DARTT PROMMIMIG.GINGRANGAVE:. ADVNCE)

dart

DART PROGRAMMING LANGUAGE:

ADVANCE

(Integrative-Generative AI Edition)

STUDENT PRICE BOOK CENTER

คำนำ

ในโลกของการพัฒนาโปรแกรมสมัยใหม่ การเขียนโค้ดที่มีประสิทธิภาพ ยืดหยุ่น และปลอดภัย เป็นสิ่งจำเป็นสำหรับนักพัฒนาที่ต้องการสร้างซอฟต์แวร์คุณภาพสูง ภาษา Dart ได้รับความนิยมอย่างรวดเร็ว โดยเฉพาะในการพัฒนาแอปพลิเคชันข้ามแพลตฟอร์มร่วมกับ Flutter ด้วยความเรียบง่าย แต่ทรงพลัง Dart สามารถรองรับทั้ง OOP, Functional Programming และ Asynchronous Programming ได้อย่างครบถ้วน

หนังสือเล่มนี้ **Dart Programming Language: Advance** ถูกออกแบบมาสำหรับผู้ที่มีพื้นฐาน Dart แล้วและต้องการต่อยอดสู่การใช้งานขั้นสูง ในแต่ละบทจะเน้นทั้งการเข้าใจเชิงลึกและการประยุกต์ใช้งานจริง พร้อมตัวอย่างบูรณาการที่ช่วยให้ผู้อ่านสามารถนำแนวคิดไปใช้พัฒนาโปรแกรมที่ซับซ้อนได้อย่างมั่นใจ

บทที่ 12 **OOP ขั้นสูง (Advanced OOP)** จะพาผู้อ่านเจาะลึกแนวคิดเชิงวัตถุขั้นสูง ครอบคลุม **Inheritance**, **Abstract Classes**, **Interfaces**, **Method Overriding**, **Static Members**, และ **Factory Constructors** รวมถึงตัวอย่างบูรณาการเพื่อให้ผู้อ่านสามารถออกแบบโครงสร้าง class ที่ซับซ้อน ยืดหยุ่น และง่ายต่อการบำรุงรักษา

บทที่ 13 **Generics** แนะนำการใช้ Generics กับ class และ function พร้อม **Constraints** ด้วยคำสั่ง extends เพื่อเพิ่ม type safety และความยืดหยุ่นในการจัดการ collections ช่วยให้โค้ดสามารถรองรับหลายชนิดข้อมูลโดยไม่สูญเสียความถูกต้องและความปลอดภัยของ type

บทที่ 14 **Asynchronous Programming** เจาะลึกการเขียนโปรแกรมแบบ asynchronous ใน Dart ครอบคลุมการใช้งาน **Future** ร่วมกับ **async/await**, การจัดการผลลัพธ์ด้วย **then()** และ **catchError()**, การหน่วงเวลาโดย **Future.delayed**, การใช้งาน **Stream**, และ *async/yield** เพื่อสร้าง asynchronous generator บทนี้ช่วยให้ผู้อ่านพัฒนาโปรแกรมที่ตอบสนองรวดเร็วและจัดการงานเบื้องหลังได้อย่างมั่นใจ

บทที่ 15 **Mixins และ Extension Methods** นำเสนอวิธีการเพิ่มความสามารถให้ class โดยไม่ต้องแก้ไขโครงสร้างหลักของ class **Mixins** ช่วยแชร์พฤติกรรมระหว่าง class หลายตัวอย่างยืดหยุ่น ขณะที่ **Extension Methods** ช่วยเพิ่ม method หรือความสามารถให้ class ที่มีอยู่แล้วโดยไม่แก้ไข source code เทคนิคเหล่านี้ช่วยให้โค้ด modular, reusable และ maintainable

บทที่ 16 **Functional Programming Features** ครอบคลุมแนวคิด functional programming ใน Dart เช่น **map, where, reduce, fold**, การทำ **chain calls** และ **lambda expressions** ช่วยให้โค้ดกระชับ, declarative, และลดข้อผิดพลาด บทเรียนนี้ช่วยให้ผู้อ่านประมวลผลข้อมูลได้อย่างยืดหยุ่นและสร้างโปรแกรมที่ maintainable และ scalable

