

Họ tên: Lê Phúc Hưng

MSSV: 20215276

Mã lớp: 151902

Môn học: Phát triển ứng dụng cho thiết bị di động (IT4785)

Source Code: <https://github.com/lephuchung/HustMobile>

Bài: Lesson 3 - Classes and objects

Contents

1. Using Classes and Objects in Kotlin	3
1.1. Terminology.....	3
1.2. Create a class	3
1.2.1. Create a package.....	3
1.2.2. Create a class with properties	4
1.2.3. Create a main() function.....	5
1.2.4. Add a method.....	6
1.3. Add class constructors	7
1.3.1. Create a constructor	7
1.3.2. Add init blocks.....	9
1.3.3. Learn about secondary constructors	10
1.3.4. Add a new property getter	12
1.3.5. Add a property setter	13
1.4. Learn about visibility modifiers	14
1.5. Learn about subclasses and inheritance.....	15
1.5.1. Make the Aquarium class open.....	15
1.5.2. Create a subclass.....	17
1.6. Compare abstract classes and interfaces.....	18
1.6.1. Create an abstract class.....	19
1.6.2. Create an interface	21
1.6.3. When to use abstract classes versus interfaces.....	23
1.7. Use interface delegation	24

1.7.1. Make a new interface.....	24
1.7.2. Make a singleton class.....	25
1.7.3. Add interface delegation for FishColor	26
1.7.4. Add interface delegation for FishAction	27
1.8. Create a data class.....	29
1.8.1. Create a data class	29
1.8.2. Use destructuring.....	30
1.9. Learn about singletons and enums	31
1.9.1. Recall singleton classes	31
1.9.2. Create an enum	31
2. Pairs/triples, collections, constants, and writing extension functions	33
2.1. Create a Companion Object.....	33
2.2. Learn about pairs and triples	33
2.3. Learn more about collections	36
2.3.1. Understand more about lists	36
2.3.2. Try out hash maps.....	38
2.4. Organize and define constants	39
2.4.1. Learn about const vs. val	39
2.4.2. Create a companion object	40
2.5. Understand extension functions	40
2.5.1. Write an extension function.....	41
2.5.2. Learn the limitations of extensions.....	41
2.5.3. Add an extension property	43
3. Create Android app Hello World.....	46
3.1. Steps to implement the program.....	46
3.2. The program Hello World.....	46

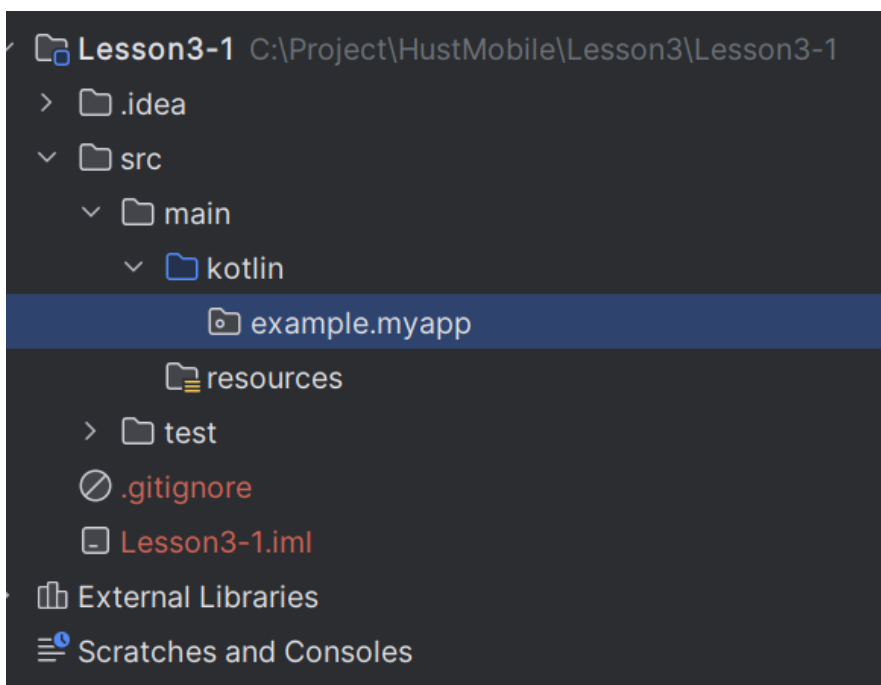
1. Using Classes and Objects in Kotlin

1.1. Terminology

- Các lớp là bản thiết kế cho các đối tượng. Ví dụ, một lớp Aquarium là bản thiết kế để tạo ra một đối tượng Aquarium.
- Các đối tượng là các thể hiện của các lớp; một đối tượng Aquarium là một Aquarium thực sự tồn tại trong bộ nhớ.
- Các thuộc tính là các đặc điểm của các lớp, chẳng hạn như chiều dài, chiều rộng và chiều cao của một Aquarium.
- Các phương thức, còn được gọi là các hàm thành viên, là chức năng của lớp. Các phương thức là những gì bạn có thể "làm" với đối tượng. Ví dụ, bạn có thể fillWithWater() một đối tượng Aquarium.
- Giao diện là một thông số kỹ thuật mà một lớp có thể triển khai. Ví dụ, việc vệ sinh là phổ biến đối với các đối tượng khác ngoài aquarium và việc vệ sinh thường diễn ra theo những cách tương tự đối với các đối tượng khác nhau. Vì vậy, bạn có thể có một giao diện có tên là Clean định nghĩa phương thức clean(). Lớp Aquarium có thể triển khai giao diện Clean để vệ sinh bể cá bằng miếng bọt biển mềm.
- Các gói là một cách để nhóm các mã liên quan để giữ cho chúng được sắp xếp hợp lý hoặc để tạo một thư viện mã. Sau khi tạo một gói, bạn có thể sử dụng import để cho phép bạn tham chiếu trực tiếp các lớp trong gói đó.

1.2. Create a class

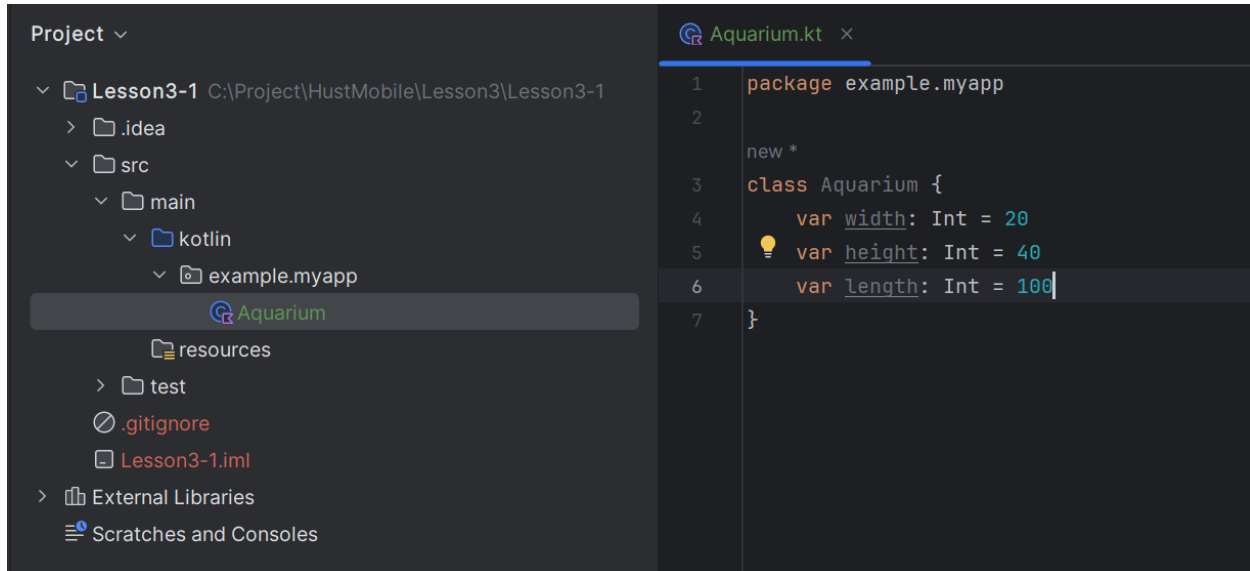
1.2.1. Create a package



Các bước thực hiện:

1. Chuột phải vào thư mục **src > main > kotlin**
2. Chọn **New > Package**, đặt tên là **example.myapp**

1.2.2. Create a class with properties

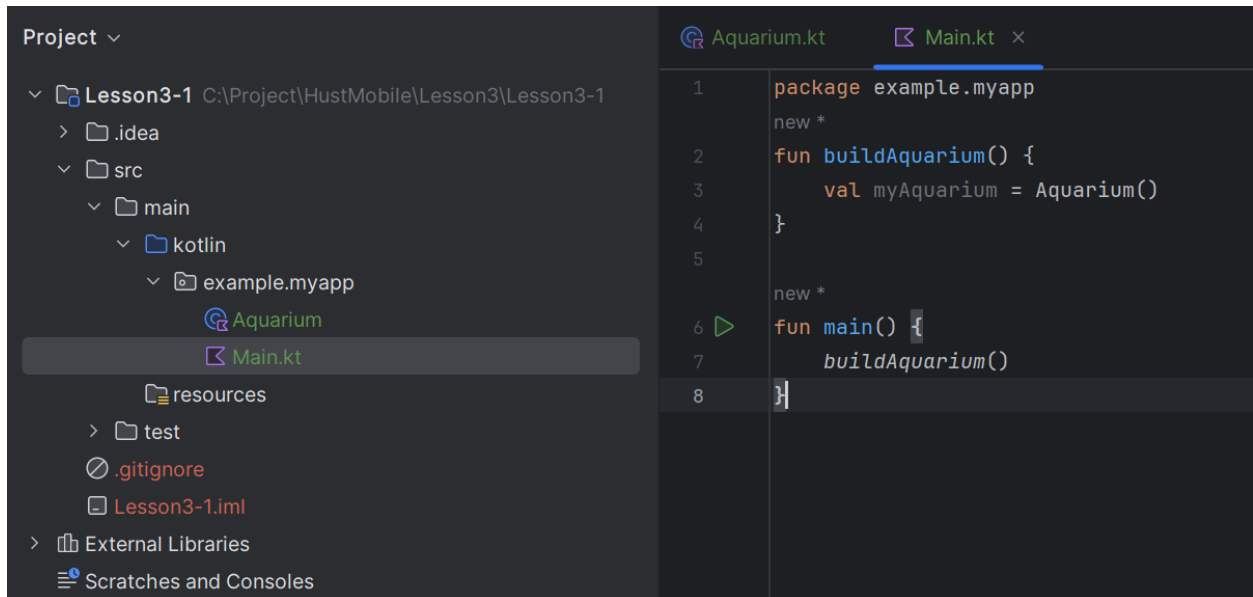


Các bước thực hiện:

Các lớp được định nghĩa bằng từ khóa **class** và theo quy ước, tên lớp bắt đầu bằng chữ in hoa.

1. Nhấp chuột phải vào gói **example.myapp**.
2. Chọn **New > Kotlin File/Class**.
3. Trong **Kind**, chọn **Class** và đặt tên cho lớp là **Aquarium**. IntelliJ IDEA bao gồm tên gói trong tệp và tạo một lớp **Aquarium** trống cho bạn.
4. Bên trong lớp **Aquarium**, hãy định nghĩa và khởi tạo các thuộc tính **var** cho chiều rộng, chiều cao và chiều dài (tính bằng cm). Khởi tạo các thuộc tính bằng các giá trị mặc định.

1.2.3. Create a main() function



Các bước thực hiện:

1. Tạo một tệp mới có tên Main.kt để chứa hàm main().
2. Trong ngăn Project ở bên trái, nhấp chuột phải vào gói example.myapp.
3. Chọn New > Kotlin File / Class.
4. Trong danh sách thả xuống Kind, giữ nguyên lựa chọn là File và đặt tên cho tệp là Main.kt. IntelliJ IDEA bao gồm tên gói nhưng không bao gồm định nghĩa lớp cho tệp.
5. Xác định hàm buildAquarium() và bên trong tạo một thể hiện của Aquarium. Để tạo một thể hiện, hãy tham chiếu lớp như thể đó là một hàm, Aquarium(). Thao tác này sẽ gọi hàm tạo của lớp và tạo một thể hiện của lớp Aquarium, tương tự như khi sử dụng từ khóa new trong các ngôn ngữ khác.
6. Xác định hàm main() và gọi buildAquarium().

1.2.4. Add a method

The image shows a Kotlin IDE with two panels. The top panel displays the project structure on the left and the code for `Aquarium.kt` on the right. The bottom panel shows the same project structure and code for `Main.kt`.

