NGÔN NGỮ LẬP TRÌNH

Trang Thanh Trí
tritrang@ctu.edu.vn

Tuần 3

Collections

- **Iterable**: Lớp cha. Bất kỳ class nào kế thừa từ interface này đại diện cho một chuỗi các phần tử mà chúng ta có thể duyệt qua
- **MutableIterable**: Iterable hỗ trợ việc xóa các phần tử trong khi danh duyệt
- **MutableCollection**: Collection hỗ trợ việc thêm hoặc xóa các phần tử: các hàm *add, remove, clear...*
- **List**: Collection có lẽ là được dùng nhiều nhất. Nó đại diện cho một tập các phần tử có thứ tự. Bởi vì có thứ tự, ta có thể truy cập các phần tử thông qua chỉ số
- **MutableList**: List hỗ trợ việc thêm hoặc xóa các phần tử

Collections

- **Set**: một tập các phần tử không có thứ tự và không hỗ trợ lưu các phần tử trùng
- **MutableSet**: Set hỗ trợ việc thêm và xóa các phần tử
- **Map**: một tập các cặp key - value với key trong map là duy nhất
- **MutableMap**: Map hỗ trợ việc thêm và xóa các phần tử

Collections

- var list: List<Int> = listOf(1, 2, 3, 4, 5) // [1,2,3,4,5]
- list.add(5) //compiler sẽ báo lỗi

- var list: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5) // [1,2,3,4,5]
- list.add(5) // OK
- list.clear() // OK

- val items = listOf(1, 2, 3, 4)
- items.first() == 1
- items.last() == 4
- items.filter { it % 2 == 0 }

Collections

- **Any**
  - var list = mutableListOf(3, 5, 7)
  - list.any { it == 3 } // true
- **All**
  - list.all { it % 3 == 0 } //true
- **Count**
  - list.count { it % 3 == 0 } // 1
- **fold**
  - val result = list.fold(0, { total: Int, i: Int -> total + i }) // 15

Collections

- **forEach**
  - list.forEach { println(it) }
- **Max**
  - print(list.max())
- **Min**
  - print(list.min())
- **Reduce**
  - list.reduce { total: Int, i: Int -> total + i }
- **sumBy**
  - list.sumBy { it % 3 }

Collections

- **Drop (xóa n phần tử đầu)**
  - var list = mutableListOf(3, 5, 6, 7, 9)
  - var result = list.drop(3) //[7, 9]
- **Filter**
  - var result = list.filter { it % 3 == 0 } // [3, 6, 9]
- **Map**
  - var result = list.map { it + 1 } // [4,6,7,8,10]
- **Contains**
  - var list = mutableListOf("a", "b", "c", null, "d")
  - var result = list.contains("a") // true

Collections

- **elementAt**
  - var result = list.elementAt(1) // “a”
- **indexOf**
  - var result = list.indexOf(“a”) // 0
- **Plus**
  - var list = listOf(1, 2, 4)
  - var listA = listOf(3, 4)
  - var result = list.plus(listA) // [1, 2, 4, 3, 4]
- **Zip**
  - var result = list.zip(listA) // [(1,3), (2,4)]
- **Sorted**
- **sortedDescending**

Collections

Iterate over a collection.

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
//sampleStart
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

Check if a collection contains an object using in operator.

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
//sampleStart
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
//sampleEnd
}
```

Collections

Using lambda expressions to filter and map collections:

```
fun main() {
//sampleStart
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
      .filter { it.startsWith("a") }
      .sortedBy { it }
      .map { it.uppercase() }
      .forEach { println(it) }
//sampleEnd
}
```

Idioms

- **Filter a list**
  - val positives = list.filter { it > 0 }
- **Check the presence of an element in a collection**
  - if ("john@example.com" in emailsList) { ... }
- **String interpolation**
  - println("Name $name")
- **Read-only map**
  - val map = mapOf("a" to 1, "b" to 2, "c" to 3)
  - println(map["key"])
- **Traverse a map or a list of pairs**
  - for ((k, v) in map) {
  - println("$k -> $v")
  - }

Idioms

- **Iterate over a range**
  - for (i in 1..100) { ... } // closed range: includes 100
  - for (i in 1 until 100) { ... } // half-open range: does not include 100
  - for (x in 2..10 step 2) { ... }
  - for (x in 10 downTo 1) { ... }
  - (1..10).forEach { ... }
- **Create a singleton**
  - object Resource { val name = "Name" }
- **try-catch expression**

