Kotlin Fundamental

bangkit

Ground Rules

Observe the following rules to ensure a supportive, inclusive, and engaging classes

Give full attention in class

Mute your microphone when you're not talking

Keep your camera on

Turn on the CC Feature on Meet

Use raise hand or chat to ask questions

Make this room a safe place to learn and share

bangkit

87%

of top 1000 apps use Kotlin

bangkit

Basic
Programming

- Variables & Data Types
- Control Flow
- Functional Programming
- Object-Oriented Programming
- Data Classes & Collection

bangkit

Variables & Data Types

Variables & Data Types

```
// val vs var
val name: String = "Dico" // cannot be reassigned
var age: Int = 7 // can be reassigned
age++ // increment operator

// type inference
val oneMillion = 1_000_000 // Int → underscore for readable
val weight = 60.2 // Double
val isLogin = true // Boolean

// built-in function
println(name.uppercase()) // convert to upper case
println(oneMillion.toString()) // parse to String
println(weight.plus(6)) // operator method

// string template
println("Number of char in $name \nis ${name.length}") // escaped string
```

Important Notes for Variables & Data Types

//val vs var
var : value can be reassigned (mutable)
val : value can NOT be reassigned (read-only)
Give the imagery that the different of them like an opened box and isolated box.
Recommend to use val where possible, use var only when needed.

//type inference
Type inference means some type information in the code may be omitted, to be inferred by the compiler.

//build in function
With build in function, we do not need to build common functionalities
This is not an exhaustive list, so you can look up more built in functions in this link
https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/

//string template
Instead of using + to concatenate string, Tell that using string template make code easy to read.

Nullable Types

Example:

```
val text: String? = null
```

Wrong example:

```
val text: String = null
// compile time error
```

We are using nullable types when we need to represent an undefined value of an underlying type

Safe call operator (?.)

```
val text: String? = null
val textLength = text?.length
// textLength = null (not error)
```

Elvis operator (?:)

```
val text: String? = null
val textLength= text?.length ?: 0
// textLength = 0
```

Control Flow

If Statement and When Expression Comparison

If Statement

```
val now = 7 // 07.00
if (now > 7 && now <= 17) {
   print("Office is open")
} else if (now == 7){
   print("Wait a minute")
} else {
   print("Office is closed")
}
```

When Expression

```
val now = 7
val officeStatus = when(now) {
   12, 15 -> "Rest time"
   in 8..17 ->"Office is open"
   7 -> "Wait a minute"
   is Int -> "Office is closed"
   else -> "Invalid office time"
}
print(officeStatus)
```

While and For Comparison

While

```
var x = 0
while (x <= 5){
   println("Hello World")
   x++
}
```

Do..While

```
var y = 0
do {
   println("Hello Kotlin")
   y++
} while (y <= 5)
```

For

```
// iterate from range
for (i in 0..5){} // 0 to 5
for (i in 0 until 5){} // 0 to 4
for (i in 0..5 step 2){} // 1,3,5
for (i in 5 downTo 1){} // 5 to 1
repeat(5) {} // loop 5x

// iterate from array
val array = arrayOf("A", "B", "C")
for (element in array) {
   println(element)
}
array.forEach { element ->
   println(element)
}
```

Important Notes for While and For Comparison

Demo: https://pl.kotl.in/mzzR4zGHk

Object Oriented Programming (OOP)

4 Pillars of OOP

OOP Concept

```
class Cat(
    val name: String,
    val height: Double,
    val length: Double,
    val weight: Double
){
   fun playing() {
       println("$name is playing")
   }
   fun eat() {
       println("$name is eating")
   }
}
```

```
fun main() {
    val garfield = Cat(
       "Garfield",
       22.0,
       39.0,
       2.0
    )
    garfield.eat()
    garfield.playing()
}
// Output:
Garfield is eating
Garfield is playing
```