หนังสือเล่มนี้มุ่งหวังที่จะเป็นคู่มือสำหรับนักพัฒนาที่ต้องการก้าวสู่ระดับสูงของ Dart ทั้งในด้าน **OOP ขั้นสูง, Generics, Asynchronous Programming, Mixins, Extension Methods และ Functional Programming** การศึกษาผ่านตัวอย่างและการประยุกต์ใช้งานจริงจะช่วยให้ผู้อ่านสามารถเขียน

โปรแกรมที่มีคุณภาพสูง ปลอดภัย และยืดหยุ่น พร้อมรับมือกับความซับซ้อนของโปรแกรมสมัยใหม่ได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 12
OOP ขั้นสูง
(Advanced OOP)

เนื้อหา

- เจาะลึกเนื้อหาของ Object-Oriented Programming (OOP) ขั้นสูง
- Inheritance (การสืบทอด)
- Abstract Classes
- Interfaces (implements)
- Method Overriding (การเขียนทับเมธอด)
- Static Members
- Factory Constructors
- ตัวอย่างบูรณาการ

บทนำ

เมื่อการพัฒนาโปรแกรมซับซ้อนขึ้น การเข้าใจและใช้งานแนวคิด **Object-Oriented Programming (OOP) ขั้นสูง** จะช่วยให้นักพัฒนาสามารถออกแบบโค้ดที่ยืดหยุ่น นำกลับมาใช้ซ้ำได้ และง่ายต่อการบำรุงรักษา บทนี้จะเจาะลึกการใช้คุณสมบัติขั้นสูงของ OOP ในภาษา Dart ซึ่งเป็นก้าวต่อยอดจากพื้นฐาน OOP ที่ผู้เรียนได้ศึกษามาแล้ว

หนึ่งในแนวคิดสำคัญคือ **Inheritance (extends)** ซึ่งช่วยให้ class ใหม่สามารถสืบทอดคุณสมบัติและพฤติกรรมจาก class เดิมได้ ทำให้โค้ดลดความซ้ำซ้อนและเพิ่มความสามารถในการนำกลับมาใช้ซ้ำ ผู้เรียนจะได้เรียนรู้วิธีการสร้าง subclass และการออกแบบ class hierarchy อย่างมีประสิทธิภาพ

Abstract Classes เป็นอีกเครื่องมือสำคัญที่ใช้กำหนดแม่แบบของ class ที่ไม่สามารถสร้าง object โดยตรงได้ แต่สามารถกำหนด method ที่ subclass ต้อง implement เอง การใช้ abstract class ช่วยให้การออกแบบโปรแกรมมีความชัดเจนและบังคับให้ subclass ปฏิบัติตามสัญญาที่กำหนดไว้

นอกจากนี้ Dart ยังสนับสนุน **Interfaces (implements)** ซึ่งช่วยให้ class สามารถสืบทอด method signature จากหลายแหล่งโดยไม่ต้องสืบทอด implementation โดยตรง ทำให้สามารถสร้างสถาปัตยกรรมโปรแกรมที่ยืดหยุ่นและโมดูลาร์มากขึ้น

การปรับแต่งพฤติกรรมของ subclass ทำได้ด้วย **Method Overriding** ซึ่งอนุญาตให้ subclass เขียนทับ (override) method ของ superclass เพื่อให้ทำงานแตกต่างหรือเพิ่มเติมตามความต้องการ เทคนิคนี้ช่วยให้ polymorphism ทำงานได้อย่างเต็มประสิทธิภาพ

นอกจากนี้ บทนี้ยังครอบคลุม **Static Members** ซึ่งเป็น fields หรือ methods ที่อยู่ในระดับ class แทนที่จะเป็น object ทำให้สามารถเข้าถึงโดยไม่ต้องสร้าง instance ของ class และ **Factory Constructors** ที่ช่วยควบคุมการสร้าง object เช่น การคืนค่าจาก cache หรือ singleton pattern

