객체 지향 다트 프로그래밍

다트에서 객체 지향 프로그래밍의 이해와 활용

학습목표

- 다트에서 객체 지향 프로그래밍의 핵심 개념 이해 (클래스, 객체, 상속, 인터페이스, 믹스인)
- 클래스, 생성자, 메서드를 활용하여 객체 지향 프로그래밍 구현
- 상속, 인터페이스, 믹스인을 사용하여 코드 재사용성 및 확장성 향상
- 캡슐화, 다형성, 제네릭을 통해 안정적이고 유연한 코드 작성

객체 지향 프로그래밍 기초

- 클래스는 객체를 생성하기 위한 설계도 또는 템플릿
- 객체는 클래스의 인스턴스이며, 실제 데이터와 메서드를 포함
- 상속은 기존 클래스의 속성과 메서드를 재사용하고 확장하는 메커니즘
- 인터페이스는 클래스가 구현해야 하는 메서드 시그니처를 정의
- 믹스인은 클래스에 기능을 추가하는 재사용 가능한 코드 블록

클래스 정의와 생성자

- 클래스는 `class` 키워드를 사용하여 정의
- 멤버 변수(필드)와 메서드를 포함하여 객체의 상태와 행동을 정의
- 생성자는 객체 생성 시 초기화를 담당하며, 클래스 이름과 동일한 이름을 가짐
- `late` 키워드로 나중에 초기화될 변수 선언 가능 (null safety)
- 생성자 매개변수에 `this` 키워드로 멤버 변수 직접 초기화 가능 (syntactic sugar)

상속과 확장

- `extends` 키워드를 사용하여 기존 클래스 확장 (부모 클래스, 슈퍼 클래스)
- 부모 클래스의 속성과 메서드를 상속받아 재사용
- `@override` 애노테이션으로 메서드 재정의 (Overriding)
- `super` 키워드로 부모 클래스의 생성자 또는 메서드 호출
- 다중 상속은 지원하지 않음 (믹스인을 통해 유사한 기능 구현 가능)

인터페이스 활용

- `implements` 키워드로 인터페이스 구현
- 다중 인터페이스 구현 가능
- 인터페이스에 정의된 모든 추상 메서드 구현 필수
- 인터페이스를 통한 다형성 구현 (하나의 인터페이스로 여러 구현체 사용)
- 코드의 유연성과 확장성 향상

Book.dart

기본적인 도서 정보를 관리하는 클래스 예제입니다.

```
class Book {
  String title;
  String author;
  String publisher;
}
```

DaysLeftInWeek.dart

주중 남은 날짜를 계산하는 클래스 예제입니다.

```dart
const numDays = 7;

class DaysLeftInWeek {
  int currentDay = 0;

  // 생성자로 객체 인스턴스를 초기화
  DaysLeftInWeek() {
    currentDay = DateTime.now().weekday;
  }

  int howManyDaysLeft() {
    return numDays - currentDay;
  }
}

void main() {
  var daysLeft = DaysLeftInWeek();
  print('Days left in the week: ${daysLeft.howManyDaysLeft()}');
}
```

Media.dart

미디어 타입을 관리하는 추상 클래스 정의입니다.

```
abstract class Media {
  late String myId;
  late String myTitle;
  late String myType;

  void setMediaTitle(String mediaTitle);
  String getMediaTitle();

  void setMediaType(String mediaType);
  String getMediaType();

  void setMediaId(String mediaId);
  String getMediaId();
}
```

BookImplementation.dart

Media 인터페이스를 구현한 도서 클래스입니다.

```dart
class Book implements Media {
  @override
  late String myId;
  @override
  late String myTitle;
  @override
  late String myType;

  Book(String mediaTitle, String mediaType, String mediaId) {
    myTitle = mediaTitle;
    myType = mediaType;
    myId = mediaId;
  }

  @override
  void setMediaTitle(String mediaTitle) => myTitle = mediaTitle;

  @override
  String getMediaTitle() => myTitle;

  @override
  void setMediaType(String mediaType) => myType = mediaType;

  @override
  String getMediaType() => myType;

  @override
  void setMediaId(String mediaId) => myId = mediaId;

  @override
  String getMediaId() => myId;
}

void main() {
  var book = Book('The Lord of the Rings', 'Book', '12345');
  print('Title: ${book.getMediaTitle()}');
  print('Type: ${book.getMediaType()}');
  print('ID: ${book.getMediaId()}');
}
```

