성할 경우 사용
- 생성자를 `const`로 정의하고 모든 인스턴스 변수를 `final`로 선언

```
class Person {
  // 상수 생성자
  const Person(this.name, this.age);

  // 멤버 변수 => 모두 final 키워드를 붙임
  final String name;
  final int age;

  // 멤버 함수
  showInfo() {
    return '$name , $age';
  }
}

main() {
  Person personOne = const Person('Kim', 25); // 새로운 객체 생성
  Person personTwo = const Person('Kim', 25); // 동일한 객체 참조
  Person personThree = new Person('Kim', 25); // 새로운 객체 생성
  Person personFour = new Person('Kim', 25); // 새로운 객체 생성

  // 각 객체의 멤버 변수 값을 출력
  print('one : ${personOne.showInfo()}');
  print('two : ${personTwo.showInfo()}');
  print('three : ${personThree.showInfo()}');
  print('four : ${personFour.showInfo()}');

  // 각 객체가 실제로 동일한 객체인지 확인
  print(identical(personOne, personTwo));
  print(identical(personTwo, personThree));
  print(identical(personThree, personFour));
}
```

g. 팩토리 생성자(Factory constructor)

> 팩토리 메서드 패턴 : 객체 생성을 공장(Factory) 클래스로 캡슐화 처리하여 대신 생성하게 하는 생성 디자인 패턴

- 항상 클래스의 새로운 인스턴스를 생성하지 않는 생성자를 구현하고 싶다면, `factory` 키워드를 사용

```
class Person {
  Person.init();

  // 팩토리 생성자
  factory Person([String type = 'Person']) {
    switch (type) {
```

```
      case 'Student':
        return Student();
      case 'Employee':
        return Employee();
      default:
        return Person.init();
    }
  }

  // 멤버 함수
  String getType() {
    return 'Person';
  }
}

class Student extends Person {
  Student() : super.init();

  @override
  String getType() {
    return 'Student';
  }
}

class Employee extends Person {
  Employee() : super.init();

  @override
  String getType() {
    return 'Employee';
  }
}

main() {
  var person = Person();
  var student = Person('Student');
  var employee = Person('Employee');

  print('person variable type : ${person.getType()}');
  print('student variable type : ${student.getType()}');
  print('employee variable type : ${employee.getType()}');
 }
```

3. 상속

- 부모 클래스의 멤버(변수, 함수)를 자식 클래스에 물려주는 것
- 코드의 재사용으로 클래스가 간소화되고 수정 및 추가가 효율적

```
class Person {
  // 멤버 변수
  String? name;

  // 생성자
  Person() {
    print('Person Objet Created');
  }
  // 멤버 함수
  setName(String name) {
```

```
    this.name = name;
  }

  getName() {
    return this.name;
  }

  showInfo() {
    print('name is $name');
  }
}

class Student extends Person {
  // 자식 클래스의 멤버 변수
  int? studentId;

  Student() {
    // 자식 클래스의 생성자를 호출하는 경우 부모의 생성자가 자동 호출
    print('Student Object Created');
  }

  @override // 부모의 메서드를 오버라이딩하는 경우로 필요에 따라 어노테이션은 생략가능
  showInfo() {
    // 자식 클래스 내부에서 부모 클래스의 멤버에 접근할 때 super 키워드를 사용
    // super.showInfo();
    print('name is ${super.getName()} and ID is $studentId');
  }

  // 자식 클래스의 멤버 함수
  getType() {
    return 'Student';
  }
}

main() {
  // 자식 클래스로 객체 생성
  Student student = Student();
  student.studentId = 1024; // 자식 클래스의 멤버 변수
  student.setName('Kim'); // 부모 클래스의 멤버 함수
  student.showInfo(); // 자식 클래스가 오버라이딩한 멤버 함수
  print(student.getType()); // 자식 클래스의 고유 멤버 함수
}
```

```
Person Objet Created
Student Object Created
name is Kim and ID is 1024
Student
```

4. 접근 지정자

> **객체지향 4대 특징**
>
> 1. 추상화

- 객체의 공통적인 속성과 기능을 도출하여 클래스로 정의하는 과정

2. 캡슐화
    - 객체를 구성하는 데이터(멤버 변수)와 데이터를 처리하는 메서드(멤버 함수)를 묶고
      그 중 일부를 외부에서 접근하지 못하도록 숨김으로써 객체의 상세 구현을 외부로부터 감추는 것
3. 상속
    - 기존 클래스의 속성(멤버 변수)와 기능(멤버 함수)을 다르클래스가 물려받는 것
    - 기존 코드의 재사용과 확장이 가능해지며 중복을 최소화
4. 다형성
    - 하나의 참조 변수로 여러타입의 객체를 참조하거나 같은 이름의 메서드가 다양한 방식으로 동작하는것
    - 타입변환 + 오버라이딩/오버로딩