บทที่ 12 นี้มีเป้าหมายเพื่อให้ผู้อ่านเข้าใจแนวคิด OOP ขั้นสูงและสามารถนำไปประยุกต์ใช้ในโปรแกรม Dart ขนาดใหญ่ ทั้งในแง่ของการออกแบบ class, การจัดการ inheritance, การสร้าง interface, และการใช้ method overriding, static members, และ factory constructors อย่างเหมาะสม เพื่อสร้างซอฟต์แวร์ที่ยืดหยุ่น แข็งแรง และง่ายต่อการดูแลรักษา

OOP ขั้นสูง

- Inheritance (extends)
- Abstract Classes
- Interfaces (implements)
- Method Overriding
- Static Members
- Factory Constructors

Inheritance (การสืบทอด)

Inheritance คือกลไกที่อนุญาตให้ Class หนึ่ง (subclass หรือ child class) สามารถ **สืบทอด** คุณสมบัติและพฤติกรรมต่างๆ จากอีก Class หนึ่ง (superclass หรือ parent class) ได้ โดยใช้คีย์เวิร์ด **extends**

- **หลักการ:** Subclass จะได้ Fields และ Methods ทั้งหมดของ Superclass มาโดยอัตโนมัติ ทำให้เราสามารถนำโค้ดกลับมาใช้ใหม่ได้ (code reusability)
- **ตัวอย่าง:**

Dart

```
class Animal {
  void eat() {
    print('Animal is eating.');
  }
}

class Dog extends Animal {
```

```
  void bark() {
    print('Dog is barking.');
  }
}

void main() {
  var myDog = Dog();
  myDog.eat();  // สืบทอดมาจาก Animal
  myDog.bark(); // เป็น Method ของ Dog เอง
}
```

Abstract Classes

Abstract Class คือ Class ที่ไม่สามารถสร้าง Object ขึ้นมาได้โดยตรง มีไว้เพื่อให้เป็นแม่แบบสำหรับ Class อื่นๆ ที่จะมาสืบทอดไปใช้งาน โดยใช้คีย์เวิร์ด **abstract**

- **คุณสมบัติ:**
  - สามารถมี **Abstract Methods** (เมธอดที่ไม่มีเนื้อหา) ได้
  - Class ที่สืบทอด Abstract Class จะต้อง **implement** เมธอดที่เป็น abstract ทั้งหมด
- **ตัวอย่าง:**

Dart

```
abstract class Shape {
  // Abstract Method: ทุก Class ที่สืบทอดต้องมี Method นี้
  double calculateArea();
}

class Circle extends Shape {
  double radius;
  Circle(this.radius);

  // ต้อง implement Method จาก Abstract Class
  @override
  double calculateArea() {
    return 3.14 * radius * radius;
  }
}
```

Interfaces (การเชื่อมต่อ)

ใน Dart ทุก Class เป็น **Interface** โดยอัตโนมัติ การใช้คีย์เวิร์ด **implements** จะบังคับให้ Class ที่ทำการเชื่อมต่อ ต้องสร้าง Fields และ Methods ทั้งหมดของ Interface นั้นๆ ขึ้นมาเอง

- **หลักการ:** Interfaces ใช้สำหรับการกำหนด "สัญญา" (contract) ว่า Class นั้นๆ จะต้องมีพฤติกรรมอะไรบ้าง
- **ความแตกต่างกับ extends:** extends ใช้เพื่อสืบทอดคุณสมบัติ ส่วน implements ใช้เพื่อบังคับให้สร้างโค้ดขึ้นมาใหม่
- **ตัวอย่าง:**

Dart

```
class Greetable {
  void greet();
}

class Person implements Greetable {
  String name;
  Person(this.name);

  @override
  void greet() {
    print('Hello, my name is $name.');
  }
}
```

Method Overriding (การเขียนทับเมธอด)

Method Overriding คือการที่ Subclass สร้าง Method ที่มีชื่อและประเภทเหมือนกับ Method ใน Superclass เพื่อเปลี่ยนแปลงหรือขยายพฤติกรรมเดิม

- **หลักการ:** ใช้คำสั่ง **@override** เพื่อบอกคอมไพเลอร์ว่าเรากำลังจะเขียนทับ Method เดิม
- **ตัวอย่าง:**

Dart

```
class Animal {
  void makeSound() {
    print('An animal makes a sound.');
  }
```

```
}

class Cat extends Animal {
  @override
  void makeSound() {
    print('The cat meows.');
  }
}
```

