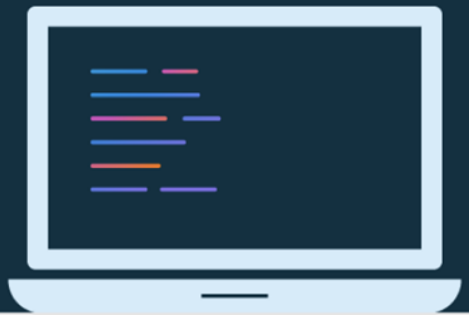
Bài học 3: Lớp và đối tượng



Giới thiệu về bài học này

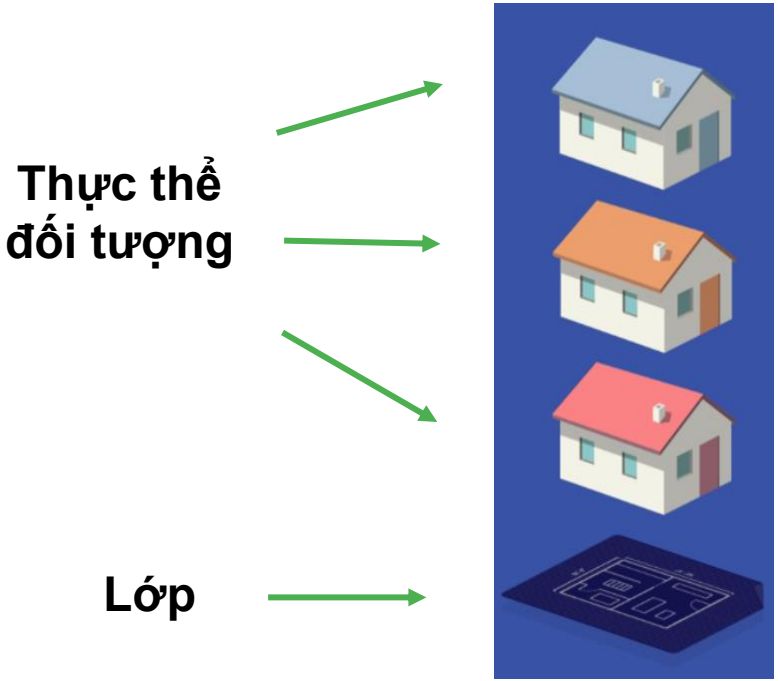
Bài học 3: Lớp và đối tượng

- Lớp
- Tính kế thừa
- Hàm mở rộng
- Lớp đặc biệt
- Sắp xếp mã
- Tóm tắt

Lớp

Lớp

- Lớp là sơ đồ thiết kế của các đối tượng
- Lớp xác định các phương thức hoạt động trên thực thể đối tượng của lớp



Lớp và thực thể đối tượng

Lớp cho ngôi nhà

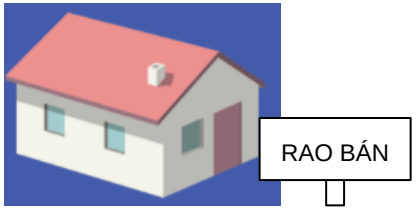
Dữ liệu

- Màu sắc của ngôi nhà (Chuỗi)
- Số lượng cửa sổ (Số nguyên)
- Rao bán (Boolean)

Hành vi

- `updateColor()`
- `putOnSale()`

Thực thể đối tượng



Xác định và dùng lớp

Xác định lớp

```
class House {
  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}
  ...
}
```

Tạo thực thể đối tượng mới

```
val myHouse = House()
println(myHouse)
```

Hàm dựng

Khi một hàm dựng được xác định trong tiêu đề lớp, hàm dựng đó có thể chứa các loại sau:

- Không có tham số

  ```
  class A
  ```

- Tham số
  - Không được đánh dấu bằng `var` hoặc `val` → bản sao chỉ tồn tại trong phạm vi của hàm dựng

    ```
    class B(x: Int)
    ```

  - Được đánh dấu `var` hoặc `val` → bản sao tồn tại trong mọi thực thể của lớp

    ```
    class C(val y: Int)
    ```

Ví dụ về hàm dựng

```
class A
```

```
val aa = A()
```

```
class B(x: Int)
```

```
val bb = B(12)
println(bb.x)
=> compiler error unresolved
reference
```

```
class C(val y: Int)
```

```
val cc = C(42)
println(cc.y)
=> 42
```

Tham số mặc định

Thực thể lớp có thể chứa các giá trị mặc định.