Top Panel: Aquarium.kt

```
1 package example.myapplication
2
3 new *
4 class Aquarium {
5     var width: Int = 20
6     var height: Int = 40
7     var length: Int = 100
8
9     new *
10    fun printSize() {
11        println("Width: $width cm " +
12                "Length: $length cm " +
13                "Height: $height cm ")
14    }
15 }
```

Bottom Panel: Main.kt

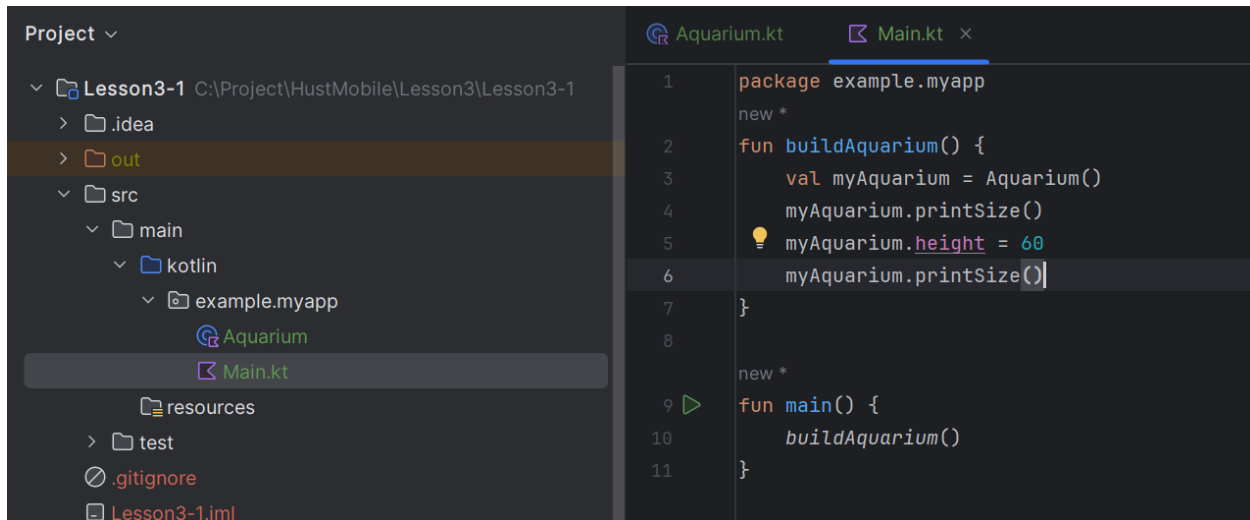
```
1 package example.myapplication
2 new *
3 fun buildAquarium() {
4     val myAquarium = Aquarium()
5     myAquarium.printSize()
6 }
7
8 new *
9 fun main() {
10    buildAquarium()
11 }
```

Output:

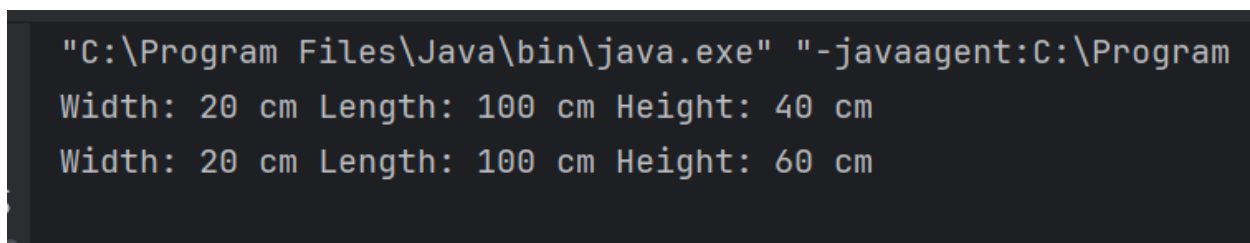
```
"C:\Program Files\Java\bin\java.exe" "-jav
Width: 20 cm Length: 100 cm Height: 40 cm
Process finished with exit code 0
```

Các bước thực hiện:

1. Trong lớp Aquarium, thêm phương thức printSize() để in các thuộc tính kích thước của bể cá.
2. Trong Main.kt, trong buildAquarium(), gọi phương thức printSize() trên myAquarium.
3. Chạy chương trình được kết quả.



Chạy đoạn code mới ta được:



1.3. Add class constructors

1.3.1. Create a constructor

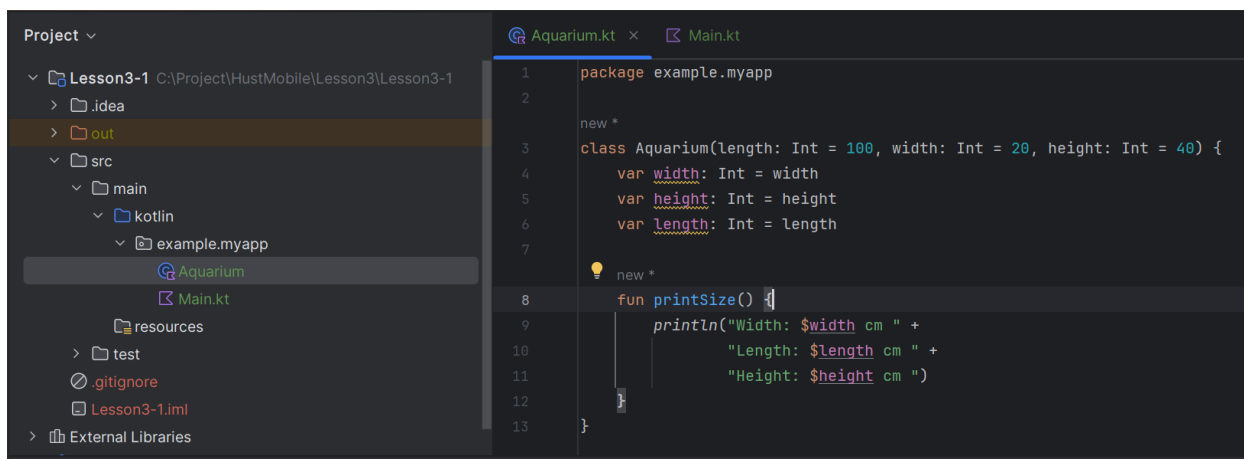
Mô tả:

Trong bước này, bạn thêm một hàm tạo vào lớp Aquarium mà bạn đã tạo trong tác vụ đầu tiên. Trong ví dụ trước đó, mọi phiên bản Aquarium đều được tạo với cùng kích thước. Bạn có thể thay đổi kích thước sau khi tạo bằng cách thiết lập các thuộc tính, nhưng sẽ đơn giản hơn nếu tạo đúng kích thước ngay từ đầu.

Trong một số ngôn ngữ lập trình như Java, hàm tạo được định nghĩa bằng cách tạo một phương thức trong lớp có cùng tên với lớp. Trong Kotlin, bạn định nghĩa hàm tạo trực tiếp trong chính khai báo lớp, chỉ định các tham số bên trong dấu ngoặc đơn như thể lớp là một phương thức. Giống như các hàm trong Kotlin, các tham số đó có thể bao gồm các giá trị mặc định.

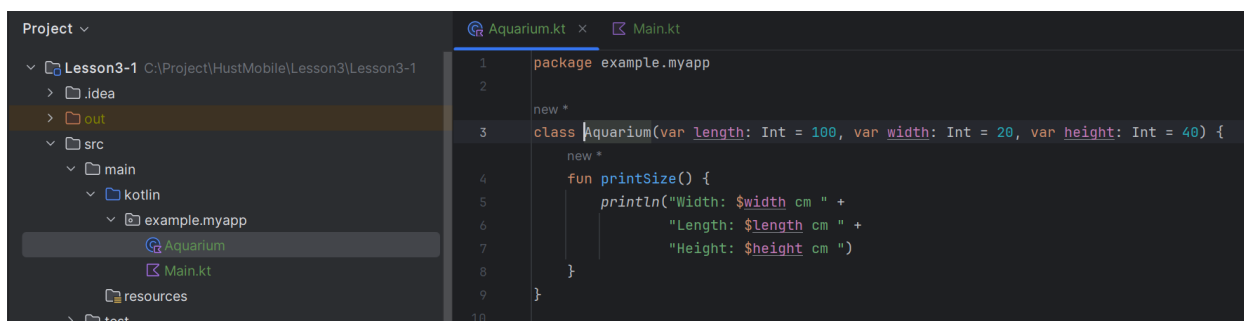
Các bước thực hiện:

1. Trong lớp Aquarium đã tạo trước đó, thay đổi định nghĩa lớp để bao gồm ba tham số hàm tạo với các giá trị mặc định cho chiều dài, chiều rộng và chiều cao, rồi gán chúng cho các thuộc tính tương ứng.
2. Cách Kotlin gọn nhẹ hơn là định nghĩa các thuộc tính trực tiếp bằng hàm tạo, sử dụng var hoặc val, và Kotlin cũng tự động tạo các getter và setter. Sau đó, ta có thể xóa các định nghĩa thuộc tính trong phần thân của lớp.
3. Khi tạo đối tượng Aquarium bằng hàm tạo đó, có thể không chỉ định đối số và lấy các giá trị mặc định, hoặc chỉ định một số trong số chúng, hoặc chỉ định tất cả chúng và tạo Aquarium có kích thước tùy chỉnh hoàn toàn. Trong hàm buildAquarium(), hãy thử các cách khác nhau để tạo đối tượng Aquarium bằng cách sử dụng các tham số được đặt tên.
4. Chạy chương trình



```
1 package example.myapp
2
3 new *
4 class Aquarium(length: Int = 100, width: Int = 20, height: Int = 40) {
5     var width: Int = width
6     var height: Int = height
7     var length: Int = length
8
9     new *
10    fun printSize() {
11        println("Width: $width cm " +
12                "Length: $length cm " +
13                "Height: $height cm ")
14    }
15 }
```

Ta có thể không cần khai báo thuộc tính trong nội dung của lớp bằng các như sau:



```
1 package example.myapp
2
3 new *
4 class Aquarium(var length: Int = 100, var width: Int = 20, var height: Int = 40) {
5     new *
6     fun printSize() {
7         println("Width: $width cm " +
8                 "Length: $length cm " +
9                 "Height: $height cm ")
10    }
11 }
```

Thêm vào hàm buildAquarium trong Main.kt:

The screenshot shows an IDE with a project named 'Lesson3-1'. The left sidebar shows the project structure: 'Lesson3-1' > 'src' > 'main' > 'kotlin' > 'example.myapp'. The main editor shows the 'Main.kt' file with the following Kotlin code:

```
7 //}
8 new *
9 fun buildAquarium() {
10     val aquarium1 = Aquarium()
11     aquarium1.printSize()
12     // default height and length
13     val aquarium2 = Aquarium(width = 25)
14     aquarium2.printSize()
15     // default width
16     val aquarium3 = Aquarium(height = 35, length = 110)
17     aquarium3.printSize()
18     // everything custom
19     val aquarium4 = Aquarium(width = 25, height = 35, length = 110)
20     aquarium4.printSize()
21 }
```

The bottom panel shows the output of the program:

```
"C:\Program Files\Java\bin\java.exe" "-javaagen
Width: 20 cm Length: 100 cm Height: 40 cm
Width: 25 cm Length: 100 cm Height: 40 cm
Width: 20 cm Length: 110 cm Height: 35 cm
Width: 25 cm Length: 110 cm Height: 35 cm
```

The bottom status bar shows the file path: 'Lesson3-1 > src > main > kotlin > example > myapp > Main.kt'.

1.3.2. Add init blocks

Mô tả:

Các constructor ví dụ ở trên chỉ khai báo các thuộc tính và gán giá trị của một biểu thức cho chúng. Nếu constructor của bạn cần nhiều mã khởi tạo hơn, nó có thể được đặt trong một hoặc nhiều khối init. Trong bước này, bạn thêm một số khối init vào lớp Aquarium.

Các bước thực hiện:

1. Trong lớp Aquarium, thêm một khối init để in ra rằng đối tượng đang khởi tạo và một khối init thứ hai để in ra thể tích tính bằng lít. Lưu ý rằng các khối init có thể có nhiều câu lệnh.
2. Chạy chương trình.

The screenshot displays an IDE with a project named 'Lesson3-1'. The file explorer on the left shows the project structure, including 'src/main/kotlin/example.myapp' with files 'Aquarium.kt' and 'Main.kt'. The main editor shows the 'Aquarium.kt' file with the following code:

```
1 package example.myapp
2
3 new *
4 class Aquarium(var length: Int = 100, var width: Int = 20, var height: Int = 40) {
5     new *
6     init {
7         println("aquarium initializing")
8     }
9     new *
10    init {
11        // 1 liter = 1000 cm^3
12        println("Volume: ${width * length * height / 1000} liters")
13    }
14    new *
15    fun printSize() {
16        println("Width: $width cm " +
17                "Length: $length cm " +
18                "Height: $height cm")
19    }
20 }
```

Below the editor, the 'Run' tab is active, showing the execution of 'MainKt'. The output is as follows:

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program File
aquarium initializing
Volume: 80 liters
Width: 20 cm Length: 100 cm Height: 40 cm
aquarium initializing
Volume: 100 liters
Width: 25 cm Length: 100 cm Height: 40 cm
aquarium initializing
Volume: 77 liters
Width: 20 cm Length: 110 cm Height: 35 cm
aquarium initializing
Volume: 96 liters
Width: 25 cm Length: 110 cm Height: 35 cm
Process finished with exit code 0
```

1.3.3. Learn about secondary constructors

Mô tả:

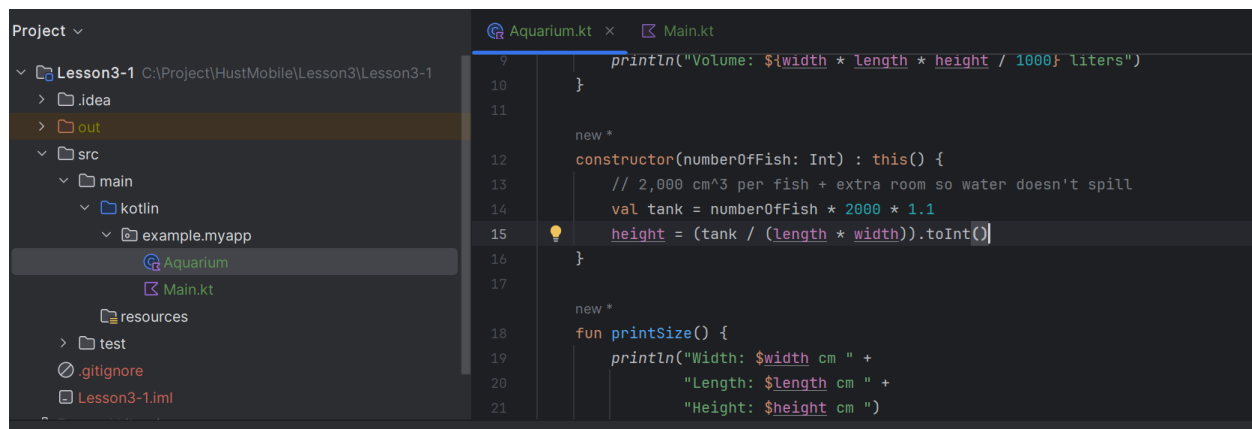
Trong bước này, bạn sẽ tìm hiểu về các hàm tạo thứ cấp và thêm một hàm tạo vào lớp của mình. Ngoài một hàm tạo chính, có thể có một hoặc nhiều khối `init`, một lớp Kotlin cũng có thể có một hoặc nhiều hàm tạo thứ cấp. Tính năng này cho phép quá tải hàm tạo, tức là các hàm tạo có các đối số khác nhau.

Phong cách mã hóa Kotlin gợi ý rằng mỗi lớp chỉ nên có một hàm tạo, sử dụng các giá trị mặc định và các tham số được đặt tên. Điều này là do việc sử dụng nhiều hàm tạo, đặc biệt là các hàm tạo quá tải, dẫn đến nhiều đường dẫn mã hơn và khả năng một hoặc nhiều đường dẫn sẽ không được kiểm tra. Nếu bạn cần nhiều hàm tạo, hãy cân nhắc xem hàm tạo gốc có hoạt động để giữ cho định nghĩa lớp sạch không.

Mỗi hàm tạo thứ cấp phải gọi hàm tạo chính trước, trực tiếp bằng cách sử dụng `this()` hoặc gián tiếp bằng cách gọi một hàm tạo thứ cấp khác. Điều này có nghĩa là bất kỳ khối `init` nào trong hàm tạo chính sẽ được thực thi cho tất cả các hàm tạo và tất cả mã trong hàm tạo chính sẽ được thực thi trước.

Các bước thực hiện:

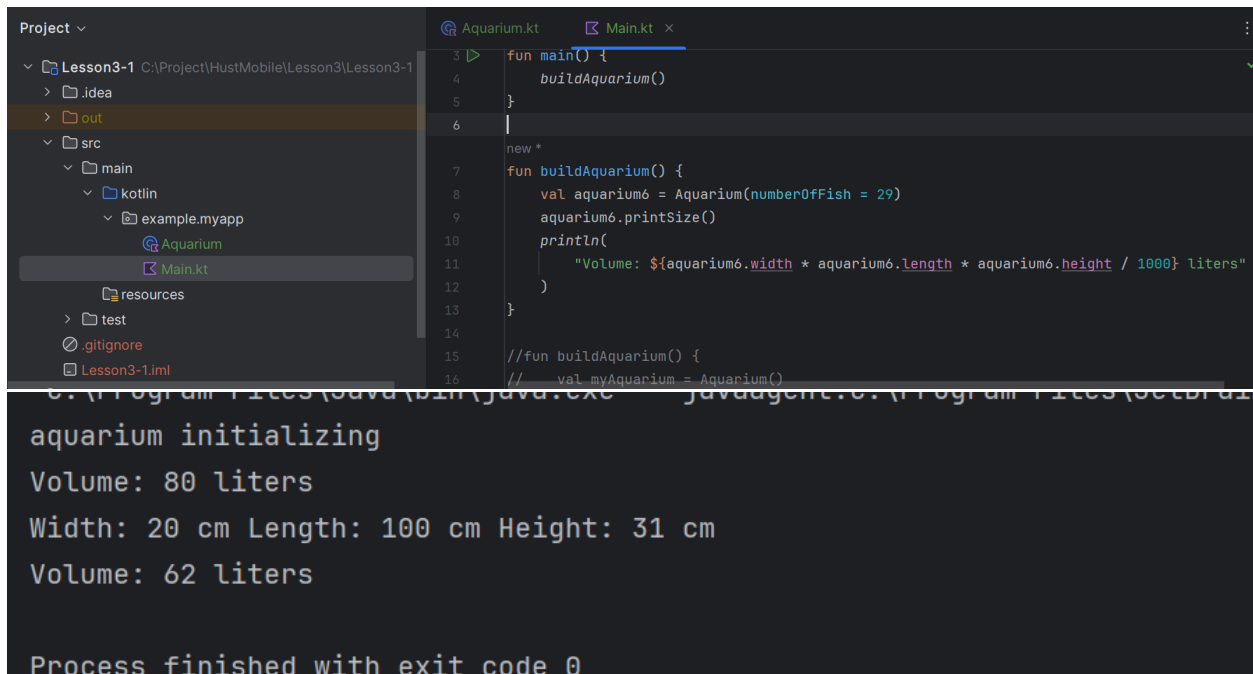
1. Trong lớp `Aquarium`, thêm một constructor thứ cấp lấy một số lượng cá làm đối số của nó, sử dụng từ khóa constructor. Tạo một thuộc tính `val tank` cho thể tích tính toán của bể cá tính bằng lít dựa trên số lượng cá. Giả sử 2 lít (2.000 cm^3) nước cho mỗi con cá, cộng thêm một chút không gian để nước không tràn ra ngoài.
2. Bên trong constructor thứ cấp, giữ nguyên chiều dài và chiều rộng (được đặt trong constructor chính) và tính chiều cao cần thiết để làm cho bể có thể tích đã cho.
3. Trong hàm `buildAquarium()`, thêm lệnh gọi để tạo `Aquarium` bằng constructor thứ cấp mới. In kích thước và thể tích.
4. Chạy chương trình.



```
Project
└─ Lesson3-1
   └─ src
      └─ main
         └─ kotlin
            └─ example.myapp
               └─ Aquarium
                  └─ Main.kt

resources
└─ test
└─ .gitignore
└─ Lesson3-1.iml

9      println("Volume: ${width * length * height / 1000} liters")
10     }
11
12     new *
13     constructor(numberOfFish: Int) : this() {
14         // 2,000 cm^3 per fish + extra room so water doesn't spill
15         val tank = numberOfFish * 2000 * 1.1
16         height = (tank / (length * width)).toInt()
17     }
18
19     new *
20     fun printSize() {
21         println("Width: $width cm " +
22             "Length: $length cm " +
23             "Height: $height cm ")
24     }
```



```
Project
├── Lesson3-1
│   ├── .idea
│   ├── out
│   ├── src
│   │   ├── main
│   │   │   ├── kotlin
│   │   │   │   ├── example.myapp
│   │   │   │   ├── Aquarium.kt
│   │   │   │   └── Main.kt
│   │   ├── resources
│   │   └── test
│   ├── .gitignore
│   └── Lesson3-1.iml
└── ...

3 fun main() {
4     buildAquarium()
5 }
6
7 new *
8 fun buildAquarium() {
9     val aquarium6 = Aquarium(numberOfFish = 29)
10    aquarium6.printSize()
11    println(
12        "Volume: ${aquarium6.width * aquarium6.length * aquarium6.height / 1000} liters"
13    )
14 }
15 //fun buildAquarium() {
16 //    val myAquarium = Aquarium()
17 //}
```

```
aquarium initializing
Volume: 80 liters
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters

Process finished with exit code 0
```

1.3.4. Add a new property getter

Mô tả:

Trong bước này, bạn thêm một phương thức lấy thuộc tính rõ ràng. Kotlin tự động định nghĩa các phương thức lấy và thiết lập khi bạn định nghĩa các thuộc tính, nhưng đôi khi giá trị của một thuộc tính cần được điều chỉnh hoặc tính toán. Ví dụ ở trên, bạn đã in thể tích của Aquarium. Bạn có thể làm cho thể tích khả dụng dưới dạng một thuộc tính bằng cách định nghĩa một biến và một phương thức lấy cho nó. Vì thể tích cần được tính toán, nên phương thức lấy cần trả về giá trị đã tính toán, bạn có thể thực hiện điều này bằng một hàm một dòng theo ngay sau tên và loại thuộc tính.

Các bước thực hiện:

1. Trong lớp Aquarium, hãy định nghĩa một thuộc tính Int có tên là volume và định nghĩa một phương thức get() để tính toán volume ở dòng tiếp theo.
2. Xóa khối init in volume.
3. Xóa mã trong buildAquarium() in volume.
4. Trong phương thức printSize(), hãy thêm một dòng để in volume.
5. Chạy chương trình

```
Project
└─ Lesson3-1 C:\Project\HustMobile\Lesson3\Lesson3-1
   └─ .idea
      └─ out
         └─ src
            └─ main
               └─ kotlin
                  └─ example.myapp
                     └─ Aquarium
                        └─ Main.kt
                           └─ resources
                              └─ test
                                 └─ .gitignore
                                    └─ Lesson3-1.iml
```

```
new *
24 val volume: Int
25     get() = width * height * length / 1000 // 1000 cm^3 = 1 liter
26
new *
27 fun printSize() {
28     println("Width: $width cm " +
29         "Length: $length cm " +
30         "Height: $height cm ")
31 }
32 // 1 liter = 1000 cm^3
33 println("Volume: $volume liters")
34
35 }
```

```
7 fun buildAquarium() {
8     val aquarium6 = Aquarium(numberOfFish = 29)
9     aquarium6.printSize()
10 //     println(
11 //         "Volume: ${aquarium6.width * aquarium6.length * aquarium6.height} cm^3"
12 //     )
13 }
```

```
aquarium initializing
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters

Process finished with exit code 0
```

1.3.5. Add a property setter

Mô tả: Trong bước này, ta tạo một trình set thuộc tính mới cho khối lượng

Các bước thực hiện:

1. Trong lớp Aquarium, hãy đổi volume thành var để có thể thiết lập nhiều lần.
2. Thêm một setter cho thuộc tính volume bằng cách thêm phương thức set() bên dưới getter, phương thức này sẽ tính toán lại chiều cao dựa trên lượng nước được cung cấp. Theo quy ước, tên của tham số setter là value, nhưng bạn có thể thay đổi nếu muốn.
3. Trong buildAquarium(), hãy thêm mã để thiết lập thể tích của Aquarium thành 70 lít. In kích thước mới.
4. Chạy lại chương trình và quan sát chiều cao và thể tích đã thay đổi.

```
Project
└─ Lesson3-1 C:\Project\HustMobile\Lesson3\Lesson3-1
   └─ .idea
      └─ out
         └─ src
            └─ main
               └─ kotlin
                  └─ example.myapp
                     └─ Aquarium
                        └─ Main.kt
                           └─ resources
                              └─ test
                                 └─ .gitignore
                                    └─ Lesson3-1.iml
```

```
Aquarium.kt
19 //      println("Width: $width cm " +
20 //            "Length: $length cm " +
21 //            "Height: $height cm ")
22 //    }
23
24 new *
25 var volume: Int
26     get() = width * height * length / 1000 // 1000 cm^3 = 1 liter
27     set(value) {
28         height = (value * 1000) / (width * length)
29     }
30
31 new *
32 fun printSize() {
33     println("Width: $width cm " +
34           "Length: $length cm " +
35           "Height: $height cm ")
36 }
```

```
Main.kt
1 new *
2 fun buildAquarium() {
3     val aquarium6 = Aquarium(numberOfFish = 29)
4     aquarium6.printSize()
5     aquarium6.volume = 70
6     aquarium6.printSize()
7     //      println(
8     //          "Volume: ${aquarium6.width * aquarium6.length * aquarium6.height}
9     //      )
10 }
11
12 new *
```

```
Terminal
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
aquarium initializing
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters
Width: 20 cm Length: 100 cm Height: 35 cm
Volume: 70 liters
```

1.4. Learn about visibility modifiers

Cho đến nay, chưa có trình sửa đổi khả năng hiển thị nào, chẳng hạn như public hoặc private, trong mã. Đó là vì theo mặc định, mọi thứ trong Kotlin đều là public, nghĩa là mọi thứ đều có thể được truy cập ở mọi nơi, bao gồm các lớp, phương thức, thuộc tính và biến thành viên.

Trong Kotlin, các lớp, đối tượng, giao diện, hàm tạo, hàm, thuộc tính và trình thiết lập của chúng có thể có trình sửa đổi khả năng hiển thị:

- `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.
- `internal` nghĩa là chỉ hiển thị trong mô-đun đó. Mô-đun là một tập hợp các tệp Kotlin được biên dịch cùng nhau, ví dụ: thư viện, máy khách hoặc ứng dụng, ứng dụng máy chủ trong dự án IntelliJ. Lưu ý rằng cách sử dụng "module" ở đây không liên quan đến các mô-đun Java được giới thiệu trong Java 9.
- `public` nghĩa là hiển thị bên ngoài lớp. Theo mặc định, mọi thứ đều là `public`, bao gồm các biến và phương thức của lớp.