Static Members

Static Members คือ Fields และ Methods ที่เป็นของ **Class** ไม่ใช่ของ Object แต่ละตัว เราสามารถเรียกใช้ Static Members ได้โดยตรงผ่านชื่อ Class โดยไม่ต้องสร้าง Object ขึ้นมาก่อน

- **หลักการ:** ใช้คีย์เวิร์ด **static**
- **ตัวอย่าง:**

Dart

```
class MathUtils {
  // Static Field
  static const double pi = 3.14159;

  // Static Method
  static double calculateArea(double radius) {
    return pi * radius * radius;
  }
}

void main() {
  // เรียกใช้ผ่านชื่อ Class
  print(MathUtils.pi);
  print(MathUtils.calculateArea(5));
}
```

Factory Constructors

Factory Constructors คือ Constructor ที่ไม่ได้สร้าง Object ใหม่เสมอไป แต่สามารถคืนค่า Object ที่มีอยู่แล้วได้ มีประโยชน์ในการสร้าง Object ที่มี Logic ซับซ้อน

- **หลักการ:** ใช้คีย์เวิร์ด **factory**
- **ตัวอย่าง:**

Dart

```
class Logger {
  static final Map<String, Logger> _cache = <String, Logger>{};

  // Factory Constructor
  factory Logger(String name) {
    // คืนค่า Object ที่มีอยู่แล้ว ถ้ามี
    return _cache.putIfAbsent(name, () => Logger._internal(name));
  }

  // Private Constructor สำหรับสร้าง Object จริงๆ
  Logger._internal(this.name);

  final String name;
}
```

เจาะลึกเนื้อหาของ Object-Oriented Programming (OOP) ขั้นสูง

Inheritance (การสืบทอด)

Inheritance เป็นกลไกที่ช่วยให้ Class หนึ่ง (subclass) สามารถรับเอาคุณสมบัติและพฤติกรรมจากอีก Class หนึ่ง (superclass) ได้โดยอัตโนมัติ โดยใช้คีย์เวิร์ด **extends**

- **หลักการ:** subclass จะเป็น superclass ด้วยเสมอ (is-a relationship) เช่น Dog เป็น Animal และ Car เป็น Vehicle การสืบทอดทำให้เราสามารถเขียนโค้ดซ้ำ (code duplication) ได้น้อยลง
- **การทำงานเชิงลึก:** เมื่อคุณสร้าง Object ของ subclass ตัว Dart Virtual Machine (DVM) จะทำการสร้าง Object ของ superclass ก่อน จากนั้นจึงเพิ่ม Fields และ Methods ของ subclass เข้าไป
  - **Constructor:** เมื่อสร้าง subclass Constructor ของ superclass จะถูกเรียกใช้ก่อนเสมอ ซึ่งเป็นเหตุผลว่าทำไมคุณต้องใช้ super เพื่อส่งค่าไปยัง Constructor ของ Class แม่
  - **Single Inheritance:** Dart รองรับการสืบทอดจาก Class เดียวเท่านั้น (single inheritance) เพื่อป้องกันปัญหาที่ซับซ้อนที่เรียกว่า "Diamond Problem"

Abstract Classes

Abstract Class คือ Class ที่เป็นเพียงแม่แบบ (blueprint) และไม่สามารถสร้าง Object ได้โดยตรง (instantiate) โดยใช้คีย์เวิร์ด **abstract**

- **หลักการ:** Abstract Class ถูกสร้างขึ้นมาเพื่อกำหนดโครงสร้างที่ชัดเจนให้กับ Class ลูกที่จะมาสืบทอด
- **คุณสมบัติ:**
  - สามารถมี Fields และ Methods ที่มีเนื้อหา (Concrete Methods) ได้เหมือน Class ปกติ
  - สามารถมี Abstract Methods ซึ่งเป็น Method ที่ไม่มีเนื้อหาและลงท้ายด้วยเครื่องหมาย ; เท่านั้น
  - Class ลูกที่ **extends** Abstract Class จะถูกบังคับให้ต้องเขียนเนื้อหาให้กับ Abstract Methods ทั้งหมด เพื่อให้โค้ดมีความสมบูรณ์
- **ประโยชน์:** ใช้ในการกำหนดสัญญา (contract) ที่ Class ลูกต้องทำตาม แต่ยังคงมีความสามารถในการนำโค้ดกลับมาใช้ใหม่จาก Concrete Methods ได้