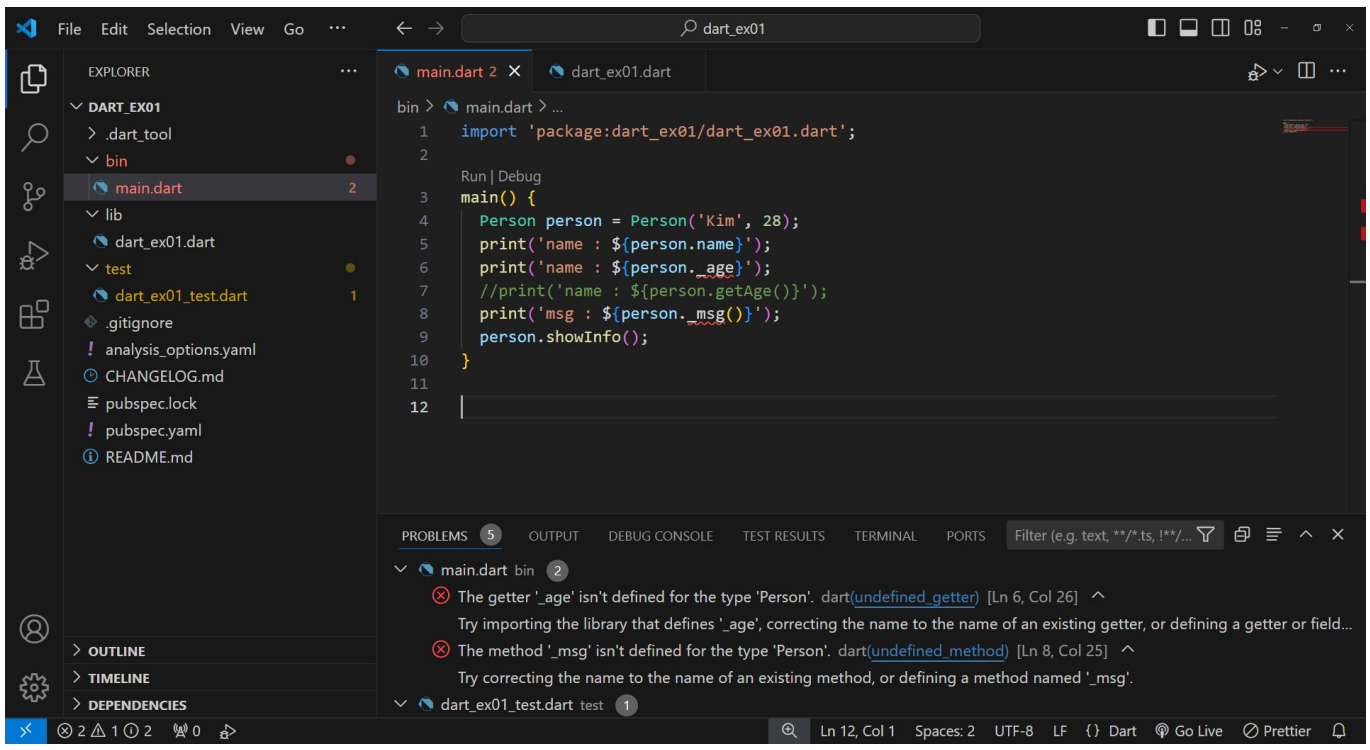
- 클래스의 멤버 변수 또는 메서드에 접근할 수 있는 범위를 지정

1. public
    - 멤버 변수 또는 멤버 함수 앞에 어떤 키워드도 없는 경우 기본적으로 설정되는 지정자
    - 범위에 제한이 없음
2. private
    - 멤버 변수 또는 멤버 함수 앞에 _(밑줄)을 붙이는 경우 설정되는 지정자
    - 범위는 라이브러리(=자바 기준 패키지) 기준 내부에서만 접근 가능

```
class Person {
  // 멤버 변수
  String? name;  // public
  int? _age;     // private

  // 생성자
  Person(this.name, this._age);

  // 멤버 함수
  getAge() {     // public
    return this._age;
  }
  _getMsg(){     // private
    return 'Hello !';
  }

  showInfo() {     // public
    print('${_getMsg()}, My name is $name and age is $_age');
  }
}
```

```
import 'package:dart_ex01/dart_ex01.dart';

main() {
  Person person = Person('Kim', 28);
  print('name : ${person.name}');
  //print('name : ${person._age}');  // _age는 private이라 접근 불가
  print('name : ${person.getAge()}');
```

```
  //print('msg : ${person._msg()}'); // _msg()는 private이라 접근 불가
  person.showInfo();
}
```