Constructor

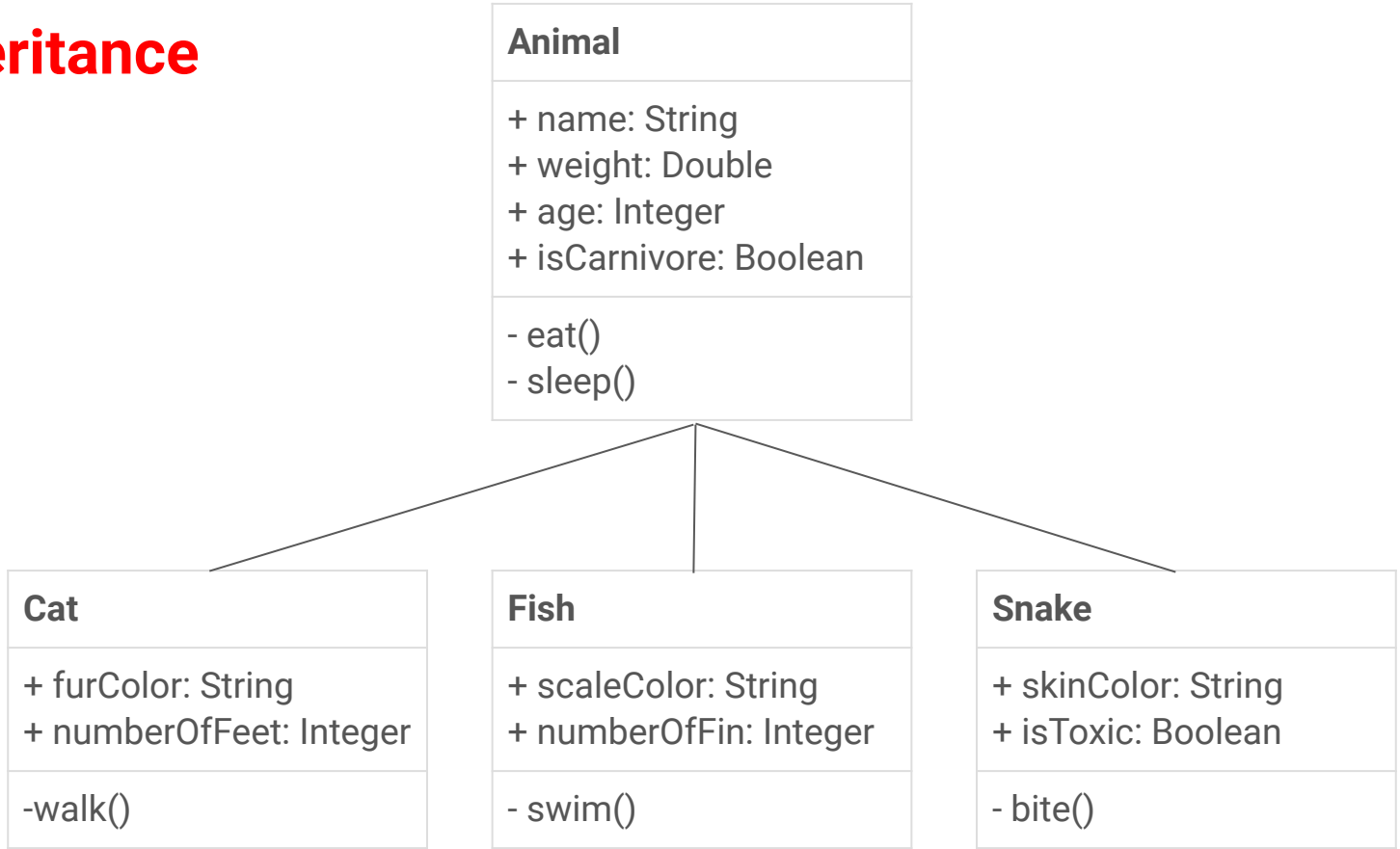
```
class Cat(name: String, weight: Double){
    var name: String
    var weight: Double

    init {
        this.name = name
        this.weight = weight
    }
    constructor(name: String) : this (name, 1.0)
}
fun main() {
    val garfield = Cat("Garfield", 2.0)
    val kitty = Cat("Kitty")
}
```

Primary Constructor
(pass data to class without setter)

Init Block
(first function that executed when object created)

Secondary Constructor
(another way to create object)

Inheritance



Inheritance

```
open class Animal(){

  open fun eat(){
    println("Animal eats")
  }

  open fun sleep(){
    println("Animal sleeps")
  }
}
```

Subclass

```
class Cat : Animal() {

  fun walk () {
    println("Cat can walk")
  }

  override fun eat(){
    println("Cat eats fish")
  }
}
```

Abstraction : Abstract Class and Interface

```
abstract class Animal(){
   ...
}

class Cat : Animal() {
   ...
}

fun main(){
  val garfield = Animal()
  // Cannot create an instance of
an abstract class
  val garfield = Cat()
  // OK
}
```

```
interface IWalk {
   fun walk()
}

class Cat : Animal(), IWalk {
   // this function must exist
   override fun walk () {
       println("Cat can walk")
   }
   ...
}
```

Important Notes for Abstraction

Demo: https://pl.kotl.in/LoLedaz6g

Encapsulation : Visibility Modifier

By specifying these permissions, we can restrict data access to a class.