Interfaces (การเชื่อมต่อ)

ใน Dart ทุก Class เป็น **Interface** โดยปริยาย การใช้คีย์เวิร์ด **implements** หมายถึงการที่ Class นั้นๆ ต้องทำตาม "สัญญา" ของ Interface อย่างครบถ้วน

- **หลักการ:** Interface ไม่ได้เน้นการนำโค้ดกลับมาใช้ใหม่ แต่เน้นที่การกำหนดพฤติกรรม
- **การทำงานเชิงลึก:** เมื่อ Class หนึ่ง implements อีก Class หนึ่ง คอมไพเลอร์จะบังคับให้ต้องสร้าง Fields และ Methods ทั้งหมดของ Interface ขึ้นมาใหม่ทั้งหมด (@override)
  - **Multiple Interfaces:** Dart อนุญาตให้ Class หนึ่งสามารถ implements ได้หลาย Interfaces ซึ่งเป็นวิธีที่แก้ปัญหาการสืบทอดจากหลาย Class
- **ความแตกต่างระหว่าง extends และ implements:**
  - extends คือการนำโค้ดของ Class แม่มาใช้ซ้ำและเพิ่มส่วนของตัวเอง
  - implements คือการทำตามสัญญาของ Interface และเขียนโค้ดขึ้นมาใหม่ทั้งหมด

Method Overriding (การเขียนทับเมธอด)

Method Overriding คือการที่ subclass สร้าง Method ที่มีชื่อ, จำนวนพารามิเตอร์, และชนิดข้อมูลของพารามิเตอร์เหมือนกับ Method ใน superclass เพื่อเปลี่ยนพฤติกรรมเดิม

- **หลักการ:** ใช้คำสั่ง **@override** เพื่อบอกคอมไพเลอร์ว่าเรากำลังเขียนทับ Method เดิม
- **การทำงานเชิงลึก:** เมื่อมีการเรียกใช้ Method ที่ถูก Overridden โปรแกรมจะเลือกใช้ Method ของ Class ที่เป็นประเภทของ Object นั้นๆ (polymorphism)
  - **super**: คุณสามารถใช้คีย์เวิร์ด super ภายใน Method ที่ถูก Overridden เพื่อเรียกใช้ Method เดิมของ superclass ได้ ซึ่งมีประโยชน์ในการขยายการทำงานเดิมแทนการเขียนทับทั้งหมด

Static Members

Static Members คือ Fields และ Methods ที่เป็นของ **Class** ไม่ใช่ของ Object แต่ละตัว

- **หลักการ:** ใช้คีย์เวิร์ด **static**
- **การทำงานเชิงลึก:** Static Members จะถูกจัดสรรในหน่วยความจำเพียงครั้งเดียวเมื่อโปรแกรมเริ่มทำงาน ไม่ว่าคุณจะสร้าง Object กี่ตัวก็ตาม
  - **Fields:** static Field จะมีค่าเพียงค่าเดียวที่ถูกใช้ร่วมกันทั้ง Class
  - **Methods:** static Method ไม่สามารถเข้าถึง Fields และ Methods ที่ไม่เป็น static ได้ เพราะมันไม่ได้ผูกติดอยู่กับ Object ใดๆ
- **การใช้งาน:** เหมาะสำหรับค่าคงที่ (constants) หรือฟังก์ชันอรรถประโยชน์ (utility functions) ที่ไม่จำเป็นต้องอาศัยสถานะของ Object เช่น Math.pi หรือ DateTime.now()

Factory Constructors

Factory Constructors คือ Constructor ที่ไม่ได้มีไว้เพื่อสร้าง Object ใหม่เสมอไป แต่มีไว้เพื่อควบคุมการสร้าง Object ที่ซับซ้อนและมี Logic พิเศษ