```
new *
var volume: Int
    get() = width * height * length / 1000 // 1000 cm^3 = 1 liter
    private set(value) {
        height = (value * 1000) / (width * length)
    }

new *
7 fun buildAquarium() {
8     val aquarium6 = Aquarium(numberOfFish = 29)
9     aquarium6.printSize()
10    aquarium6.volume = 70
11    aquarium6.printSize()
12    // println(
```

1.5. Learn about subclasses and inheritance

Trong nhiệm vụ này, bạn sẽ tìm hiểu cách các lớp con và kế thừa hoạt động trong Kotlin. Chúng tương tự như những gì bạn đã thấy trong các ngôn ngữ khác, nhưng có một số điểm khác biệt.

Trong Kotlin, theo mặc định, các lớp không thể được phân lớp con. Bạn phải đánh dấu một lớp là mở để cho phép phân lớp con. Trong các lớp con đó, bạn cũng phải đánh dấu các thuộc tính và biến thành viên là mở để ghi đè chúng trong lớp con. Từ khóa mở là bắt buộc, để ngăn chặn việc vô tình rò rỉ thông tin chi tiết về triển khai như một phần của định nghĩa lớp.

1.5.1. Make the Aquarium class open

Các bước thực hiện:

1. Đánh dấu lớp Aquarium và tất cả các thuộc tính của nó bằng từ khóa open.
2. Thêm một thuộc tính hình dạng mở với giá trị "rectangle".
3. Thêm một thuộc tính nước mở với một phương thức lấy trả về 90% thể tích của Aquarium.
4. Thêm mã vào phương thức printSize() để in hình dạng và lượng nước theo phần trăm thể tích.
5. Trong buildAquarium(), hãy thay đổi mã để tạo Aquarium với chiều rộng = 25, chiều dài = 25 và chiều cao = 40.
6. Chạy chương trình.

```

1 package example.myapp
2 open class Aquarium(var length: Int = 100, var width: Int = 20, var height: Int = 40) {
3     new *
4     init {
5         println("aquarium initializing")
6     }
7     new *
8     constructor(numberOfFish: Int) : this() {
9         // 2,000 cm^3 per fish + extra room so water doesn't spill
10        val tank = numberOfFish * 2000 * 1.1
11        height = (tank / (length * width)).toInt()
12    }
13    new *
14    open var volume: Int
15    get() = width * height * length / 1000
16    set(value) {
17        height = (value * 1000) / (width * length)
18    }
19    open val shape = "rectangle"
20    new *
21    open var water: Double = 0.0
22    get() = volume * 0.9
23    new *
24    fun printSize() {
25        println(shape)
26        println("Width: $width cm " +
27            "Length: $length cm " +
28            "Height: $height cm ")
29        println("Volume: $volume liters Water: $water liters (${water / volume * 100.0}% full)")
30    }
31 }
  
```

```

1 package example.myapp
2
3 fun main() {
4     buildAquarium()
5 }
6
7 fun buildAquarium() {
8     val aquarium6 = Aquarium(length = 25, width = 25, height = 40)
9     aquarium6.printSize()
10 }
11
12 //fun buildAquarium() {
13 //    val aquarium6 = Aquarium(numberOfFish = 29)
14 //    aquarium6.printSize()
15 //    aquarium6.volume = 70
16 //}
  
```



```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Intel
aquarium initializing
rectangle
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 25 liters Water: 22.5 liters (90.0% full)

Process finished with exit code 0
```

1.5.2. Create a subclass

Các bước thực hiện:

1. Tạo một lớp con của Aquarium có tên là TowerTank, lớp này triển khai một bể hình trụ tròn thay vì bể hình chữ nhật. Bạn có thể thêm TowerTank bên dưới Aquarium, vì bạn có thể thêm một lớp khác trong cùng một tệp với lớp Aquarium.
2. Trong TowerTank, ghi đè thuộc tính height, được định nghĩa trong hàm tạo. Để ghi đè thuộc tính, hãy sử dụng từ khóa override trong lớp con.
3. Các lớp con phải khai báo rõ ràng các tham số hàm tạo của chúng.
4. Làm cho hàm tạo cho TowerTank lấy một đường kính. Sử dụng đường kính cho cả chiều dài và chiều rộng khi gọi hàm tạo trong lớp cha Aquarium.
5. Ghi đè thuộc tính volume để tính một hình trụ. Công thức cho một hình trụ là pi nhân với bán kính bình phương nhân với chiều cao. Lưu ý IntelliJ có thể đánh dấu PI là không xác định. Bạn cần nhập hằng số PI từ java.lang.Math ở đầu Main.kt.
6. Trong TowerTank, ghi đè thuộc tính water thành 80% thể tích.
7. Ghi đè hình dạng thành "hình trụ".
8. Trong buildAquarium(), tạo một TowerTank có đường kính 25 cm và chiều cao 45 cm. In kích thước.
9. Chạy chương trình.

```
new *
open class Aquarium(var length: Int = 100, var width: Int = 20, open var height: Int = 40) {
    new *
    init {
        println("aquarium initializing")
    }
}
```

```

31 class TowerTank (override var height: Int, var diameter: Int)
32     : Aquarium(height = height, width = diameter, length = diameter) {
33     new *
34     override var volume: Int
35         // ellipse area =  $\pi * r1 * r2$ 
36         get() = (width/2 * length/2 * height / 1000 * PI).toInt()
37         set(value) {
38             height = ((value * 1000 / PI) / (width/2 * length/2)).toInt()
39         }
40     override var water = volume * 0.8
41     override val shape = "cylinder"
42 }

```

Project: Lesson3-1 C:\Project\HustMobile\Lesson3\Lesson3-1

- .idea
- out
- src
 - main
 - kotlin
 - example.myapp
 - Aquarium.kt
 - Main.kt
 - resources
 - test
 - .gitignore
 - Lesson3-1.iml
- External Libraries
- Scratches and Consoles

```

1 package example.myapp
2
3 new *
4 fun main() {
5     buildAquarium()
6 }
7
8 new *
9 fun buildAquarium() {
10     val myAquarium = Aquarium(width = 25, length = 25, height = 40)
11     myAquarium.printSize()
12     val myTower = TowerTank(diameter = 25, height = 40)
13     myTower.printSize()
14 }
15
16 //fun buildAquarium() {
17 //    val aquarium6 = Aquarium(length = 25, width = 25, height = 40)
18 //    aquarium6.printSize()
19 //}
20
21 //fun buildAquarium() {
22 //    val aquarium6 = Aquarium(length = 25, width = 25, height = 40)
23 //    aquarium6.printSize()
24 //}

```

Run: MainKt

```

"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\lib\idea_rt.jar=59639:C:\Program Files\Java\bin" -Didea.platform.prefix=java -jar C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\bin\idea_rt.jar 59639
aquarium initializing
rectangle
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 25 liters Water: 22.5 liters (90.0% full)
aquarium initializing
cylinder
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 18 liters Water: 14.4 liters (80.0% full)
Process finished with exit code 0

```

1.6. Compare abstract classes and interfaces

Đôi khi bạn muốn định nghĩa hành vi hoặc thuộc tính chung để chia sẻ giữa một số lớp liên quan. Kotlin cung cấp hai cách để thực hiện điều đó, giao diện và lớp trừu tượng. Trong nhiệm vụ này, bạn tạo một lớp AquariumFish trừu tượng cho các thuộc tính chung

cho tất cả các loài cá. Bạn tạo một giao diện có tên là FishAction để định nghĩa hành vi chung cho tất cả các loài cá.

- Không thể khởi tạo một lớp trừu tượng hoặc giao diện, các lớp trừu tượng có thể có các hàm tạo.
- Vì chúng không phải là lớp, nên giao diện không thể có bất kỳ logic hàm tạo nào
- Giao diện không thể lưu trữ bất kỳ trạng thái nào.

Các lớp trừu tượng là các lớp được định nghĩa một phần. Các lớp con của lớp trừu tượng có trách nhiệm định nghĩa các phương thức và thuộc tính của nó. Các lớp trừu tượng luôn mở; bạn không cần phải đánh dấu chúng bằng open. Các thuộc tính và phương thức của một lớp trừu tượng là không trừu tượng trừ khi bạn đánh dấu chúng một cách rõ ràng bằng từ khóa abstract. Nếu các thuộc tính và phương thức này không có từ khóa abstract, thì các lớp con có thể sử dụng chúng như đã cho. Nếu các thuộc tính hoặc phương thức là trừu tượng, thì các lớp con phải triển khai chúng.

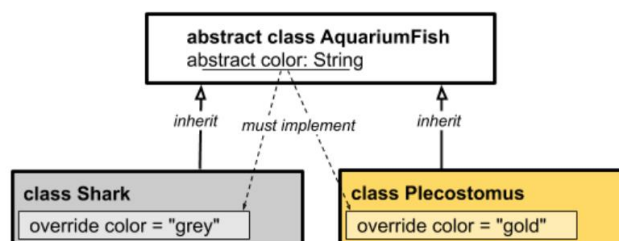
1.6.1. Create an abstract class

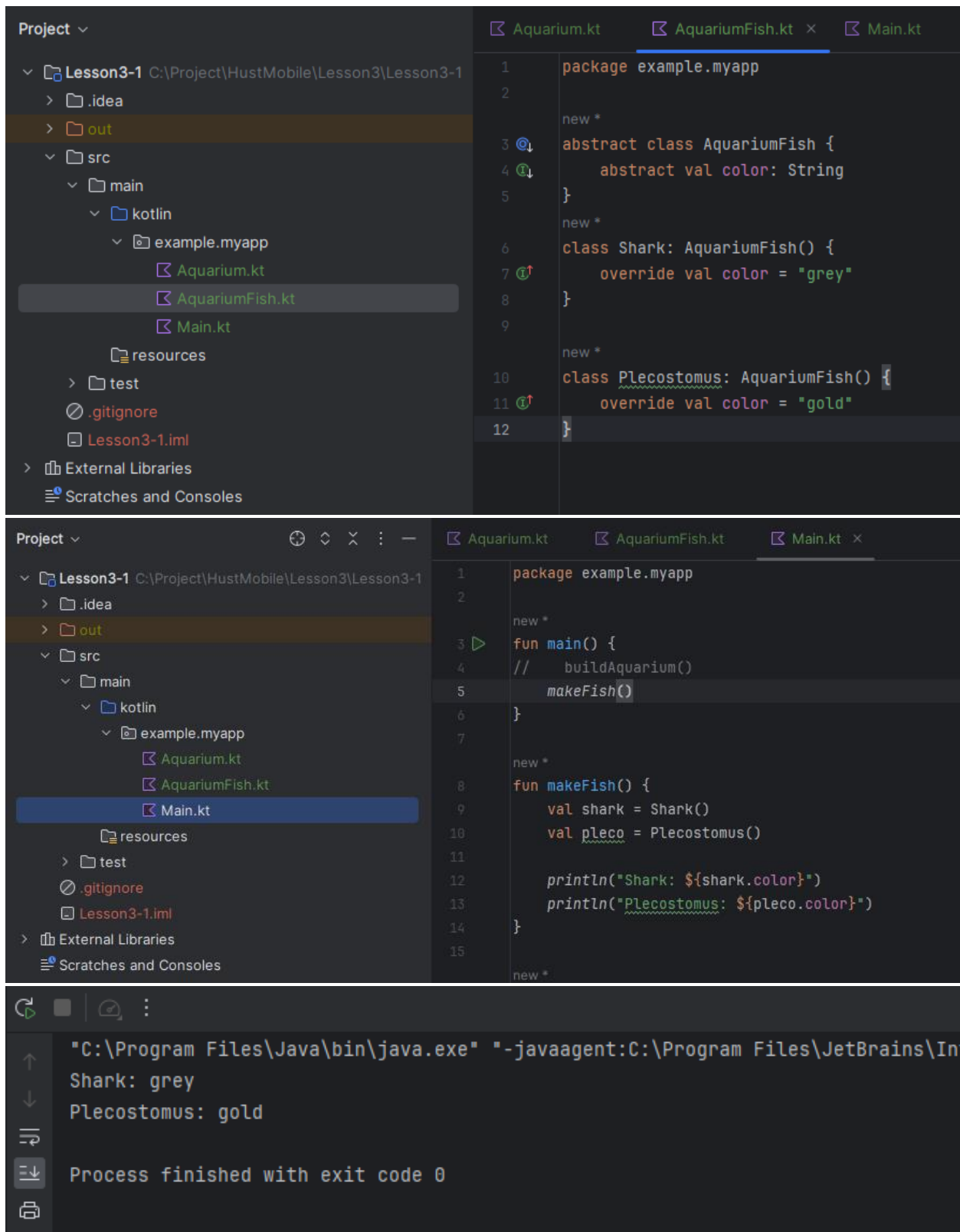
Các bước thực hiện:

1. Trong example.myapp, tạo một tệp mới, AquariumFish.kt.
2. Tạo một lớp, còn được gọi là AquariumFish, và đánh dấu là trừu tượng.
3. Thêm một thuộc tính String, color và đánh dấu là trừu tượng.
4. Tạo hai lớp con của AquariumFish, Shark và Plecostomus.
5. Vì color là trừu tượng, các lớp con phải triển khai nó. Làm cho Shark thành màu xám và Plecostomus thành màu vàng.
6. Trong Main.kt, tạo một hàm makeFish() để kiểm tra các lớp của bạn. Khởi tạo một Shark và một Plecostomus, sau đó in màu của từng lớp.
7. Xóa mã kiểm tra trước đó của bạn trong main() và thêm lệnh gọi đến makeFish(). Mã của bạn sẽ trông giống như mã bên dưới.
8. Chạy chương trình.

Sơ đồ sau đây biểu diễn hệ thống phân cấp lớp của ứng dụng của chúng ta. Cả lớp Shark và lớp Plecostomus đều là lớp con của lớp trừu tượng, AquariumFish.

One abstract class, two subclasses





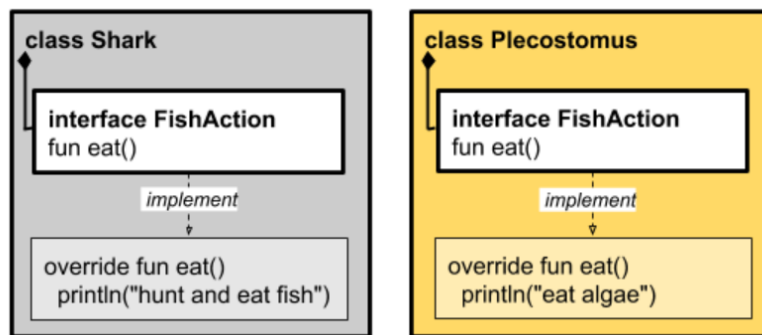
1.6.2. Create an interface

Các bước thực hiện:

1. Trong AquariumFish.kt, tạo một giao diện có tên là FishAction với phương thức eat().
2. Thêm FishAction vào từng lớp con và triển khai eat() bằng cách để nó in ra những gì con cá làm.
3. Trong hàm makeFish() trong Main.kt, để mỗi con cá bạn tạo ra ăn thứ gì đó bằng cách gọi eat().
4. Chạy chương trình.

Sơ đồ sau đây biểu diễn lớp Shark và lớp Plecostomus, cả hai đều triển khai giao diện FishAction.

Two classes, one interface



Project ▾
Lesson3-1 C:\Project\HustMobile\Lesson3\Lesson3-1
 > .idea
 > out
 > src
 > main
 > kotlin
 > example.myapp
 Aquarium.kt
 AquariumFish.kt
 Main.kt
 resources
 test
 .gitignore
 Lesson3-1.iml
 > External Libraries
 Scratches and Consoles

Aquarium.kt AquariumFish.kt x Main.kt
1 package example.myapp
2
3 new *
4 abstract class AquariumFish {
5 abstract val color: String
6 }
7 new *
8 interface FishAction {
9 new *
10 fun eat()
11 }
12 new *
13 class Shark: AquariumFish(), FishAction {
14 override val color = "grey"
15 new *
16 override fun eat() {
17 println("hunt and eat fish")
18 }
19 }
20 new *
21 class Plecostomus: AquariumFish(), FishAction {
22 override val color = "gold"
23 new *
24 override fun eat() {
25 println("eat algae")
26 }
27 }

Project ▾
Lesson3-1 C:\Project\HustMobile\Lesson3\Lesson3-1
 > .idea
 > out
 > src
 > main
 > kotlin
 > example.myapp
 Aquarium.kt
 AquariumFish.kt
 Main.kt
 resources
 test
 .gitignore
 Lesson3-1.iml
 > External Libraries
 Scratches and Consoles

Aquarium.kt AquariumFish.kt Main.kt x
1 package example.myapp
2
3 new *
4 fun main() {
5 // buildAquarium()
6 makeFish()
7 }
8 new *
9 fun makeFish() {
10 val shark = Shark()
11 val pleco = Plecostomus()
12 println("Shark: \${shark.color}")
13 shark.eat()
14 println("Plecostomus: \${pleco.color}")
15 pleco.eat()
16 }

↑
↓
≡
⇅
🖨
🗑

"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae

Process finished with exit code 0

1.6.3. When to use abstract classes versus interfaces

Các ví dụ trên rất đơn giản, nhưng khi bạn có nhiều lớp liên quan, các lớp trừu tượng và giao diện có thể giúp bạn giữ cho thiết kế của mình sạch hơn, có tổ chức hơn và dễ bảo trì hơn.

Như đã lưu ý ở trên, các lớp trừu tượng có thể có các hàm tạo còn giao diện thì không, nhưng ngoài ra chúng rất giống nhau. Vậy, khi nào bạn nên sử dụng từng lớp?

Khi bạn sử dụng giao diện để thiết kế một lớp, chức năng của lớp được mở rộng bằng các phương thức trong giao diện mà lớp đó triển khai. Sử dụng các đặc điểm được xác định trong giao diện sẽ giúp mã dễ sử dụng lại và dễ hiểu hơn so với kế thừa từ một lớp trừu tượng. Ngoài ra, bạn có thể triển khai nhiều giao diện trong một lớp, nhưng bạn chỉ có thể phân lớp từ một lớp. Nguyên tắc chung là ưu tiên thành phần (tức là giao diện và tham chiếu thể hiện) hơn là phân lớp khi có thể.

Sử dụng lớp trừu tượng bất kỳ lúc nào bạn không thể hoàn thành một lớp. Ví dụ, quay lại lớp `AquariumFish`, bạn có thể khiến tất cả `AquariumFish` triển khai `FishAction` và cung cấp triển khai mặc định cho `eat` trong khi để màu trừu tượng, vì thực sự không có màu mặc định cho fish.