믹스인 활용

- `with` 키워드를 사용하여 믹스인 적용
- 클래스에 여러 믹스인을 동시에 사용 가능 (다중 상속의 대안)
- 코드 재사용성 극대화 및 기능의 모듈화
- 믹스인은 클래스의 기능을 확장하는 재사용 가능한 코드 블록
- 믹스인은 상태(멤버 변수)를 가질 수 있음

ChocolateBar.dart

믹스인을 활용한 초콜릿 바 클래스 구현입니다.

```dart
abstract class BaseChocolate {
  bool hasChocolate = true;
}

mixin HasNuts {
  bool hasHazelnut = true;
  bool hasAlmond = false;
}

mixin HasRice {
  bool hasRice = true;
}

class ChocolateBar extends BaseChocolate with HasNuts, HasRice {
  List<String> ingredients = [];

  ChocolateBar() {
    if (hasChocolate) ingredients.add('Chocolate');
    if (hasHazelnut) ingredients.add('Hazelnut');
    if (hasRice) ingredients.add('Rice');
    if (hasAlmond) ingredients.add('Almond');
  }

  List<String> getIngredients() => ingredients;

  void displayIngredients() {
    print('Ingredients: ${ingredients.join(", ")}');
  }
}

void main() {
  var chocolateBar = ChocolateBar();
  chocolateBar.displayIngredients();
}
```

캡슐화와 접근 제어

- 변수명 앞에 `_` (underscore) 를 붙여 private 선언 (library-private)
- getter와 setter를 통한 접근 제어 (데이터 은닉)
- 데이터 은닉과 보안성 강화
- 내부 구현 세부사항 숨김 (정보 은닉)
- 코드의 안정성 및 유지보수성 향상

```dart
class BankAccount {
  String _accountNumber; // private 멤버 변수
  double _balance = 0.0;

  BankAccount(this._accountNumber);

  String get accountNumber => _accountNumber; // getter

  double get balance => _balance; // getter

  void deposit(double amount) {
    if (amount > 0) {
      _balance += amount;
      print('Deposited: \$$amount');
      print('New balance: \$$_balance');
    }
  }

  bool withdraw(double amount) {
    if (amount > 0 && _balance >= amount) {
      _balance -= amount;
      print('Withdrawn: \$$amount');
      print('New balance: \$$_balance');
      return true;
    }
    print('Insufficient funds');
    return false;
  }
}

void main() {
  var account = BankAccount('123-456-789');
  print('Account Number: ${account.accountNumber}');
  print('Balance: \$${account.balance}');
  account.deposit(100);
  account.withdraw(50);
  account.withdraw(100);
}
```

EncapsulationExample.dart

캡슐화를 적용한 클래스 구현입니다.

```dart
class BankAccount {
  String _accountNumber;
  double _balance = 0.0;

  BankAccount(this._accountNumber);

  String get accountNumber => _accountNumber;

  double get balance => _balance;

  void deposit(double amount) {
    if (amount > 0) {
      _balance += amount;
      print('Deposited: \$$amount');
      print('New balance: \$$_balance');
    }
  }

  bool withdraw(double amount) {
    if (amount > 0 && _balance >= amount) {
      _balance -= amount;
      print('Withdrawn: \$$amount');
      print('New balance: \$$_balance');
      return true;
    }
    print('Insufficient funds');
    return false;
  }
}
```

다형성 활용

- 하나의 인터페이스 또는 추상 클래스로 여러 구현체 사용
- 유연한 코드 설계 가능 (OCP: Open/Closed Principle)
- 코드 재사용성 향상 및 확장성 있는 프로그램 구조 구현
- 유지보수 용이성 증가

다형성 활용: 도형 클래스 1/2

```
abstract class Shape {
  double getArea();
  double getPerimeter();
  void draw();
}

class Circle implements Shape {
  final double radius;

  Circle(this.radius);

  @override
  double getArea() => 3.14 * radius * radius;

  @override
  double getPerimeter() => 2 * 3.14 * radius;

  @override
  void draw() {
    print('Drawing Circle with radius: $radius');
  }
}
```