- **หลักการ:** ใช้คีย์เวิร์ด **factory**
- **ความแตกต่างกับ Generative Constructor:**
  - **Generative Constructor** (ClassName()) ต้องคืนค่า Object ใหม่เสมอ
  - **Factory Constructor** สามารถคืนค่า Object ที่มีอยู่แล้วได้
- **การใช้งาน:**
  - **Singleton Pattern:** ใช้เพื่อรับประกันว่าจะมี Object ของ Class นั้นๆ เพียงตัวเดียวในโปรแกรม
  - **Caching:** คืนค่า Object ที่เคยสร้างไปแล้วเพื่อประหยัดหน่วยความจำ
  - **การสร้าง Object ที่มี Logic ซับซ้อน:** เช่น การสร้าง Object จากข้อมูล JSON
  - **Polymorphism:** สามารถคืนค่า Object ของ subclass ได้

แนวคิดเหล่านี้เป็นรากฐานที่สำคัญที่ทำให้คุณสามารถเขียนโค้ดที่มีโครงสร้างดี, ยืดหยุ่น และง่ายต่อการขยายในอนาคต

Inheritance (การสืบทอด)

Inheritance คือแนวคิดพื้นฐานใน Object-Oriented Programming (OOP) ที่ช่วยให้ Class หนึ่ง (subclass) สามารถรับคุณสมบัติและพฤติกรรมจากอีก Class หนึ่ง (superclass) ได้โดยอัตโนมัติ ทำให้โค้ดสามารถนำกลับมาใช้ใหม่ได้ (code reusability)

หลักการทำงาน

การสืบทอดในภาษา Dart ใช้คีย์เวิร์ด **extends** โดยมีความสัมพันธ์แบบ "เป็น" (is-a relationship) ซึ่งหมายความว่า Object ของ Class ลูกก็คือ Object ของ Class แม่ด้วย

- **Subclass (คลาสลูก)**: Class ที่สืบทอดมาจาก Class อื่น
- **Superclass (คลาสแม่)**: Class ที่ถูกสืบทอด

ตัวอย่าง:

สมมติว่าเรามี Class Animal ซึ่งเป็น Class แม่ และ Class Dog ซึ่งเป็น Class ลูกที่สืบทอดมาจาก Animal

Dart

```
class Animal {
  // Field (คุณสมบัติ)
  String species;

  // Method (พฤติกรรม)
  void eat() {
    print('$species is eating.');
  }
}

class Dog extends Animal {
  // เพิ่ม Field เฉพาะของ Dog
  String breed;

  // เพิ่ม Method เฉพาะของ Dog
  void bark() {
    print('The $breed dog is barking.');
  }
}
```

ในตัวอย่างนี้ Dog จะได้รับ Field species และ Method eat() จาก Animal โดยอัตโนมัติ และยังสามารถเพิ่ม Fields และ Methods ของตัวเองได้อีกด้วย

การเรียกใช้

Dart

```
void main() {
  // สร้าง Object ของ Class ลูก (Dog)
  var myDog = Dog();

  // เข้าถึง Fields และ Methods ของ Class แม่ที่สืบทอดมา
```

```
  myDog.species = 'Canine';
  myDog.eat(); // Output: Canine is eating.

  // เข้าถึง Fields และ Methods ของ Class ลูกเอง
  myDog.breed = 'Golden Retriever';
  myDog.bark(); // Output: The Golden Retriever dog is barking.
}
```

Constructor และ super

เมื่อคุณสร้าง Constructor ของ Class ลูก คุณมักจะต้องเรียก Constructor ของ Class แม่ด้วย โดยใช้คีย์เวิร์ด **super** เพื่อส่งค่าเริ่มต้นให้กับ Class แม่ก่อน

Dart

```
class Animal {
  String species;

  Animal(this.species); // Constructor ของ Class แม่
}

class Dog extends Animal {
  String breed;

  // Constructor ของ Class ลูก เรียกใช้ super เพื่อส่งค่าให้ Class แม่
  Dog(String species, this.breed) : super(species);
}

void main() {
  var myDog = Dog('Canine', 'Golden Retriever');
  print('Species: ${myDog.species}'); // Output: Species: Canine
  print('Breed: ${myDog.breed}');    // Output: Breed: Golden Retriever
}
```