```
25     class GoldenFish: AquariumFish() {
26         override val color = "orange"
27     }

8     fun makeFish() {
9         val shark = Shark()
10        val pleco = Plecostomus()
11        val goldenFish = GoldenFish()
12        println("Shark: ${shark.color}")
13        shark.eat()
14        println("Plecostomus: ${pleco.color}")
15        pleco.eat()
16        println("GoldenFish: ${goldenFish.color}")
17        goldenFish.eat()
18    }
```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetB
Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae
GoldenFish: orange
yum
```

1.7. Use interface delegation

Nhiệm vụ trước đã giới thiệu các lớp trừu tượng và giao diện. Giao diện ủy quyền là một kỹ thuật thiết kế nâng cao trong đó các phương thức của giao diện được triển khai bởi một đối tượng trợ giúp (hoặc ủy quyền), sau đó được một lớp sử dụng. Kỹ thuật này có thể hữu ích khi bạn sử dụng một giao diện trong một loạt các lớp không liên quan. Bạn triển khai chức năng giao diện cần thiết trong một lớp trợ giúp riêng biệt. Mỗi lớp không liên quan sau đó sử dụng một thể hiện của lớp trợ giúp đó để có được chức năng.

Trong nhiệm vụ này, bạn sử dụng giao diện ủy quyền để thêm chức năng vào một lớp.

1.7.1. Make a new interface

Các bước thực hiện:

1. Trong AquariumFish.kt, hãy xóa lớp AquariumFish. Thay vì kế thừa từ lớp AquariumFish, Plecostomus và Shark sẽ triển khai giao diện cho cả hành động của cá và màu sắc của chúng.
2. Tạo một giao diện mới, FishColor, định nghĩa màu sắc là một chuỗi.
3. Thay đổi Plecostomus để triển khai hai giao diện, FishAction và FishColor. Bạn cần ghi đè màu sắc từ FishColor và eat() từ FishAction.
4. Thay đổi lớp Shark của bạn để cũng triển khai hai giao diện, FishAction và FishColor, thay vì kế thừa từ AquariumFish.

```
8  interface FishColor {
9      val color: String
10 }
11
12 new *
13 interface FishAction {
14     new *
15     fun eat()
16 }
```



```
16 class Plecostomus: FishAction, FishColor {
17     override val color = "gold"
18     new *
19     override fun eat() {
20         println("eat algae")
21     }
22 }
23
24 new *
25 class Shark: FishAction, FishColor {
26     override val color = "grey"
27     new *
28     override fun eat() {
29         println("hunt and eat fish")
30     }
31 }
```

↑ ↓ ↺ ↻ ⌵ ⌶ ⌷ ⌸

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrain
Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae
GoldenFish: orange
yummy
```

1.7.2. Make a singleton class

Mô tả:

Tiếp theo, bạn triển khai thiết lập cho phần ủy quyền bằng cách tạo một lớp trợ giúp triển khai FishColor. Bạn tạo một lớp cơ bản có tên là GoldColor triển khai FishColor—tất cả những gì nó làm là nói rằng màu của nó là vàng.

Không hợp lý khi tạo nhiều phiên bản GoldColor, vì tất cả chúng đều thực hiện chính xác cùng một việc. Vì vậy, Kotlin cho phép bạn khai báo một lớp mà bạn chỉ có thể tạo một phiên bản của nó bằng cách sử dụng từ khóa object thay vì class. Kotlin sẽ tạo một phiên bản đó và phiên bản đó được tham chiếu theo tên lớp. Sau đó, tất cả các đối tượng khác chỉ có thể sử dụng phiên bản này. Bạn không thể tạo các phiên bản khác của lớp này. Nếu bạn quen thuộc với mẫu singleton, đây là cách bạn triển khai singleton trong Kotlin.

Các bước thực hiện: Trong AquariumFish.kt, tạo một đối tượng cho GoldColor. Ghi đè màu.

```

16      object GoldColor : FishColor {
17          override val color = "gold"
18      }

```

1.7.3. Add interface delegation for FishColor

Các bước thực hiện:

1. Trong AquariumFish.kt, xóa ghi đè màu khỏi Plecostomus.
2. Thay đổi lớp Plecostomus để lấy màu của nó từ GoldColor. Bạn thực hiện việc này bằng cách thêm by GoldColor vào khai báo lớp, tạo ra ủy quyền. Điều này có nghĩa là thay vì triển khai FishColor, hãy sử dụng triển khai do GoldColor cung cấp. Vì vậy, mỗi khi truy cập màu, nó sẽ được ủy quyền cho GoldColor.
3. Thay đổi lớp Plecostomus để lấy fishColor được truyền vào với hàm tạo của nó và đặt mặc định của nó thành GoldColor. Thay đổi sự ủy quyền từ by GoldColor thành by fishColor.

```

new *
16      object GoldColor : FishColor {
17          override val color = "gold"
18      }
19
new *
20      class Plecostomus: FishAction, FishColor by GoldColor {
new *
21          override fun eat() {
22              println("eat algae")
23          }
24      }

```

Run MainKt x

```

"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\Jet
Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae
GoldenFish: orange
yummy
>

```

```
16      object GoldColor : FishColor {
17          override val color = "gold"
18      }
19
20      new *
21      class Plecostomus(fishColor: FishColor = GoldColor): FishAction,
22          FishColor by fishColor {
23          new *
24          override fun eat() {
25              println("eat algae")
26          }
27      }
```

Run MainKt x

↑ "C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\Jet
Shark: grey
↓ hunt and eat fish
⇌ Plecostomus: gold
⇓ eat algae
☰ GoldenFish: orange
> yummy

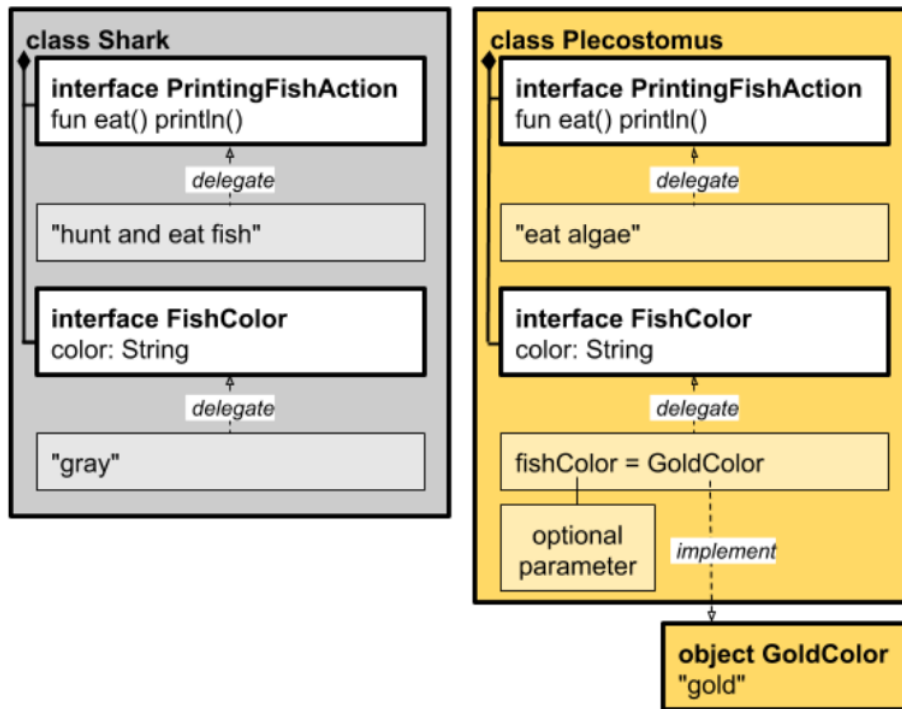
1.7.4. Add interface delegation for FishAction

Các bước thực hiện:

1. Trong AquariumFish.kt, tạo một lớp PrintingFishAction triển khai FishAction, lấy một String food làm tham số khởi tạo, sau đó in ra những gì cá ăn.
2. Trong lớp Plecostomus, xóa hàm ghi đè eat(), vì bạn sẽ thay thế nó bằng một lệnh ủy quyền.
3. Trong khai báo của Plecostomus, ủy quyền FishAction cho PrintingFishAction, truyền "eat algae".
4. Với tất cả các lệnh ủy quyền đó, không có mã nào trong phần thân của lớp Plecostomus, vì vậy hãy xóa {}, vì tất cả các lệnh ghi đè đều được xử lý bằng lệnh ủy quyền giao diện.

Nếu bạn tạo một thiết kế tương tự cho Shark, sơ đồ sau sẽ biểu diễn cả lớp Shark và Plecostomus. Cả hai đều bao gồm các giao diện PrintingFishAction và FishColor, nhưng ủy quyền việc triển khai cho chúng

Two classes, two interfaces with delegation



```

16      class PrintingFishAction(val food: String) : FishAction {
17          new *
18              override fun eat() {
19                  println(food)
20              }
21      }
22
23      new *
24      object GoldColor : FishColor {
25          override val color = "gold"
26      }
27
28      new *
29      class Plecostomus (fishColor: FishColor = GoldColor):
30          FishAction by PrintingFishAction( food: "eat algae"),
31          FishColor by fishColor

```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae
GoldenFish: orange
yummy
```

1.8. Create a data class

Một lớp dữ liệu tương tự như một struct trong một số ngôn ngữ khác. Nó tồn tại chủ yếu để chứa một số dữ liệu. Các lớp dữ liệu Kotlin đi kèm với một số lợi ích bổ sung, chẳng hạn như các tiện ích để in và sao chép. Trong nhiệm vụ này, bạn tạo một lớp dữ liệu đơn giản và tìm hiểu về hỗ trợ mà Kotlin cung cấp cho các lớp dữ liệu.