Modifier	Description
public	Visible everywhere
private	Visible inside the same class only
internal	Visible inside the same module
protected	Visible inside the same class and its subclasses

Polymorphism : Overloading

Makes the same function same with different parameters.

```
class Cat {
  fun eat(food: String){
    println("Cat eat $food")
  }

  fun eat(food: String, quantity: Int){
    println("Cat eat $food as much as
      $quantity grams!")
  }
}
```

```
fun main(){
   val garfield = Cat()
   garfield.eat("fish")
   garfield.eat("fish", 20)
}

// Output:
// Cat eat fish
// Cat eat fish as much as 20 grams!
```

Property Delegation

```
class DelegateName {
    private var value: String = "Default"

    operator fun getValue(classRef: Any?, property: KProperty<*>) : String {
        println("Do something when get value")
        return value
    }

    operator fun setValue(classRef: Any?, property: KProperty<*>, newValue: String){
        println("Do something when set value")
        value = newValue
    }
}
class Animal {
    var name: String by DelegateName()
}

class Person {
   val name: String by lazy { "Fikri" }
}
```

Classes & Collections

Data Class and Destructuring Declarations

Provide and retrieve properties in class. Example:

```
data class DataUser(var name : String, var age : Int)
// automatically generated equals(), hashCode(), toString(), and copy()
fun main(){
    val dataUser = DataUser("arif", 17)
     println(dataUser) //User(name=arif, age=17), not User@4d7e1886
    // set data
    dataUser.name = "faizin"
    dataUser.age = 24
    // get data
    val userName = dataUser.name
    val userAge = dataUser.age
    // destructuring declarations
    val (name,age) = dataUser
    println("My name is $name, I am $age years old")
}
```

Enum and Sealed Class

Enum Class

```
enum class Status {
    SUCCESS,
    ERROR,
    LOADING
}
```

Sealed Class

```
sealed class Result {
    data class Success(val data: T) : Result()
    data class Error(val error: String) : Result()
    object Loading : Result()
}
```

Important Notes or Enum & Sealed Classes

Demo: https://pl.kotl.in/PyoqC1WwJ

Collection and Operator

```
// List: to store independent values with index (may be the same)
val integerList = listOf(4, 2, 1, 5, 1, 2)// [4, 2, 1, 5, 1, 2]
integerList[3] // 5

// Set: to store unique values
val integerSet = setOf(4, 2, 1, 5, 1, 2) // [4, 2, 1, 5]
integerSet.filter { it % 2 == 0 } // 2, 4
integerSet.map { it * 5 } //10, 20
integerSet.sortedDescending() // 5, 4, 2, 1

// Map: to save in key to value format, unique keys cannot be the same.
val capital = mapOf(
    "Jakarta" to "Indonesia",
    "London" to "England",
    "New Delhi" to "India"
)
println(capital["Jakarta"]) //Indonesia
```

mutableList and list comparison

mutableList can be changed (**mutable**)

List can NOT be changed (**immutable**)

```
val anyList = mutableListOf('a', "Kotlin", 3, true, User())

anyList.add('d') // add item in at last index
anyList.add(1, "love") // add item at 1st index
anyList[3] = false // change value item at 3rd index
anyList.removeAt(1) // remove item at 1st index
```

Note:
Besides **mutableList**, there is also **mutableSet** and **mutableMap**.

Functional Programming (FP)

Kotlin Function

Visibility modifier
(default is public)

Function Name
(lowerCamelCase)

Parameters
(separated by comma)

Return Type
(default is Unit)

Function body
(inside curly braces)

```
private fun setUser(name: String, age: Int): String {
    return "Your name is $name, and you $age years old"
}
```

Named Argument
(can be not sequential)

```
val word = getWord(middle=" is ", last="Awesome")

fun getWord(first: String = "Java", middle: String, last: String) =
"$first $middle $last"
```

Default Argument

Single Expression

bangkit

Extensions

Add new functions/properties to existing basic data types without direct inherited.