5. getter & setter

- public으로 선언된 모든 멤버 변수는 접근에 제한이 없으므로 누구나 자유롭게 변경이 가능
  ⇒ 해당 멤버 변수가 가져야 하는 도메인을 벗어난 값이 할당될 위험이 존재
- 클래스의 내부 정보를 외부에 공개하지 않도록 정보 은닉을 위한 방법으로 전용 메서드가 존재
- 멤버 변수를 private으로 선언하고 해당 변수에 접근하는 메서드를 public으로 선언

```
// 기본 형태
class Person {
  // 멤버 변수
  String _name; // private으로 선언

  // 생성자
  Person(this._name);

  // 멤버 함수 : 일반
  /*
  String get name {  // 매개변수를 인자로 받을 수 없음
    return _name;
  }

  void set name(String name) {
    this._name = name;
  }
  */

  // 멤버 함수 : 람다식
  String get name => _name;
  set name(String name) => _name = name;
}

main() {
  Person person = Person('Kim');
  print(person.name);
```

```
  person.name = 'Hong';
  print(person.name);
}
```

- getter & setter 은 기본적으로 함수의 형태를 띄고 있으므로 값을 할당하거나 반환할 경우 제어가 가능

```
class Car {
  String name;
  double _speed;

  Car(this.name) : _speed = 0;

  double get speed => _speed;

  // 자동차의 속도는 음수를 가질 수 없다.
  set speed(double speed) => _speed = (speed > 0) ? speed : 0;

  /*
  void set speed(double speed) {
    if (speed > 0) {
      this._speed = speed;
    } else {
      this._speed = 0;
    }
  }
  */
}

main() {
  Car myCar = Car('CASPER');
  print('speed init : ${myCar.speed}');

  myCar.speed = 100;
  print('speed update : ${myCar.speed}');

  myCar.speed = -50;
  print('speed error : ${myCar.speed}');
}
```

6. 추상 클래스

- 기본

```
// 추상 클래스
// - 추상 메서드를 가질 수 있다.
// - 인스턴스를 생성할 수 없으며 반드시 구현 클래스를 통해 생성
abstract class Person {
  work(); // 추상 메서드 : 함수의 몸체가 없다.

  // 필요에 따라 일반 메서드를 작성할 수 있다.
  study() {
    print('People are studying.');
  }
}

// 추상 클래스 Person을 구현한 클래스
class Developer implements Person {
```

```
  // 추상 메서드든 일반 메서드든 구분없이
  // 추상 클래스 내부에 선언된 모든 메서드는 오버라이딩해야 한다.
  @override
  work() {
    print('Developers are developing.');
  }

  @override
  study() {
    print('Developers are studying.');
  }
}

main() {
  // 추상 클래스의 경우 구현 클래스를 통해 인스턴스를 생성하나
  // 타입으로는 사용가능
  Person person = Developer();
  person.work();
  person.study();
}
```

- 상속과 달리 추상 클래스를 구현하는 구현 클래스의 경우
  여러 추상 클래스를 함께 구현할 수 있음

```
// 추상 클래스
abstract class Person {
  eat();

  study() {
    print('People are studying.');
  }
}

abstract class Junior {
  work() {
    print('work hard');
  }
}

// 추상 클래스 Person와 Junior을 구현한 클래스
class Developer implements Person, Junior {
  // Person
  @override
  eat() {
    print('Developer eat a meal');
  }

  @override
  study() {
    print('Developers are studying.');
  }

  // Junior
  @override
  work() {
    print('Developers are developing.');
  }
}
```

```
main() {
  // 타입을 Person으로 선언한 경우 Junior의 메서드 사용불가
  Person person = Developer();
  person.eat();
  person.study();

  // 타입을 Junior으로 선언한 경우 Person의 메서드 사용불가
  Junior junior = Developer();
  junior.work();

  // 타입을 Developer으로 선언한 경우 오버라이딩한 모든 메서드 사용가능
  Developer developer = Developer();
  developer.eat();
  developer.study();
  developer.work();
}
```