1.8.1. Create a data class

Các bước thực hiện:

1. Thêm một gói decor mới trong gói example.myapp để chứa mã mới. Nhấp chuột phải vào example.myapp trong ngăn Project và chọn File > New > Package.
2. Trong gói, tạo một lớp mới có tên là Decoration.
3. Để biến Decoration thành một lớp dữ liệu, hãy thêm tiền tố khai báo lớp bằng từ khóa data.
4. Thêm một thuộc tính String có tên là rocks để cung cấp cho lớp một số dữ liệu.
5. Trong tệp, bên ngoài lớp, thêm hàm makeDecorations() để tạo và in một thể hiện của Decoration với "granite".
6. Thêm hàm main() để gọi makeDecorations() và chạy chương trình của bạn. Lưu ý đầu ra hợp lý được tạo ra vì đây là một lớp dữ liệu.
7. Trong makeDecorations(), khởi tạo thêm hai đối tượng Decoration đều là "slate" và in chúng.
8. Trong makeDecorations(), thêm một câu lệnh print để in kết quả so sánh decoration1 với decoration2 và một câu lệnh thứ hai so sánh decoration3 với decoration2. Sử dụng phương thức equals() do các lớp dữ liệu cung cấp.
9. Chạy mã.

Lưu ý: Bạn có thể sử dụng == để kiểm tra xem decoration1 == decoration2 và decoration3 == decoration2. Trong Kotlin, sử dụng == trên các đối tượng lớp dữ liệu giống như sử dụng equals() (bằng nhau về mặt cấu trúc). Nếu bạn cần kiểm tra xem hai biến có tham chiếu đến cùng một đối tượng hay không (bằng nhau về mặt tham chiếu), hãy sử dụng toán tử ===. Đọc thêm về bằng nhau trong Kotlin trong tài liệu ngôn ngữ.

Lưu ý: Mặc dù chúng tương tự như struct trong một số ngôn ngữ, nhưng hãy nhớ rằng các đối tượng lớp dữ liệu là các đối tượng. Việc gán một đối tượng lớp dữ liệu cho một biến khác sẽ sao chép tham chiếu đến đối tượng đó, không phải nội dung. Để sao chép nội dung vào một đối tượng mới, hãy sử dụng phương thức `copy()`.

Cảnh báo: `Copy()`, `equals()` và các tiện ích lớp dữ liệu khác chỉ tham chiếu đến các thuộc tính được xác định trong hàm tạo chính.

```
1 package example.myapp.decor
2 new *
3 data class Decoration(val rocks: String) {
4
5 }
6
7 new *
8 fun makeDecorations() {
9     val decoration1 = Decoration( rocks: "granite")
10    println(decoration1)
11    val decoration2 = Decoration( rocks: "slate")
12    println(decoration2)
13    val decoration3 = Decoration( rocks: "slate")
14    println(decoration3)
15    println (decoration1.equals(decoration2))
16    println (decoration3.equals(decoration2))
17 }
18
19 new *
20 fun main(){
21     makeDecorations()
22 }
23 }
```

↑ "C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Int
↓ Decoration(rocks=granite)
↔ Decoration(rocks=slate)
↔ Decoration(rocks=slate)
⇓ false
⇓ true
🗑 Process finished with exit code 0

1.8.2. Use destructuring

```
5 data class Decoration2(val rocks: String, val wood: String, val diver: String){
6 }
7
8 new *
9 fun makeDecorations() {
10     val d5 = Decoration2( rocks: "crystal", wood: "wood", diver: "diver")
11     println(d5)
12
13     // Assign all properties to variables.
14     val (rock, wood, diver) = d5
15     println(rock)
16     println(wood)
17     println(diver)
18 }
19
```

```
↑ "C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\bin\jetbrains-agent.jar" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA\conf -Didea.home.path=C:\Program Files\JetBrains\IntelliJ IDEA\bin -Didea.platform.prefix=Java -jar C:\Program Files\JetBrains\IntelliJ IDEA\bin\idea64.exe
↓ Decoration2(rocks=crystal, wood=wood, diver=diver)
⇌ crystal
⇌ wood
⇌ diver
⇌
⇌ Process finished with exit code 0
⇌
```

1.9. Learn about singletons and enums

1.9.1. Recall singleton classes

Nhớ lại ví dụ trước đó với lớp GoldColor.

```
22 object GoldColor : FishColor {
23     override val color = "gold"
24 }
25
```

Vì mọi trường hợp của GoldColor đều thực hiện cùng một việc, nên nó được khai báo là một đối tượng thay vì là một lớp để biến nó thành một singleton. Chỉ có thể có một trường hợp của nó.

1.9.2. Create an enum

Kotlin cũng hỗ trợ enum, là một tập hợp các giá trị hoặc hằng số được đặt tên. Enum là một loại lớp đặc biệt trong Kotlin cho phép bạn tham chiếu đến giá trị theo tên, giống như trong các ngôn ngữ khác. Chúng có thể tăng cường khả năng đọc mã của bạn. Mỗi hằng số trong enum là một đối tượng. Khai báo enum bằng cách thêm tiền tố khai báo bằng

từ khóa enum. Một khai báo enum cơ bản chỉ cần một danh sách các tên, nhưng bạn cũng có thể xác định một hoặc nhiều trường được liên kết với mỗi tên.

Enum tương tự như singleton—chỉ có thể có một và chỉ một giá trị trong phép liệt kê. Ví dụ, chỉ có thể có một `Color.RED`, một `Color.GREEN` và một `Color.BLUE`. Trong ví dụ này, các giá trị RGB được gán cho thuộc tính `rgb` để biểu diễn các thành phần màu. Có những đặc điểm hữu ích khác của enum. Ví dụ, bạn có thể lấy giá trị thứ tự của một enum bằng cách sử dụng thuộc tính thứ tự và tên của nó bằng cách sử dụng thuộc tính `name`.

Các bước thực hiện:

1. Trong `Decoration.kt`, hãy thử một ví dụ về enum.
2. Hãy thử một ví dụ khác về enum trong REPL.

```
8  enum class Color(val rgb: Int) {
9      RED( rgb: 0xFF0000), GREEN( rgb: 0x00FF00), BLUE( rgb: 0x0000FF);
10 }
11
12 new *
13 enum class Direction(val degrees: Int) {
14     NORTH( degrees: 0), SOUTH( degrees: 180), EAST( degrees: 90), WEST( degrees: 270)
15 }
16
38 fun main() {
39     makeDecorations()
40     println(Direction.EAST.name)
41     println(Direction.EAST.ordinal)
42     println(Direction.EAST.degrees)
43 }
```

Output:

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\bin\idea_rt.jar"
Decoration2(rocks=crystal, wood=wood, diver=diver)
crystal
wood
diver
EAST
2
90

Process finished with exit code 0
```


2. Pairs/triples, collections, constants, and writing extension functions

2.1. Create a Companion Object

Đôi khi bạn cần một hàm hoặc thuộc tính singleton được liên kết với một lớp chứ không phải một thể hiện. Trong các ngôn ngữ khác như Java, bạn có thể sử dụng thành viên tĩnh. Kotlin cung cấp đối tượng đồng hành cho mục đích này. Đối tượng đồng hành không phải là một thể hiện và không có nghĩa là được sử dụng riêng lẻ.

Đối tượng đồng hành là đối tượng Kotlin thực sự. Chúng có thể triển khai giao diện và mở rộng các lớp khiến chúng có rất nhiều tính năng trong khi vẫn tiết kiệm bộ nhớ với một đối tượng đơn lẻ.

Sự khác biệt cơ bản giữa đối tượng đồng hành và đối tượng thông thường là:

1. Đối tượng đồng hành được khởi tạo từ hàm tạo tĩnh của lớp chứa, nghĩa là chúng được tạo khi đối tượng được tạo.
2. Đối tượng thông thường được khởi tạo một cách lười biếng khi truy cập lần đầu vào đối tượng đó; nghĩa là khi chúng được sử dụng lần đầu.
3. Còn nhiều điều nữa, nhưng tất cả những gì bạn cần biết hiện tại là cách đóng gói hằng số trong các lớp trong đối tượng đồng hành.

```
45 class Choice {
46     new *
47     companion object {
48         var name: String = "lyric"
49         new *
50         fun showDescription(name:String) = println("My favorite $name")
51     }
52 }
53
54 new *
55 fun main() {
56     println(Choice.name)
57     Choice.showDescription( name: "pick")
58     Choice.showDescription( name: "selection")
59 }
```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBr
lyric
My favorite pick
My favorite selection
```

2.2. Learn about pairs and triples

Trong nhiệm vụ này, bạn sẽ tìm hiểu về cặp và bộ ba, cũng như cách sử dụng chúng. Cặp và bộ ba là các lớp dữ liệu được tạo sẵn cho 2 hoặc 3 mục chung. Ví dụ, điều này có thể hữu ích khi có một hàm trả về nhiều hơn một giá trị.

2.2.1. Make some pairs and triples

Các bước thực hiện:

1. Mở REPL (Công cụ > Kotlin > Kotlin REPL).
2. Tạo một cặp, liên kết một thiết bị với mục đích sử dụng của nó, sau đó in các giá trị. Bạn có thể tạo một cặp bằng cách tạo một biểu thức kết nối hai giá trị, chẳng hạn như hai chuỗi, với từ khóa `to`, sau đó sử dụng `.first` hoặc `.second` để tham chiếu đến từng giá trị.
3. Tạo một bộ ba và in nó bằng `toString()`, sau đó chuyển đổi nó thành một danh sách bằng `toList()`. Bạn tạo một bộ ba bằng `Triple()` với 3 giá trị. Sử dụng `.first`, `.second` và `.third` để tham chiếu đến từng giá trị.
4. Tạo một cặp trong đó phần đầu tiên của cặp chính là một cặp.

```
val equipment = "fish net" to "catching fish"
println("${equipment.first} used for ${equipment.second}")
fish net used for catching fish

val numbers = Triple(6, 9, 42)
println(numbers.toString())
println(numbers.toList())
(6, 9, 42)[0, 9, 42]
```

Aquarium.kt AquariumFish.kt Main.kt Decoration.kt x

```
50      }
      new *
51  ▶ fun main() {
52      println(Choice.name)
53      Choice.showDescription( name: "pick")
54      Choice.showDescription( name: "selection")
55      println("-----")
56      val equipment = "fish net" to "catching fish"
57      println("${equipment.first} used for ${equipment.second}")
58      println("-----")
59      val numbers = Triple( first: 6, second: 9, third: 42)
60      println(numbers.toString())
61      println(numbers.toList())
62  }
```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 26
lyric
My favorite pick
My favorite selection
-----
fish net used for catching fish
-----
(6, 9, 42)
[6, 9, 42]

Process finished with exit code 0
```

2.2.2. Destructure some pairs and triples

Mô tả: Việc tách cặp và bộ ba thành các phần của chúng được gọi là giải cấu trúc. Gán cặp hoặc bộ ba cho số lượng biến thích hợp và Kotlin sẽ gán giá trị của từng phần theo thứ tự.

Các bước thực hiện:

1. Phân tách một cặp và in các giá trị.
2. Phân tách một bộ ba và in các giá trị.
3. Lưu ý rằng việc phân tách các cặp và bộ ba hoạt động giống như với các lớp dữ liệu, đã được đề cập trong một bài học codelab trước đó.

```
Aquarium.kt  AquariumFish.kt  Main.kt  Decoration.kt x
new *
51 fun main() {
52     println(Choice.name)
53     Choice.showDescription(name: "pick")
54     Choice.showDescription(name: "selection")
55     println("-----")
56     val equipment = "fish net" to "catching fish"
57     val (tool, use) = equipment
58     println("$tool is used for $use")
59     // val equipment = "fish net" to "catching fish"
60     // println("${equipment.first} used for ${equipment.second}")
61     println("-----")
62     val numbers = Triple(first: 6, second: 9, third: 42)
63     val (n1, n2, n3) = numbers
64     println("$n1 $n2 $n3")
65     // val numbers = Triple(6, 9, 42)
66     // println(numbers.toString())
67     // println(numbers.toList())
68 }
```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\In
lyric
My favorite pick
My favorite selection
-----
fish net is used for catching fish
-----
ó 9 42

Process finished with exit code 0
```

2.3. Learn more about collections

Trong nhiệm vụ này, bạn sẽ tìm hiểu thêm về các bộ sưu tập, bao gồm danh sách và một loại bộ sưu tập mới, HashMap.