```
try {
      count() }
catch (e: ArithmeticException) { throw IllegalStateException(e) }
```

Idioms

- **Swap two variables**
  - var a = 1
  - var b = 2
  - a = b.also { b = a }
- **Configure properties of an object (apply)**
- val myRectangle = Rectangle().apply {
  length = 4
  breadth = 5
  color = 0xFAFAFA }

Nullable values and null checks

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
//sampleEnd

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

Type checks and automatic casts

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object
is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

Lớp và tính kế thừa | Classes and inheritance

Classes

- Tạo lớp đơn giản sử dụng từ khóa class
  - Ví dụ: class Person { /*...*/ }

- Định nghĩa 1 lớp bao tên lớp, phần mở đầu và phần thân trong dấu { }, phần đầu và thân có thể có hoặc không.
  - Ví dụ: class Empty

Lớp và tính kế thừa | Classes and inheritance

Constructors

- Một lớp trong Kotlin có thể có 1 phương thức xây dựng và một hoặc nhiều phương thức xây dựng thứ 2.
- Phương thức xây dựng nguyên thủy là 1 phần của phần mở đầu lớp (class header).
  - Ví dụ: class Person constructor(firstName: String) { /*...*/ }
- Từ khóa constructor có thể bỏ qua nếu không có gì đặc biệt.
- Phương thức xây dựng nguyên thủy không chứa code, nếu muốn code khởi tạo thì viết trong block khởi tạo với từ khóa là init

Lớp và tính kế thừa | Classes and inheritance

Constructors

- Khi khởi tạo các khối lệnh khởi tạo sẽ được chạy theo đúng thứ tự trong class.

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)
    init {
            println("First initializer block that prints $name")
    }
    val secondProperty = "Second property:
${name.length}".also(::println)
    init {
            println("Second initializer block that prints
${name.length}")
    }
}
//sampleEnd
fun main() {
            InitOrderDemo("hello")
}
```

Lớp và tính kế thừa | Classes and inheritance

- Kotlin có một cú pháp ngắn gọn để khai báo các thuộc tính và khởi tạo chúng từ hàm tạo chính.

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

- Các khai báo như vậy cũng có thể bao gồm các giá trị mặc định của các thuộc tính lớp:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

Lớp và tính kế thừa | Classes and inheritance

- Cũng có thể sử dụng dấu phẩy cuối “trailing comma” khi khai báo thuộc tính
- Các thuộc tính có thể khai báo read-only (val) hoặc mutable (var)

```
class Person(
        val firstName: String,
        val lastName: String,
        var age: Int, // trailing comma ) { /*...*/ }
```

Lớp và tính kế thừa | Classes and inheritance

- Phương thức xây dựng thứ hai

```
class Person(val pets: MutableList<Pet> = mutableListOf<Pet>())
class Pet
{
        constructor(owner: Person)
        {
                owner.pets.add(this) // adds this pet to the list of its owner's pets
        }
}
```

Lớp và tính kế thừa | Classes and inheritance

- Phương thức xây dựng thứ hai
- Phải được gọi lại hàm khởi tạo đầu tiên

```
class Student (name: String){
        var subjects = mutableListOf<String>()
        constructor(initSubjects: List<String>):this(name){
                subjects.addAll(initSubjects)
        }
}
```

Lớp và tính kế thừa | Classes and inheritance

- Khởi tạo đối tượng (Kotlin không có từ khóa new)
  - val invoice = Invoice()
  - val customer = Customer("Joe Smith")
- Lớp có thể có:
  - Phương thức xây dựng và khối lệnh khởi tạo
  - Hàm
  - Thuộc tính
  - Lớp lồng nhau

Lớp và tính kế thừa | Classes and inheritance

- Tất cả lớp trong Kotlin đều có lớp chung là Any:
  - class Example // Implicitly inherits from Any
- Any có phương thức: equals(), hashCode(), and toString().
- Mặc định, lớp trong Kotlin là final – vì vậy không thể kế thừa. Để tạo lớp có thể kế thừa phải sử dụng từ khóa open:
  - open class Base // Class is open for inheritance
- To declare an explicit supertype, place the type after a colon in the class header:
  - open class Base(p: Int)
  - class Derived(p: Int) : Base(p)

Lớp và tính kế thừa | Classes and inheritance

- Overriding methods (Ghi đè phương thức)
- Kotlin requires explicit modifiers for overridable members and overrides
- Cấm ghi đè thì sử dụng từ khóa final

```
open class Shape {
        open fun draw() { /*...*/ }
        fun fill() { /*...*/ }
}
class Circle() : Shape() {
        override fun draw() { /*...*/ }
}
open class Rectangle() : Shape() {
        final override fun draw() { /*...*/ }
}
```