- Dùng các giá trị mặc định để giảm số lượng hàm dựng cần có
- Bạn có thể kết hợp tham số mặc định với tham số bắt buộc
- Ngắn gọn hơn (không cần có nhiều phiên bản hàm dựng)

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
```

Hàm dựng chính

Khai báo hàm dựng chính trong tiêu đề lớp.

```
class Circle(i: Int) {
    init {
        ...
    }
}
```

Về mặt kỹ thuật, hàm dựng này tương đương với:

```
class Circle {
    constructor(i: Int) {
        ...
    }
}
```

Khối khởi tạo

- Mọi mã khởi tạo bắt buộc đều được chạy trong một khối `init` đặc biệt
- Cho phép nhiều khối `init`
- `Khối init` sẽ trở thành phần nội dung của hàm dựng chính

Ví dụ về khối khởi tạo

Dùng từ khóa `init`:

```
class Square(val side: Int) {
    init {
        println(side * 2)
    }
}

val s = Square(10)
=> 20
```

Nhiều hàm dựng

- Dùng từ khóa `constructor` để xác định các hàm dựng phụ
- Hàm dựng phụ phải gọi:
  - Hàm dựng phụ dùng từ khóa `this`
    HOẶC
  - Một hàm dựng phụ khác gọi hàm dựng chính
- Phần nội dung hàm dựng phụ là không bắt buộc

Ví dụ về nhiều hàm dựng

```
class Circle(val radius:Double) {
    constructor(name:String) : this(1.0)
    constructor(diameter:Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }
    init {
        println("Area: ${Math.PI * radius * radius}")
    }
}
val c = Circle(3)
```

Thuộc tính

- Xác định các thuộc tính trong một lớp bằng `val` hoặc `var`
- Truy cập vào các thuộc tính này bằng ký hiệu dấu chấm **.** với tên thuộc tính
- Đặt các thuộc tính này bằng ký hiệu dấu chấm **.** với tên thuộc tính (chỉ khi khai báo bằng `var`)

Lớp Person với thuộc tính name

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name)
    person.name = "Joey"
    println(person.name)
}
```

Truy cập bằng .<property name>

Đặt bằng .<property name>

Phương thức getter và setter tùy chỉnh

Nếu bạn không muốn dùng hành vi `get`/`set` mặc định, hãy:

- Ghi đè `get()` cho một thuộc tính
- Ghi đè `set()` cho một thuộc tính (nếu được xác định là một `var`)

Định dạng:

```
var propertyName: DataType = initialValue
    get() = ...
    set(value) {
        ...
    }
```

Phương thức getter tùy chỉnh

```
class Person(val firstName: String, val lastName:String) {
    val fullName:String
        get() {
            return "$firstName $lastName"
        }
}

val person = Person("John", "Doe")
println(person.fullName)
=> John Doe
```

Phương thức setter tùy chỉnh

```
var fullName:String = ""
    get() = "$firstName $lastName"
    set(value) {
        val components = value.split(" ")
        firstName = components[0]
        lastName = components[1]
        field = value
    }

person.fullName = "Jane Smith"
```

Hàm thành phần

- Lớp cũng có thể chứa các hàm
- Khai báo các hàm như đã nêu trên trang trình bày về *Hàm* trong Bài học 2
  - `Từ khóa fun`
  - Có thể chứa tham số mặc định hoặc tham số bắt buộc
  - Chỉ định loại dữ liệu trả về (nếu không phải là `Unit`)

Tính kế thừa

Tính kế thừa

- Kotlin có tính kế thừa của một lớp mẹ
- Mỗi lớp có đúng một lớp mẹ, gọi là lớp cao cấp
- Mỗi lớp con kế thừa tất cả các thành phần của lớp cao cấp, bao gồm cả những thành phần mà bản thân lớp cao cấp đó đã kế thừa

Nếu không muốn chỉ kế thừa một lớp, bạn có thể xác định một giao diện vì bạn có thể triển khai bao nhiêu giao diện tùy thích.