ในตัวอย่างนี้ super(species) จะทำหน้าที่เรียก Constructor ของ Animal เพื่อกำหนดค่าให้กับ Field species ก่อนที่ Constructor ของ Dog จะทำงานต่อ

Inheritance (การสืบทอด) ที่เราได้เรียนรู้กัน ผมได้เตรียมตัวอย่างโปรแกรมทั้งแบบพื้นฐานและแบบประยุกต์มาให้คุณครับ เพื่อให้คุณเห็นภาพการนำคุณสมบัติจาก Class แม่มาใช้ใน Class ลูกได้อย่างชัดเจน

---

1. ตัวอย่างโปรแกรมพื้นฐาน

โปรแกรมเหล่านี้จะเน้นการสาธิตการใช้ extends เพื่อสืบทอด Fields และ Methods ในรูปแบบง่ายๆ

โปรแกรมที่ 1: การสืบทอดจาก Class สัตว์

โครงสร้างไฟล์: main.dart

คำอธิบายโค้ด:

โปรแกรมนี้สร้าง Class Animal เป็น Class แม่ที่มี Method eat() และสร้าง Class Dog เป็น Class ลูกที่ extends มา ซึ่งทำให้ Dog สามารถเรียกใช้ Method eat() ได้โดยไม่ต้องเขียนโค้ดซ้ำ

โค้ดเต็มไฟล์:

Dart

```dart
class Animal {
  String name;

  Animal(this.name);

  void eat() {
    print('$name is eating.');
  }
}

class Dog extends Animal {
  Dog(String name) : super(name);

  void bark() {
    print('$name is barking.');
  }
}

void main() {
  var myDog = Dog('Buddy');
```

```
  // เรียกใช้ Method ที่สืบทอดมาจาก Animal
  myDog.eat();

  // เรียกใช้ Method ของ Dog เอง
  myDog.bark();
}
```

ผลการรัน:

```
Buddy is eating.
Buddy is barking.
```

โปรแกรมที่ 2: การสืบทอดจาก Class Shape

โครงสร้างไฟล์: main.dart

คำอธิบายโค้ด:

โปรแกรมนี้สร้าง Class Shape เป็น Class แม่ที่มี Field color และสร้าง Class Circle เป็น Class ลูกที่ extends มา ซึ่งทำให้ Circle สามารถเข้าถึงคุณสมบัติเรื่องสีได้ทันที

โค้ดเต็มไฟล์:

Dart

```
class Shape {
  String color;

  Shape(this.color);
}

class Circle extends Shape {
  double radius;

  Circle(String color, this.radius) : super(color);

  double calculateArea() {
    return 3.14 * radius * radius;
  }
}

void main() {
```

```
  var myCircle = Circle('Red', 5.0);

  // เข้าถึง Field ที่สืบทอดมา
  print('สีของวงกลม: ${myCircle.color}');

  // เรียกใช้ Method ของ Circle เอง
  print('พื้นที่ของวงกลม: ${myCircle.calculateArea()}');
}
```

ผลการรัน:

สีของวงกลม: Red
พื้นที่ของวงกลม: 78.5

โปรแกรมที่ 3: การสืบทอด Constructor และ Method

โครงสร้างไฟล์: main.dart

คำอธิบายโค้ด:

โปรแกรมนี้สร้าง Class Vehicle เป็น Class แม่ และ Class Car เป็น Class ลูกที่ extends มา โดย Car จะเรียกใช้ super ใน Constructor เพื่อกำหนดค่าให้กับ Class แม่ และมีการใช้ super ใน Method เพื่อเรียกใช้ Method ของ Class แม่ด้วย

โค้ดเต็มไฟล์:

Dart

```
class Vehicle {
  String brand;

  Vehicle(this.brand);

  void start() {
    print('$brand engine is started.');
  }
}

class Car extends Vehicle {
  Car(String brand) : super(brand);

  void drive() {
```

```
    // เรียกใช้ Method จาก Class แม่
    super.start();
    print('$brand is driving.');
  }
}

void main() {
  var myCar = Car('Toyota');
  myCar.drive();
}
```