Lớp và tính kế thừa | Classes and inheritance

- Overriding properties (Ghi đè thuộc tính)
- Giống như ghi đè phương thức
- Lưu ý: có thể ghi đè val bằng var (ngược lại thì không được)

```
open class Shape {
        open val vertexCount: Int = 0
}
class Rectangle(override val vertexCount: Int = 4) : Shape
class Polygon : Shape {
        override var vertexCount: Int = 0
}
```

Lớp và tính kế thừa | Classes and inheritance

- Derived class initialization order

```
//sampleStart
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it")
}) {

    init { println("Initializing a derived class") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derived class: $it")
}
}
//sampleEnd

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

Lớp và tính kế thừa | Classes and inheritance

- Calling the superclass implementation

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

Lớp và tính kế thừa | Classes and inheritance

- Inside an inner class, accessing the superclass of the outer class is done using the super keyword qualified with the outer class name: super@Outer:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

//sampleStart
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of draw()
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}") //
Uses Rectangle's implementation of borderColor's get()
        }
    }
}
//sampleEnd

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

Lớp và tính kế thừa | Classes and inheritance

- Overriding rules

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

Lớp và tính kế thừa | Classes and inheritance

- Declaring properties

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
```

Lớp và tính kế thừa | Classes and inheritance

- Getters and setters: mặc định các thuộc tính đều có sẵn getter/setter

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be inferred from the getter's return type
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

Lớp và tính kế thừa | Classes and inheritance

- Getters and setters: hoặc cài đặt lại như sau

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other properties
    }
```

```
var counter = 0 // the initializer assigns the backing field directly
    set(value) {
        if (value >= 0)
            field = value
            // counter = value // ERROR StackOverflow: Using actual name 'counter' would make setter
recursive
    }
```

Enum classes

- Enum là tập hợp các giá trị hằng:
  - **enum class** CardType { SILVER, GOLD, PLATINUM }
- Enum có thể có phương thức xây dựng:

```
enum class CardType(val color: String)
    {
        SILVER("gray"),
        GOLD("yellow"),
        PLATINUM("black")
    }
```

Enum classes

- Có thể xem là lớp ẩn danh

```
enum class CardType {
    SILVER {
        override fun calculateCashbackPercent() = 0.25f
    },
    GOLD {
        override fun calculateCashbackPercent() = 0.5f
    },
    PLATINUM {
        override fun calculateCashbackPercent() = 0.75f
    };

    abstract fun calculateCashbackPercent(): Float
}
```

Enum classes

- Những cách thường dùng:
- **val** cardType = CardType.valueOf(name.toUpperCase())
- Lặp qua các giá trị trong enum:

```
for (cardType in CardType.values()) {
        println(cardType.color)
}
```

- Phương thức tĩnh

```
companion object {
    fun getCardTypeByName(name: String) = valueOf(name.toUpperCase())
    }
val cardType = CardType.getCardTypeByName("SILVER")
```

Data classes

- Được sử dụng để lưu trữ dữ liệu
  - data class User(val name: String, val age: Int)
- Những hàm cơ bản:
  - equals()/hashCode() pair
  - toString() of the form "User(name=John, age=42)“
  - componentN() functions corresponding to the properties in their order of declaration.
  - copy() function
- Phương thức xây dựng nguyên thủy phải có ít nhất 1 tham số
- Tất cả tham số phải chỉ định val hoặc var
- Data classes không thể là abstract, open, sealed, or inner.

Data classes

- Cách sử dụng hàm copy(): dùng để sao chép đối tượng (cho phép thay đổi 1 số thuộc tính, phần còn lại giữ nguyên)

- data class Person(val name: String, val age:Int=0)
- fun main() {
- var p1 = Person("abc")
- print(p1.toString())
- var p2 = p1.copy(age=1)
- print(p2.toString())
- }

Bài tập

- 1. Xây dựng lớp Animal (Thú), có các thuộc tính chiều cao (height), cân nặng (weight), sử dụng getter/setter để ngăn chặn không cho nhập giá trị âm vào 2 thuộc tính trên (nếu âm thì mặc định là 0), xây dựng phương thức di chuyển (move), phương thức này sẽ println(“moving”)
- 2. Xây dựng lớp Chó(Dog), Chim(Bird) kế thừa lớp Animal, cài đè lại phương thức di chuyển, nếu là chó thì di chuyển sẽ println(“running”), nếu là chim sẽ là println(“flying”). Cả 2 lớp Chó và Chim sẽ có thêm phương thức xây dựng thứ 2, cho phép nhập vào height, weight, name cùng lúc. Sử dụng getter cho name, khi in ra name thì viết **IN HOA** hết.
- 3. Xây dựng lớp Con người (Person), có thuộc tính là danh sách các thú cưng, họ và tên, sử dụng getter để in ra fullname. Cài đặt phương thức thêm thú cưng và phương thức in tất cả danh sách thú cưng.