Giao diện

- Cung cấp một hợp đồng mà tất cả các lớp triển khai đều phải tuân thủ
- Có thể chứa chữ ký phương thức và tên thuộc tính
- Có thể lấy từ các giao diện khác

Định dạng: `interface NameOfInterface { interfaceBody }`

Ví dụ về giao diện

```
interface Shape {
    fun computeArea() : Double
}

class Circle(val radius:Double) : Shape {
    override fun computeArea() = Math.PI * radius * radius
}


val c = Circle(3.0)
println(c.computeArea())
=> 28.274333882308138
```

Mở rộng các lớp

Cách mở rộng một lớp:

- Tạo một lớp mới dùng lớp hiện có làm lõi (lớp con)
- Thêm chức năng vào một lớp mà không cần tạo lớp mới (hàm mở rộng)

Tạo lớp mới

- Theo mặc định, các lớp trong Kotlin không thể phân lớp con
- Dùng từ khóa `open` để cho phép phân lớp con
- Các thuộc tính và hàm được xác định lại bằng từ khóa `override`

Các lớp là cuối cùng theo mặc định

Declare a class

```
class A
```

Try to subclass `A`

```
class B : A
```

```
=>Error: A is final and cannot be inherited from
```

Dùng từ khóa open

Dùng từ khóa `open` để khai báo một lớp sao cho có thể phân lớp con được.

Khai báo một lớp

```
open class C
```

Lớp con của C

```
class D : C()
```

Ghi đè

- Phải dùng từ khóa `open` cho các thuộc tính và phương thức có thể ghi đè (nếu không, bạn sẽ gặp lỗi về trình biên dịch)
- Phải dùng từ khóa `override` khi ghi đè các thuộc tính và phương thức
- Nội dung đã đánh dấu là `override` có thể được ghi đè trong các lớp con (trừ khi được đánh dấu là `final`)

Lớp trừu tượng

- Lớp được đánh dấu là `abstract`
- Không tạo được thực thể mà phải phân lớp con
- Tương tự như giao diện có thêm khả năng lưu trữ trạng thái
- Các thuộc tính và hàm được đánh dấu bằng `abstract` phải được ghi đè
- Có thể bao gồm các thuộc tính và hàm không trừu tượng

Ví dụ về lớp trừu tượng

```
abstract class Food {
    abstract val kcal : Int
    abstract val name : String
    fun consume() = println("I'm eating ${name}")
}
class Pizza() : Food() {
    override val kcal = 600
    override val name = "Pizza"
}
fun main() {
    Pizza().consume()    // "I'm eating Pizza"
}
```

Trường hợp sử dụng

- Bạn cần xác định một loạt hành vi hoặc loại? Hãy cân nhắc dùng giao diện.
- Hành vi có dành riêng cho loại đó không? Hãy cân nhắc dùng lớp.
- Bạn cần kế thừa từ nhiều lớp? Hãy cân nhắc việc tái cấu trúc mã để xem liệu hành vi nào đó có thể được tách biệt thành một giao diện hay không.
- Bạn muốn để các lớp con xác định một số thuộc tính/phương thức là trừu tượng? Hãy cân nhắc dùng lớp trừu tượng.
- Bạn chỉ có thể mở rộng một lớp, nhưng có thể triển khai một hoặc nhiều giao diện.

Hàm mở rộng

Hàm mở rộng

Thêm các hàm vào một lớp hiện có mà bạn không sửa đổi trực tiếp được.

- Hiển thị như thể trình triển khai đã thêm hàm đó
- Không thực sự sửa đổi lớp hiện có
- Không thể truy cập vào các biến thực thể riêng tư

Định dạng: `fun ClassName.functionName( params ) { body }`

Tại sao lại dùng hàm mở rộng?

- Thêm chức năng vào các lớp không phải open
- Thêm chức năng vào các lớp mà bạn không sở hữu
- Tách biệt API lõi với các phương thức trợ giúp cho những lớp mà bạn sở hữu

Xác định các hàm mở rộng ở nơi dễ dàng tìm thấy, chẳng hạn như trong cùng một tệp với lớp hoặc một hàm được đặt tên hợp lý.

Ví dụ về hàm mở rộng

Thêm `isOdd()` vào lớp `Int`:

```
fun Int.isOdd(): Boolean { return this % 2 == 1 }
```

Gọi `isOdd()` trên lớp `Int`:

```
3.isOdd()
```

Các hàm mở rộng rất hữu hiệu trong Kotlin!