2.3.1. Understand more about lists

Các bước thực hiện:

1. Danh sách và danh sách có thể thay đổi đã được giới thiệu trong bài học trước. Chúng là những cấu trúc dữ liệu được sử dụng rất phổ biến, vì vậy Kotlin cung cấp một số hàm tích hợp cho chúng. Xem lại danh sách một phần các hàm cho danh sách này. Bạn có thể tìm thấy danh sách đầy đủ trong tài liệu Kotlin cho cả List và MutableList.
2. Vẫn đang làm việc trong REPL, hãy tạo một danh sách các số và gọi sum() trên đó. Thao tác này sẽ tính tổng tất cả các phần tử.
3. Tạo một danh sách các chuỗi và tính tổng danh sách.
4. Nếu phần tử không phải là thứ mà List biết cách tính tổng trực tiếp, chẳng hạn như chuỗi, bạn có thể chỉ định cách tính tổng bằng cách sử dụng .sumBy() với hàm lambda, ví dụ, để tính tổng theo độ dài của mỗi chuỗi. Hãy nhớ từ một codelab trước đó, tên mặc định cho đối số lambda là it. Ở đây, nó tham chiếu đến từng phần tử của danh sách khi duyệt qua danh sách.
5. Bạn có thể làm nhiều hơn thế nữa với danh sách. Một cách để xem chức năng khả dụng là tạo một danh sách trong IntelliJ IDEA, thêm dấu chấm, sau đó xem danh sách tự động hoàn thành trong chú giải công cụ. Điều này có hiệu quả với bất kỳ đối tượng nào. Hãy thử với một danh sách.
6. Để xem chức năng của một lớp, hãy tạo một đối tượng trong IntelliJ IDEA, thêm dấu chấm sau tên, sau đó xem danh sách tự động hoàn thành trong chú giải công cụ. Điều này có hiệu quả với bất kỳ đối tượng nào.

7. Chọn `listIterator()` từ danh sách, sau đó duyệt qua danh sách bằng câu lệnh `for` và in tất cả các phần tử được phân tách bằng dấu cách.

```
val list = listOf(1, 5, 3, 4)
println(list.sum())
13


val list2 = listOf("a", "bbb", "cc")
println(list2.sum())
error: unresolved reference. None of the following candidates is applicable because of receiver type mismatch:
public fun Array<out Byte>.sum(): Int defined in kotlin.collections
public fun Array<out Double>.sum(): Double defined in kotlin.collections
public fun Array<out Float>.sum(): Float defined in kotlin.collections
public fun Array<out Int>.sum(): Int defined in kotlin.collections
public fun Array<out Long>.sum(): Long defined in kotlin.collections
```

```
val list2 = listOf("a", "bbb", "cc")
println(list2.sumBy { it.length })
```

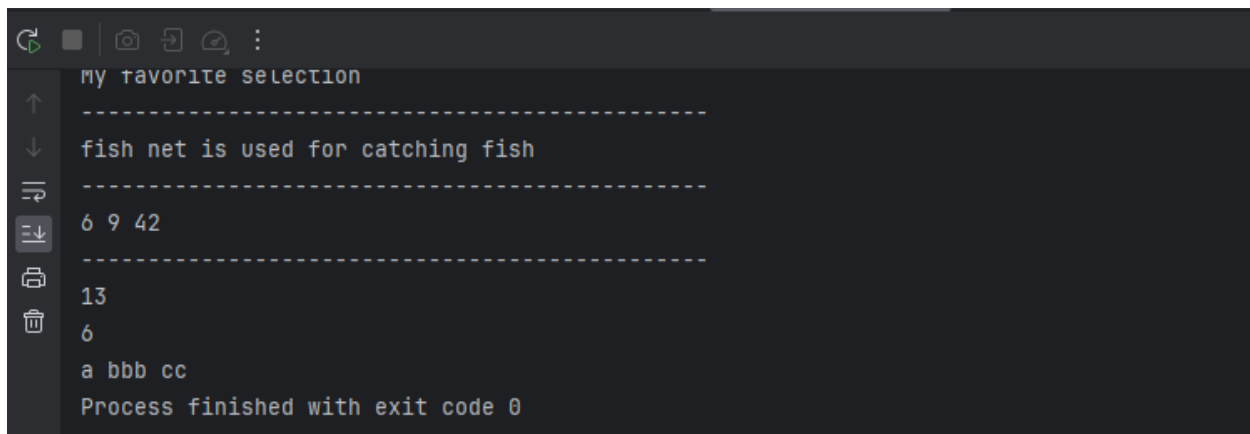
6

```
val list2 = listOf("a", "bbb", "cc")
for (s in list2.listIterator()) {
    println("$s ")
}
```

a bbb cc

 <Ctrl+Enter> to execute

```
Aquarium.kt  AquariumFish.kt  Main.kt  Decoration.kt x
66 //      println(numberOfDecorating())
67 //      println(numbers.toList())
68      println("-----")
69      val list = listOf(1, 5, 3, 4)
70      println(list.sum())
71 //      val list2 = listOf("a", "bbb", "cc")
72 //      println(list2.sum())
73      val list2 = listOf("a", "bbb", "cc")
74      println(list2.sumBy { it.length })
75      for (s in list2.listIterator()) {
76          print("$s ")
77      }
78  }
```



```
my favorite selection
-----
fish net is used for catching fish
-----
6 9 42
-----
13
-----
6
-----
a bbb cc
Process finished with exit code 0
```

2.3.2. Try out hash maps

Bản đồ băm là một cấu trúc dữ liệu hữu ích khác. Chúng cho phép bạn lưu trữ một giá trị và một đối tượng thứ cấp mà bạn có thể sử dụng để tham chiếu đến các giá trị đã lưu trữ. Ví dụ, nếu bạn muốn lưu trữ chiều cao của mọi người trong lớp hoặc thị trấn của mình và không cần biết ai là ai, bạn có thể lưu trữ chiều cao trong Danh sách. Nếu bạn muốn lưu trữ tên của một người, bạn có thể lưu trữ tên của người đó làm khóa và chiều cao làm giá trị. Trong Kotlin, bạn có thể tạo bản đồ băm liên kết (hoặc ánh xạ) hầu như bất kỳ thứ gì với bất kỳ thứ gì khác bằng cách sử dụng `hashMapOf()`. Bản đồ băm là danh sách các cặp, trong đó giá trị đầu tiên đóng vai trò là khóa tra cứu cho giá trị thứ hai.

Các bước thực hiện:

1. Tạo một bản đồ băm khớp với tên cá phổ biến (các khóa) và tên khoa học của những loài cá này (các giá trị).
2. Sau đó, bạn có thể truy xuất giá trị tên khoa học dựa trên khóa tên cá phổ biến, bằng cách sử dụng `get()` hoặc thậm chí ngắn hơn, dấu ngoặc vuông `[]`.
3. Hãy thử chỉ định một tên cá không có trong bản đồ.
4. Hãy thử tra cứu một khóa không khớp, bằng cách sử dụng `getOrElse()`.
5. Nếu bạn cần làm nhiều hơn là chỉ trả về một giá trị, Kotlin cung cấp hàm `getOrElse()`.
6. Hãy thay đổi mã của bạn để sử dụng `getOrElse()` thay vì `getOrElseDefault()`.

Thay vì trả về một giá trị mặc định đơn giản, bất kỳ mã nào nằm giữa dấu ngoặc nhọn `{}` đều được thực thi. Trong ví dụ, `else` chỉ trả về một chuỗi, nhưng nó có thể phức tạp như tìm một trang web có mô tả khoa học chi tiết và trả về mô tả đó.

Giống như `mutableListOf`, bạn cũng có thể tạo `mutableMapOf`. Một bản đồ có thể thay đổi cho phép bạn đặt và xóa các mục. Mutable chỉ có nghĩa là có thể thay đổi, immutable có nghĩa là không thể thay đổi.

Các bộ sưu tập bất biến đặc biệt hữu ích trong môi trường có luồng, nơi có thể xảy ra sự cố nếu nhiều luồng cùng chạm vào một bộ sưu tập.

The screenshot shows an IDE with a Kotlin project. The main editor displays a file named `Decoration.kt` with the following code:

```
42 val numbers = Triple(100, 9, 42)
43 val (n1, n2, n3) = numbers
44 println("$n1 $n2 $n3")
45 // val numbers = Triple(0, 9, 42)
46 // println(numbers.toString())
47 // println(numbers.toList())
48 // println("-----")
49 val list = listOf(1, 5, 3, 4)
50 println(list.sum())
51 // val list2 = listOf("a", "bbb", "cc")
52 // println(list2.sum())
53 val list2 = listOf("a", "bbb", "cc")
54 println(list2.sumBy { it, length })
55 for (s in list2.iterator()) {
56     print("$s ")
57 }
58 println("-----")
59 val scientific = hashMapOf(
60     "guppy" to "poecilia reticulata", "catfish" to "corydoras", "zebra fish" to "danio rerio"
61 )
62 println(scientific.get("guppy"))
63 println(scientific.get("zebra fish"))
64 println(scientific.get("swordtail"))
65 println(scientific.getOrElse(key="swordtail", defaultValue="sorry, I don't know"))
66 println(scientific.getOrDefault(key="swordtail") {"sorry, I don't know"})
67 }
```

The Build Output window shows two errors:

- Kotlin: Unresolved reference: swordtail:84
- Kotlin: Unsupported literal prefixes and suffixes: 84

The REPL window shows the output of the program:

```
13
6
a bbb cc
-----
poecilia reticulata
danio rerio
sorry, I don't know
sorry, I don't know
Process finished with exit code 0
```

2.4. Organize and define constants

Trong nhiệm vụ này, bạn sẽ tìm hiểu về hằng số trong Kotlin và những cách khác nhau để sắp xếp chúng.

2.4.1. Learn about const vs. val

Trong REPL, hãy thử tạo một hằng số số. Trong Kotlin, bạn có thể tạo các hằng số cấp cao nhất và gán cho chúng một giá trị tại thời điểm biên dịch bằng cách sử dụng `const` và `val`.

Giá trị được gán và không thể thay đổi, nghe có vẻ giống như khai báo một `val` thông thường. Vậy sự khác biệt giữa `const` và `val` là gì? Giá trị của `const` và `val` được xác định

tại thời điểm biên dịch, trong khi giá trị của val được xác định trong quá trình thực thi chương trình, điều đó có nghĩa là val có thể được gán bởi một hàm tại thời điểm chạy.

Điều đó có nghĩa là val có thể được gán một giá trị từ một hàm, nhưng const val thì không.

Ngoài ra, const val chỉ hoạt động ở cấp cao nhất và trong các lớp singleton được khai báo bằng object, không phải với các lớp thông thường. Bạn có thể sử dụng điều này để tạo một tập hoặc đối tượng singleton chỉ chứa các hằng số và nhập chúng khi cần.

```
89  const val rocks = 3
90  val value1 = complexFunctionCall() // OK
91  const val CONSTANT1 = complexFunctionCall() // NOT ok
    new *
92  object Constants {
93      const val CONSTANT2 = "object constant"
94  }
95  val foo = Constants.CONSTANT2
```

2.4.2. Create a companion object

Kotlin không có khái niệm về hằng số cấp lớp. Để định nghĩa hằng số bên trong một lớp, bạn phải gói chúng vào các đối tượng đồng hành được khai báo bằng từ khóa companion. Về cơ bản, đối tượng companion là một đối tượng singleton bên trong lớp.

Tạo một lớp với một đối tượng companion chứa một hằng số chuỗi.

Sự khác biệt cơ bản giữa các đối tượng companion và các đối tượng regular là:

- Các đối tượng companion được khởi tạo từ hàm tạo tĩnh của lớp chứa, nghĩa là chúng được tạo khi đối tượng được tạo.
- Các đối tượng regular được khởi tạo một cách lười biếng khi truy cập lần đầu vào đối tượng đó; nghĩa là khi chúng được sử dụng lần đầu.
- Còn nhiều điều nữa, nhưng tất cả những gì bạn cần biết hiện tại là gói các hằng số trong các lớp trong một đối tượng companion.

```
97  class MyClass {
    new *
98      companion object {
99          const val CONSTANT3 = "constant in companion"
100      }
101  }
```

2.5. Understand extension functions

Trong nhiệm vụ này, bạn sẽ tìm hiểu về việc mở rộng hành vi của các lớp. Việc viết các hàm tiện ích để mở rộng hành vi của một lớp là rất phổ biến. Kotlin cung cấp cú pháp thuận tiện để khai báo các hàm tiện ích này và gọi chúng là các hàm mở rộng.

Các hàm mở rộng cho phép bạn thêm các hàm vào một lớp hiện có mà không cần phải truy cập vào mã nguồn của lớp đó. Ví dụ: bạn có thể khai báo chúng trong tệp `Extensions.kt` là một phần của gói của bạn. Điều này thực sự không sửa đổi lớp, nhưng nó cho phép bạn sử dụng ký hiệu dấu chấm khi gọi hàm trên các đối tượng của lớp đó.

2.5.1. Write an extension function

Các bước thực hiện:

1. String là một kiểu dữ liệu có giá trị trong Kotlin với nhiều hàm hữu ích. Nhưng nếu chúng ta cần một số chức năng String bổ sung không có sẵn trực tiếp thì sao? Ví dụ, chúng ta có thể muốn xác định xem một String có bất kỳ khoảng trắng không.
2. Vẫn đang làm việc trong REPL, hãy viết một hàm mở rộng đơn giản cho lớp String, `hasSpaces()` để kiểm tra xem một chuỗi có chứa khoảng trắng không. Tên hàm được thêm tiền tố là lớp mà nó hoạt động.
3. Bạn có thể đơn giản hóa hàm `hasSpaces()`. This không cần thiết một cách rõ ràng và hàm có thể được rút gọn thành một biểu thức duy nhất và trả về.

```
107 fun String.hasSpaces(): Boolean {
108     val found = this.indexOf(char: ' ')
109     // also valid: this.indexOf(" ")
110     // returns positive number index in String or -1 if not found
111     return found != -1
112 }
new *
113 fun String.hasSpaces2() = indexOf(string: " ") != -1
114
println("\n-----")
var stringTest : String = "Xin chao Le Phuc Hung"
println(stringTest + " co khoang trang: " + stringTest.hasSpaces())
println(stringTest + " co khoang trang: " + stringTest.hasSpaces2())
}
```

2.5.2. Learn the limitations of extensions

Các hàm mở rộng chỉ có quyền truy cập vào API công khai của lớp mà chúng đang mở rộng. Các thành viên riêng tư không thể được truy cập.