Extension **Functions**

```
fun String.isMoreThan6Char():
Boolean {
    return this.length > 6
}
```

```
fun main() {
    val isValid =
    "Password".isMoreThan6Char())
}
```

Extension **Properties**

```
val Int.formatRupiah: String
    get() = "Rp $this"
```

```
fun main() {
    println(10000.formatRupiah)
}
```

Lambda

Example of regular function:

```
fun getMessage(name: String): String {
   return "Hello $name"
}
```

Example of using Lambda:

```
val message :(String) -> String = { name: String ->
   "Hello $name"
}
```

We can call the lambda just as we do in function:

```
val newMessage = message("Dicoding") //Output: Hello Dicoding
```

Higher-order Function

Example of accepting another **function as a parameter**:

```
fun main(){
   printMessage("Dicoding", ::welcomeMessage)
}


fun welcomeMessage(name: String) = "Hello $name"


fun printMessage(name: String, message: (String) -> String) {
   print(message(name))
}
```

Function reference

Lambda with Receiver

This concept is used as the basis of Domain Specific Languages (DSL).

Without DSL

```
fun main() {
  val message = buildString()
  println(message)
}

fun buildString(): String {
  val stringBuilder = StringBuilder()
  stringBuilder.append("Hello ")
  stringBuilder.append("from ")
  stringBuilder.append("lambda")
  return stringBuilder.toString()
}
```

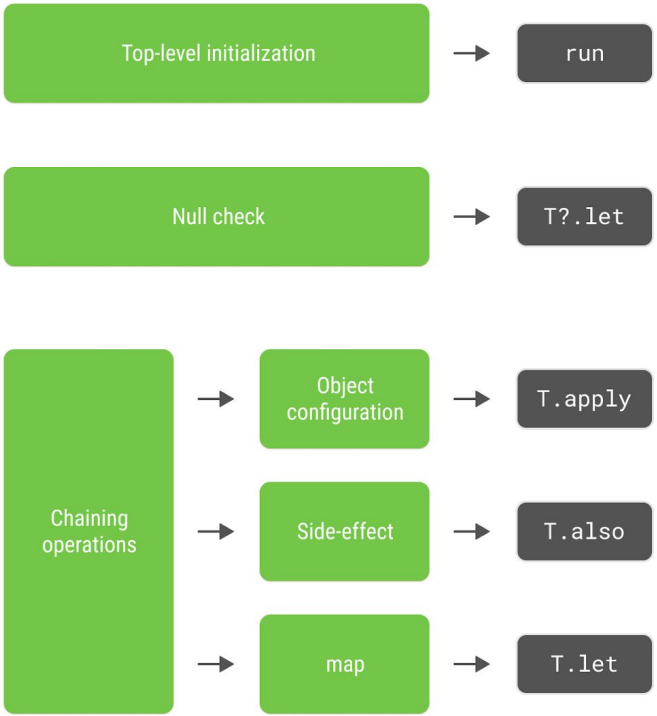
With DSL

```
fun main() {
  val message = buildString {
      append("Hello ")
      append("from ")
      append("lambda")
   }
  println(message)
}

fun buildString(action:StringBuilder.()
->Unit): String {
  val stringBuilder = StringBuilder()
  stringBuilder.action()
  return stringBuilder.toString()
}
```

Important Notes for Lambda and Receiver

Demo: https://pl.kotl.in/DHv3uqJzp

Scope Function




```
val myObj = run {
    val generator = DateStringGenerator(2008, 9, 23)
    generator.locale = "US"
    generator.localizedDate
}
```

```
class MyClass {

    fun printExceptionMessage(exception: Exception?) {
        exception?.let {
            println(exception.message)
        }
    }
```

```
    val lastWeeksDate: String = Calendar.getInstance().apply {
        add(Calendar.DAY_OF_YEAR, -7)
    }
```

```
        .also {
            println("Created calendar of type: ${it.calendarType}")
        }
```

```
        .let { calendar ->
            val format = SimpleDateFormat("dd/M/yyyy hh:mm:ss", Locale.US)
            format.format(calendar.time)
        }
}
```


bangkit

Important Notes Scope Function

Source:
https://raw.githubusercontent.com/JoseAlcerreca/kotlin-std-fun/master/Kotlin%20Standard%20Functions%20Table.png

Demo:
https://pl.kotl.in/DyvbMY8Er

Sharing

Discussion

Quiz

Thank You