다형성 활용: 도형 클래스 2/2

```
class Rectangle implements Shape {
  final double width;
  final double height;

  Rectangle(this.width, this.height);

  @override
  double getArea() => width * height;

  @override
  double getPerimeter() => 2 * (width + height);

  @override
  void draw() {
    print('Drawing Rectangle: $width x $height');
  }
}

void main() {
  List<Shape> shapes = [Circle(5), Rectangle(10, 5)];

  for (var shape in shapes) {
    shape.draw();
    print('Area: ${shape.getArea()}');
    print('Perimeter: ${shape.getPerimeter()}');
  }
}
```

제네릭 활용

- 타입 안정성 보장 (컴파일 시 타입 체크)
- 코드 재사용성 향상 (다양한 타입에 대해 동일한 로직 적용)
- 유연한 데이터 구조 구현 (List, Map 등)
- 타입 캐스팅 오류 방지

```dart
class DataStorage<T> {
  List<T> _items = [];

  void addItem(T item) {
    _items.add(item);
  }

  void removeItem(T item) {
    _items.remove(item);
  }

  T? getItem(int index) {
    if (index >= 0 && index < _items.length) {
      return _items[index];
    }
    return null;
  }

  List<T> getAllItems() => List.from(_items);

  int get count => _items.length;

  void clear() {
    _items.clear();
  }
}

void main() {
  var stringStorage = DataStorage<String>();
  stringStorage.addItem('Hello');
  stringStorage.addItem('World');
  print('Items: ${stringStorage.getAllItems()}');

  var intStorage = DataStorage<int>();
  intStorage.addItem(1);
  intStorage.addItem(2);
  print('Items: ${intStorage.getAllItems()}');
}
```

Factory 생성자

- 객체 생성을 캡슐화하고, 생성 로직을 유연하게 관리
- 싱글톤 패턴 구현에 유용
- 캐시된 인스턴스 반환 가능
- 서브클래스 인스턴스 생성 가능
- 객체 생성 로직을 클래스 외부로 분리

```dart
class DatabaseConnection {
  static DatabaseConnection? _instance; // private static 인스턴스
  final String _connectionString;

  DatabaseConnection._internal(this._connectionString) {
    print('Creating new database connection');
  }

  factory DatabaseConnection(String connectionString) {
    _instance ??= DatabaseConnection._internal(connectionString);
    return _instance!;
  }

  void connect() {
    print('Connecting to database: $_connectionString');
  }

  void disconnect() {
    print('Disconnecting from database');
  }

  void query(String sql) {
    print('Executing query: $sql');
  }
}

void main() {
  var db1 = DatabaseConnection('localhost:5432');
  var db2 = DatabaseConnection('localhost:5432'); // 동일한 인스턴스 반환

  print(identical(db1, db2)); // 출력: true
  db1.connect();
  db1.query('SELECT * FROM users');
  db1.disconnect();
}
```

표준 라이브러리 활용

- 컬렉션 클래스 활용 (List, Set, Map)
- 유틸리티 함수 사용 (Math, DateTime)
- 입출력 처리 (File, Socket)
- 날짜/시간 처리 (DateTime, Duration)
- 수학 연산 지원 (Math)
- 문자열 처리 (String)

```dart
import 'dart:math';

void main() {
  // List
  List<int> numbers = [1, 2, 3, 4, 5];
  print('Sum: ${numbers.reduce((a, b) => a + b)}');

  // Map
  Map<String, String> capitals = {'USA': 'Washington D.C.', 'Korea': 'Seoul'};
  print('Capital of USA: ${capitals['USA']}');
```

요약

- 다트는 객체 지향 프로그래밍을 완벽하게 지원하는 현대적인 프로그래밍 언어입니다.
- 클래스, 상속, 인터페이스, 믹스인 등 다양한 객체 지향 기능을 제공하여 유연하고 확장 가능한 코드 작성이 가능합니다.
- 캡슐화, 다형성, 제네릭을 통해 안정적이고 재사용 가능한 코드를 작성할 수 있습니다.

END