ผลการรัน:

```
Toyota engine is started.
Toyota is driving.
```

---

2. ตัวอย่างโปรแกรมประยุกต์

โปรแกรมเหล่านี้จะแสดงการใช้ Inheritance ในสถานการณ์ที่ซับซ้อนขึ้น

โปรแกรมที่ 4: ระบบพนักงานที่มีพนักงานประจำและชั่วคราว

โครงสร้างไฟล์: main.dart

คำอธิบายโค้ด:

โปรแกรมนี้สร้าง Class Employee เป็น Class แม่ และ Class PermanentEmployee กับ TemporaryEmployee เป็น Class ลูกที่ extends มา โดย PermanentEmployee และ TemporaryEmployee จะมีพฤติกรรมการคำนวณเงินเดือนที่แตกต่างกัน

โค้ดเต็มไฟล์:

Dart

```
class Employee {
  String name;
  double baseSalary;

  Employee(this.name, this.baseSalary);

  void displayInfo() {
    print('พนักงาน: $name');
    print('เงินเดือนพื้นฐาน: \$${baseSalary.toStringAsFixed(2)}');
  }
```

```
}

class PermanentEmployee extends Employee {
  double bonus;

  PermanentEmployee(String name, double baseSalary, this.bonus) : super(name, baseSalary);

  double calculateSalary() {
    return baseSalary + bonus;
  }
}

class TemporaryEmployee extends Employee {
  int hoursWorked;
  double hourlyRate;

  TemporaryEmployee(String name, this.hoursWorked, this.hourlyRate) : super(name, 0);

  double calculateSalary() {
    return hoursWorked * hourlyRate;
  }
}

void main() {
  var permanentEmp = PermanentEmployee('Alice', 50000.0, 10000.0);
  var temporaryEmp = TemporaryEmployee('Bob', 160, 200.0);

  print('--- พนักงานประจำ ---');
  permanentEmp.displayInfo();
  print('เงินเดือนรวม: \$${permanentEmp.calculateSalary().toStringAsFixed(2)}');

  print('\n--- พนักงานชั่วคราว ---');
  temporaryEmp.displayInfo();
  print('เงินเดือนรวม: \$${temporaryEmp.calculateSalary().toStringAsFixed(2)}');
```

```
}
```

ผลการรัน:

```
--- พนักงานประจำ ---
พนักงาน: Alice
เงินเดือนพื้นฐาน: $50000.00
เงินเดือนรวม: $60000.00

--- พนักงานชั่วคราว ---
พนักงาน: Bob
เงินเดือนพื้นฐาน: $0.00
เงินเดือนรวม: $32000.00
```

โปรแกรมที่ 5: ระบบจัดการรูปทรงเรขาคณิต

โครงสร้างไฟล์: main.dart

คำอธิบายโค้ด:

โปรแกรมนี้สร้าง Class Shape เป็น Class แม่ และมี Rectangle กับ Circle เป็น Class ลูกที่สืบทอดมา โดย Rectangle และ Circle จะมีคุณสมบัติและพฤติกรรมเฉพาะของตัวเอง

โค้ดเต็มไฟล์:

Dart

```dart
class Shape {
  String name;

  Shape(this.name);

  void display() {
    print('รูปทรง: $name');
  }
}

class Rectangle extends Shape {
  double width;
  double height;

  Rectangle(this.width, this.height) : super('สี่เหลี่ยมผืนผ้า');
```

```
  double get area => width * height;
}

class Circle extends Shape {
  double radius;

  Circle(this.radius) : super('วงกลม');

  double get area => 3.14 * radius * radius;
}

void main() {
  var myRect = Rectangle(10, 5);
  var myCircle = Circle(7);

  myRect.display();
  print('พื้นที่: ${myRect.area}');

  myCircle.display();
  print('พื้นที่: ${myCircle.area}');
}
```

ผลการรัน:

```
รูปทรง: สี่เหลี่ยมผืนผ้า
พื้นที่: 50.0
รูปทรง: วงกลม
พื้นที่: 153.86
```

Abstract Classes

Abstract Class คือ Class ที่เป็นเพียงแม่แบบ (blueprint) สำหรับ Class อื่นๆ ที่จะมาสืบทอดไปใช้งาน โดยใช้คีย์เวิร์ด **abstract** ซึ่ง Abstract Class ไม่สามารถสร้าง Object ขึ้นมาได้โดยตรง