Các bước thực hiện:

1. Hãy thử thêm các hàm mở rộng gọi một thuộc tính được đánh dấu là riêng tư.

2. Các hàm mở rộng được giải quyết tĩnh, tại thời điểm biên dịch, dựa trên kiểu của biến.
3. Hãy xem xét mã bên dưới và tìm ra nội dung sẽ in.
4. `plant.print()` in `GreenLeafyPlant`. Bạn có thể mong đợi `aquariumPlant.print()` cũng in `GreenLeafyPlant`, vì nó được gán giá trị là `plant`. Nhưng kiểu được giải quyết tại thời điểm biên dịch, do đó `AquariumPlant` được in.

The image shows three screenshots of an IDE (IntelliJ IDEA) displaying Kotlin code in a project named 'Lesson3-1'.

Top Screenshot: The file explorer on the left shows the project structure. The code editor on the right shows the `AquariumPlant.kt` file with the following code:

```

1 package example.myapp
2
3 new *
4 class AquariumPlant(val color: String, private val size: Int)
5 new *
6 fun AquariumPlant.isRed() = color == "red" // OK
7 new *
8 fun AquariumPlant.isBig() = size > 50 // gives error

```

Middle Screenshot: The file explorer shows the project structure. The code editor on the right shows the `AquariumPlant.kt` file with the following code:

```

1 package example.myapp
2
3 new *
4 open class AquariumPlant(val color: String, private val size: Int)
5 new *
6 class GreenLeafyPlant(size: Int) : AquariumPlant(color: "green", size)
7 new *
8 fun AquariumPlant.print() = println("AquariumPlant")
9 new *
10 fun GreenLeafyPlant.print() = println("GreenLeafyPlant")
11

```

Bottom Screenshot: The file explorer shows the project structure. The code editor on the right shows the `Main.kt` file with the following code:

```

1 package example.myapp
2
3 new *
4 fun main() {
5 // buildAquarium()
6 // makeFish()
7 val plant = GreenLeafyPlant(size = 10)
8 plant.print()
9 println("\n")
10 val aquariumPlant: AquariumPlant = plant
11 aquariumPlant.print() // what will it print?
12 }

```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\lib\idea_rt.jar=
GreenLeafyPlant
AquariumPlant
Process finished with exit code 0
```

2.5.3. Add an extension property

Ngoài các hàm mở rộng, Kotlin còn cho phép bạn thêm các thuộc tính mở rộng. Giống như các hàm mở rộng, bạn chỉ định lớp bạn đang mở rộng, theo sau là dấu chấm, theo sau là tên thuộc tính.

Các bước thực hiện:

1. Vẫn đang làm việc trong REPL, thêm thuộc tính mở rộng `isGreen` vào `AquariumPlant`, thuộc tính này là `true` nếu màu là xanh lá cây.
2. In thuộc tính `isGreen` cho biến `aquariumPlant` và quan sát kết quả.

The screenshot shows the IntelliJ IDEA interface with two tabs open: `AquariumPlant.kt` and `Main.kt`. The `Project` view on the left shows the project structure for `Lesson3-1`.

AquariumPlant.kt

```
1 package example.myapp
2
3 new *
4 @
5 open class AquariumPlant(val color: String, private val size: Int)
6
7 new *
8 class GreenLeafyPlant(size: Int) : AquariumPlant( color="green", size)
9
10 new *
11 fun AquariumPlant.print() = println("AquariumPlant")
12 new *
13 fun GreenLeafyPlant.print() = println("GreenLeafyPlant")
14
15 new *
16 val AquariumPlant.isGreen: Boolean
17     get() = color == "green"
```

Main.kt

```
1 package example.myapp
2
3 new *
4 fun main() {
5     // buildAquarium()
6     // makeFish()
7     val plant = GreenLeafyPlant(size = 10)
8     plant.print()
9     println("\n")
10     val aquariumPlant: AquariumPlant = plant
11     aquariumPlant.print() // what will it print?
12     println("aquariumPlant.isGreen = " + aquariumPlant.isGreen)
13 }
```

```
"C:\Program Files\Java\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.2\lib\
GreenLeafyPlant

 AquariumPlant
 aquariumPlant.isGreen = true

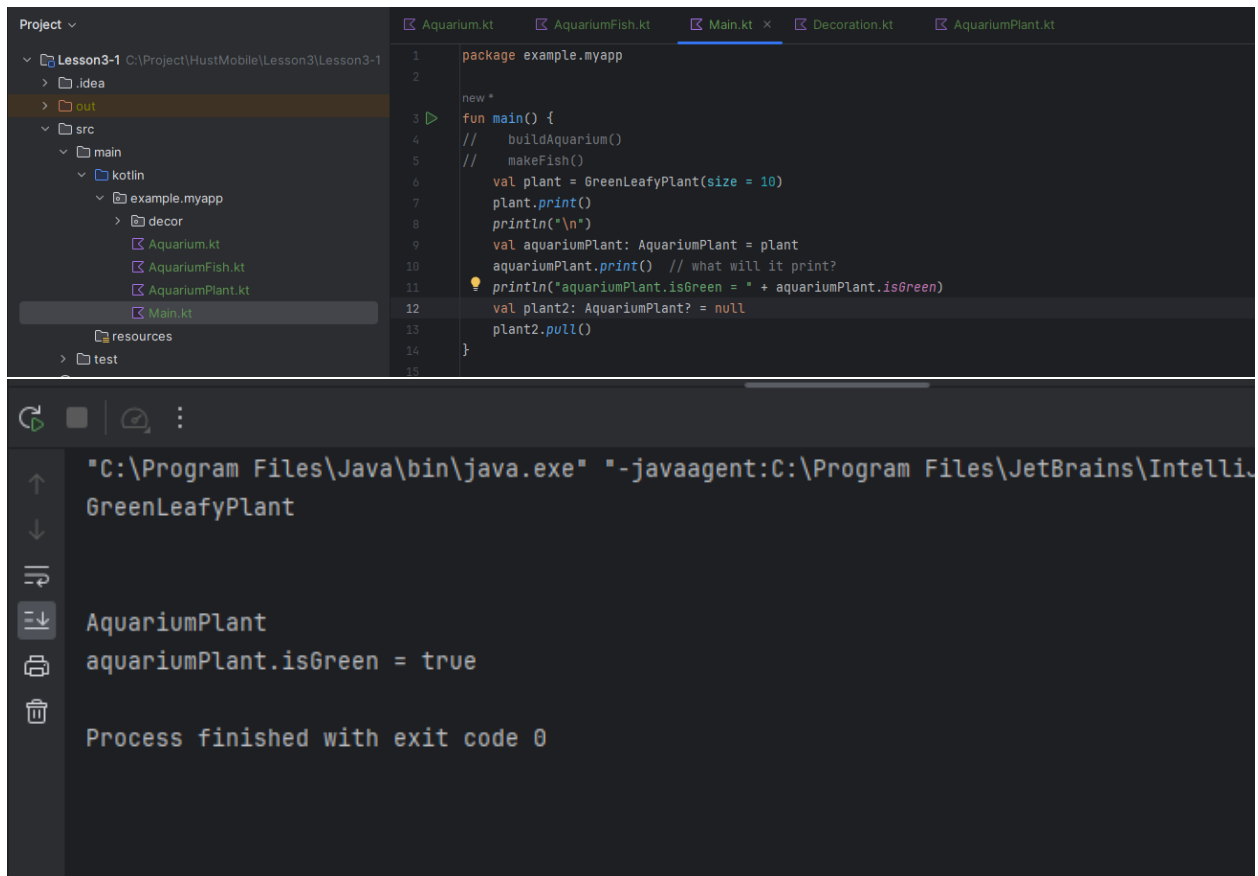
Process finished with exit code 0
```

2.5.4. Know about nullable receivers

Lớp bạn mở rộng được gọi là bộ thu, và có thể làm cho lớp đó có thể null. Nếu bạn làm vậy, biến this được sử dụng trong phần thân có thể là null, vì vậy hãy đảm bảo bạn kiểm tra điều đó. Bạn sẽ muốn lấy một bộ thu có thể null nếu bạn mong đợi rằng người gọi sẽ muốn gọi phương thức mở rộng của bạn trên các biến có thể null hoặc nếu bạn muốn cung cấp một hành vi mặc định khi hàm của bạn được áp dụng cho null.

Các bước thực hiện:

1. Vẫn đang hoạt động trong REPL, hãy định nghĩa phương thức pull() lấy một bộ thu có thể chấp nhận giá trị null. Điều này được chỉ ra bằng dấu chấm hỏi ? sau kiểu, trước dấu chấm. Bên trong phần thân, bạn có thể kiểm tra xem đây có phải là null không bằng cách sử dụng ?.apply.
2. Trong trường hợp này, không có đầu ra nào khi bạn chạy chương trình. Vì plant là null nên println() bên trong không được gọi.

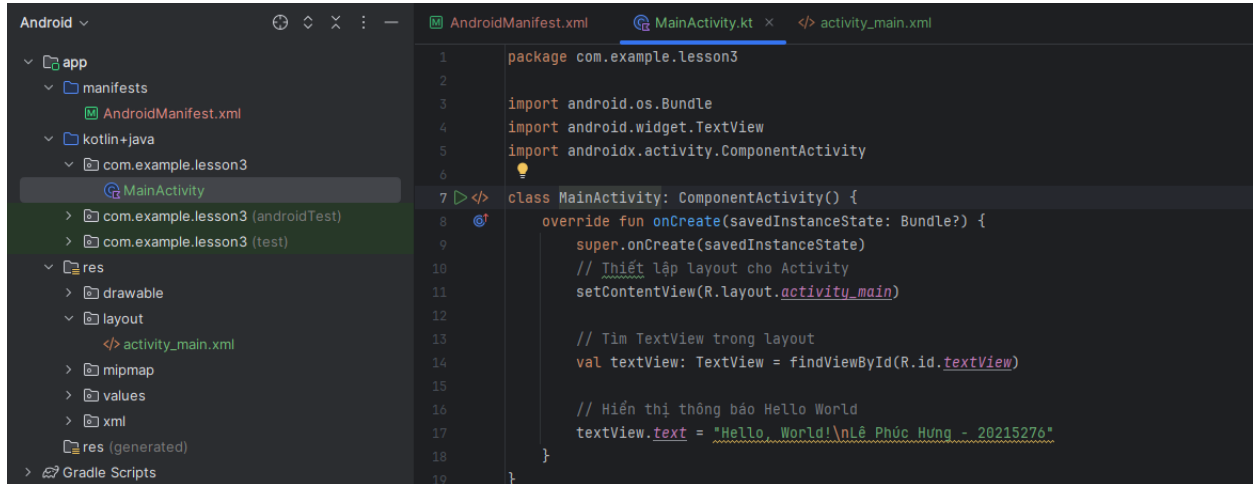


SS

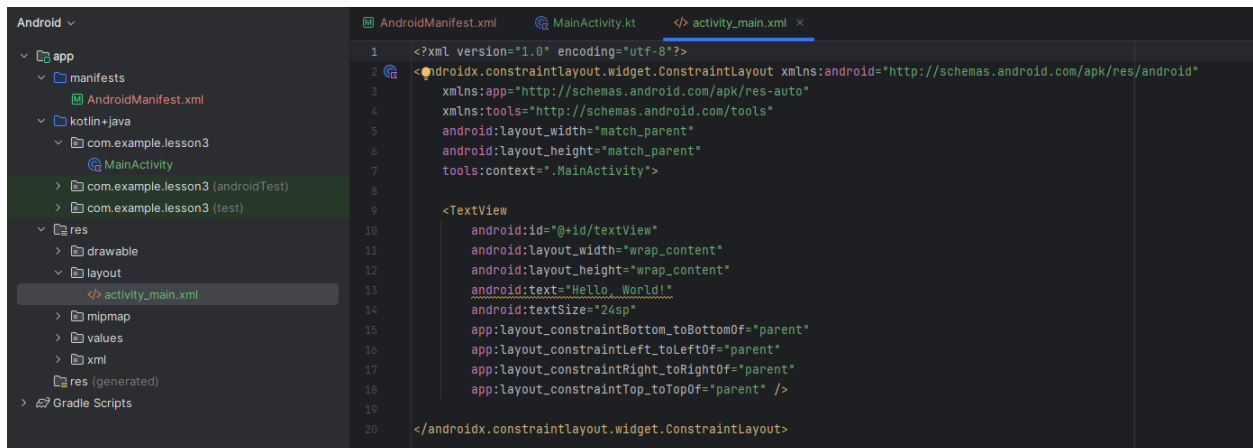
3. Create Android app Hello World

3.1. Steps to implement the program

Tạo dự án rỗng trong Android studio



Viết code cho File MainActivity như hình



Tạo file activity_main trong thư mục layout trong res như hình

3.2. The program Hello World