Lớp đặc biệt

Lớp dữ liệu

- Lớp đặc biệt tồn tại chỉ để lưu trữ một tập dữ liệu
- Đánh dấu lớp bằng từ khóa `data`
- Tạo phương thức getter cho mỗi thuộc tính (và cả phương thức setter cho var)
- Tạo phương thức `toString()`, `equals()`, `hashCode()`, `copy()` và toán tử phá hủy

Định dạng: `data class <NameOfClass>( parameterList )`

Ví dụ về lớp dữ liệu

Xác định lớp dữ liệu:

```
data class Player(val name: String, val score: Int)
```

Dùng lớp dữ liệu:

```
val firstPlayer = Player("Lauren", 10)
println(firstPlayer)
=> Player(name=Lauren, score=10)
```

Các lớp dữ liệu giúp mã ngắn gọn hơn nhiều!

Pair và Triple

- `Pair` và `Triple` là các lớp dữ liệu được xác định trước giúp lưu trữ 2 hoặc 3 đoạn dữ liệu tương ứng
- Truy cập vào các biến bằng `.first`, `.second`, `.third` tương ứng
- Các lớp `dữ liệu` được đặt tên thường là lựa chọn tốt hơn (tên có ý nghĩa hơn cho trường hợp sử dụng của bạn)

Ví dụ về Pair và Triple

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")

println(bookAuthor)
=> (Harry Potter, J.K. Rowling)

val bookAuthorYear = Triple("Harry Potter", "J.K. Rowling", 1997)
println(bookAuthorYear)

println(bookAuthorYear.third)
=> (Harry Potter, J.K. Rowling, 1997)
    1997
```

Biến thể to của Pair

`Biến thể to` đặc biệt của Pair cho phép bạn loại bỏ dấu ngoặc đơn và dấu chấm (hàm infix).

Biến thể này giúp mã dễ đọc hơn

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K. Rowling)
```

Biến thể này cũng được dùng trong các tập hợp như Map và HashMap

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
=> map of Int to String {1=x, 2=y, 3=zz}
```

Lớp enum

Là loại dữ liệu do người dùng xác định cho một tập hợp các giá trị được đặt tên

- Dùng `this` để yêu cầu các thực thể là một trong nhiều giá trị không đổi
- Theo mặc định, bạn sẽ không nhìn thấy giá trị không đổi
- Dùng `enum` phía trước từ khóa `class`

Định dạng: `enum class EnumName { NAME1, NAME2, … NAMEn }`
Referenced via `EnumName.<ConstantName>`

Ví dụ về lớp enum

Xác định lớp `enum` bằng màu đỏ, xanh lục và xanh dương.

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255)
}


println("" + Color.RED.r + " " + Color.RED.g + " " + Color.RED.b)
=> 255 0 0
```

Đối tượng/singleton

- Đôi khi, bạn chỉ muốn một thực thể của lớp tồn tại
- Dùng từ khóa `object` thay cho từ khóa `class`
- Được truy cập bằng `NameOfObject.<function or variable>`

Ví dụ về đối tượng/singleton

```
object Calculator {
    fun add(n1: Int, n2: Int): Int {
        return n1 + n2
    }
}

println(Calculator.add(2,4))
=> 6
```

Đối tượng companion

- Cho phép mọi thực thể của một lớp dùng chung một thực thể của tập hợp các biến hoặc hàm
- Dùng từ khóa `companion`
- Được tham chiếu qua `ClassName.PropertyOrFunction`

Ví dụ về đối tượng companion

```
class PhysicsSystem {
    companion object WorldConstants {
        val gravity = 9.8
        val unit = "metric"
        fun computeForce(mass: Double, accel: Double): Double {
            return mass * accel
        }
    }
}
println(PhysicsSystem.WorldConstants.gravity)
println(PhysicsSystem.WorldConstants.computeForce(10.0, 10.0))
=> 9.8100.0
```

