




Flutter 基礎 8 - Dart Introduction to Dart OOP

參考資料 + 翻轉教室輔助影片



Flutter 基礎與實作 8 - Dart Introduction to Dart OOP
APP 程式設計實務 (2021 Fall)

 <https://youtu.be/uSEadIuoJnc>



1. [What Is Object-Oriented Programming? The Four Basic Concepts of OOP | Indeed.com](#)
2. [Object-oriented Programming in 7 minutes | Mosh - YouTube](#)
3. [Fundamental Concepts of Object Oriented Programming - YouTube](#)
4. [Intro to OOP | Flutter by Example](#)
5. [Dart - Classes And Objects - GeeksforGeeks](#)
6. [Dart Programming - Classes \(tutorialspoint.com\)](#)

<https://www.youtube.com/watch?v=pTB0EiLXUC8>

[Object-oriented Programming in 7 minutes | Mosh - YouTube](#)

<https://www.youtube.com/watch?v=mMQYyJpIjg&t=195s>

[Fundamental Concepts of Object Oriented Programming - YouTube](#)

學習目標

- 基礎物件導向程式設計(OOP)概念。
- Dart 類別(classes)物件(objects)語法。
- Dart 類別設計與物件建構入門應用。

課程設計

4 Basic (Object-Oriented Programming) OOP Principles

What Is Object-Oriented Programming? The Four Basic Concepts of OOP | Indeed.com

- 封裝 (Encapsulation)

The word, "encapsulate," means to enclose something. Just like a pill "encapsulates" or contains the medication inside of its coating, the principle of encapsulation works in a similar way in OOP: by forming a protective barrier around the information contained within a class from the rest of the code. In OOP, we encapsulate by binding the data and functions which operate on that data into a single unit, the class. By doing so, we can hide private details of a class from the outside world and only expose functionality that is important for interfacing with it. When a class does not allow calling code access to its private data directly, we say that it is well encapsulated.

- 抽象 (Abstraction)

Often, it's easier to reason and design a program when you can separate the interface of a class from its implementation, and focus on the interface. This is akin to treating a system as a "black box," where it's not important to understand the gory inner workings in order to reap the benefits of using it. This process is called "abstraction" in OOP, because we are abstracting away the gory implementation details of a class and only presenting a clean and easy-to-use interface via the class'

member functions. Carefully used, **abstraction helps isolate the impact of changes made to the code**, so that if something goes wrong, the change will only affect the implementation details of a class and not the outside code.

- 繼承 (Inheritance)

Object-oriented languages that support classes almost always support the notion of **"inheritance."** Classes can be **organized into hierarchies**, where a class might have one or more parent or child classes. If a class has a parent class, we say it is derived or inherited from the parent class and it represents an "IS-A" type relationship. That is to say, **the child class "IS-A" type of the parent class.** Therefore, **if a class inherits from another class, it automatically obtains a lot of the same functionality and properties from that class** and can be extended to contain separate code and data. A nice feature of inheritance is that it often **leads to good code reuse** since a parent class' functions don't need to be re-defined in any of its child classes.

- 多形 (Polymorphism)

In OOP, **polymorphism** allows for the uniform treatment of classes in a hierarchy. Therefore, calling code only needs to be written to handle objects from the root of the hierarchy, and any object instantiated by any child class in the hierarchy will be handled in the same way. **Because derived objects share the same interface as their parents, the calling code can call any function in that class' interface.** **At run-time, the appropriate function will be called depending on the type of object passed leading to possibly different behaviors.**

Classes/Objects

```
class Cat {  
  String? name;  
  String? color;  
}
```

```

}

void printCat(Cat c) {
  print("${c.name} is a ${c.color} cat.");
}

void main() {
  Cat nora = Cat();
  Cat mick = Cat();

  nora.name = "Nora";
  nora.color = "black";
  mick.name = "Mick";
  mick.color = "white";

  printCat(nora);
  printCat(mick);
}

```

Console

```

Nora is a black cat.
Mick is a white cat.

```

```

1 class Cat {
2   String name;
3   String color;
4 }
5

```

error line 2 • Non-nullable instance field 'name' must be initialized. ([view docs](#))

Try adding an initializer expression, or a generative constructor that initializes it, or mark it 'late'.

error line 3 • Non-nullable instance field 'color' must be initialized. ([view docs](#))

Try adding an initializer expression, or a generative constructor that initializes it, or mark it 'late'.

Why Objects? - Instances of classes

```

class Cat {
  String? name;
  String? color;
}

void main() {
  print(Cat.name);
}

```

```

1 class Cat {
2   String name = "";
3   String color = "";
4 }
5
6 void main() {
7   print(Cat.name);
8 }

```

error line 7 • Instance member 'name' can't be accessed using static access. ([view docs](#))

```

class Cat {
  String? name;
  String? color;
}

void main() {

```

```
Cat nora;

print(nora.name);
}
```

```
1 class Cat {
2   String name = "";
3   String color = "";
4 }
5
6 void main() {
7   Cat nora;
8
9   print(nora.name);
10 }
```

error line 9 • The non-nullable local variable 'nora' must be assigned before it can be used. ([view docs](#))

Try giving it an initializer expression, or ensure that it's assigned on every execution path.

Constructors

```
class Cat {
  String? name;
  String? color;

  Cat(String n, String c) {
    name = n;
    color = c;
  }
}

void printCat(Cat c) {
  print("${c.name} is a ${c.color} cat.");
}

void main() {
  Cat nora = Cat("Nora", "black");
  Cat mick = Cat("Mick", "white");

  printCat(nora);
  printCat(mick);
}
```

Console

```
Nora is a black cat.
Mick is a white cat.
```

Methods (Object Functions)

```
class Cat {
  String? name;
  String? color;

  Cat(String n, String c) {
    name = n;
    color = c;
  }
}
```

Console

```
Nora is a black cat.
Mick is a white cat.
```

```

void info() {
  print("$name is a $color cat.");
}

void main() {
  Cat nora = Cat("Nora", "black");
  Cat mick = Cat("Mick", "white");

  nora.info();
  mick.info();
}

```

Constructors with Ordered / Named Parameters

```

class Cat {
  String? name;
  String? color;

  Cat(this.name, this.color);

  void info() {
    print("$name is a $color cat.");
  }
}

void main() {
  Cat nora = Cat("Nora", "black");
  Cat mick = Cat("Mick", "white");

  nora.info();
  mick.info();
}

```

```

class Cat {
  String? name;
  String? color;

  Cat({this.name, this.color="white"});

  void info() {
    print("$name is a $color cat.");
  }
}

void main() {
  Cat nora = Cat(name:"Nora", color:"black");
  Cat mick = Cat(name:"Mick");

  nora.info();
  mick.info();
}

```

更詳細的操作說明，請參閱『[翻轉教室輔助影片](#)』...