Sắp xếp mã

Một tệp, nhiều thực thể

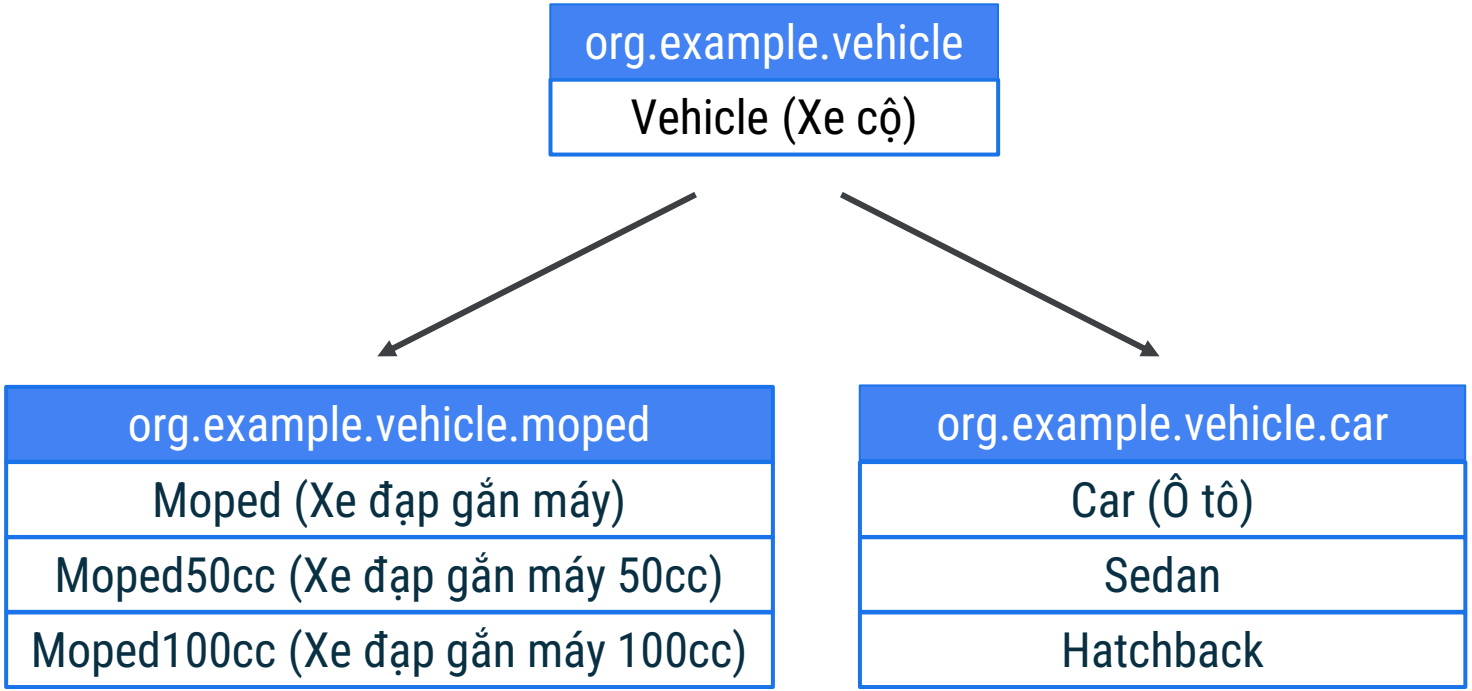
- Kotlin KHÔNG thực thi một thực thể (lớp/giao diện) cho mỗi quy ước tệp
- Bạn có thể và nên nhóm các cấu trúc liên quan trong cùng một tệp
- Lưu ý đến độ dài và sự lộn xộn của tệp

Gói

- Cung cấp phương thức để sắp xếp
- Nhìn chung, mã nhận dạng là các từ viết thường được phân tách bằng dấu chấm
- Được khai báo trong dòng mã đầu tiên không phải nhận xét trong một tệp theo sau là từ khóa `package`

```
package org.example.game
```

Ví dụ về hệ phân cấp lớp



Từ khóa xác định mức độ hiển thị

Dùng từ khóa xác định mức độ hiển thị để giới hạn thông tin mà bạn hiển thị.

- `public` nghĩa là có thể nhìn thấy bên ngoài lớp. Mọi thứ đều ở chế độ công khai theo mặc định, bao gồm cả các biến và phương thức của lớp.
- `private` nghĩa là chỉ hiển thị trong lớp đó (hoặc tệp nguồn nếu bạn đang làm việc với các hàm).
- `protected` giống như `private`, nhưng cũng sẽ hiển thị với bất kỳ lớp con nào.

Tóm tắt

Tóm tắt

Trong Bài học 3, bạn đã tìm hiểu về:

- Lớp, hàm dựng, phương thức getter và setter
- Tính kế thừa, giao diện và cách mở rộng các lớp
- Hàm mở rộng
- Các lớp đặc biệt: lớp dữ liệu, lớp enum, đối tượng/singleton, đối tượng companion
- Gói
- Từ khóa xác định mức độ hiển thị

Lộ trình

Thực hành những gì bạn đã học được bằng cách hoàn thành lộ trình này:

Bài học 3: Lớp và đối tượng