3. Kotlin

Présentation du langage de programmation *Kotlin*.

Android Studio Hedgehog Essentials, Kotlin Edition : chapitres 11 à 16

3.1. Introduction

1. *Kotlin* est un nouveau langage proposé par *Google* pour éventuellement remplacer *Java* dans le développement d'application *Android*.

   a. C'est du *Java* simplifié

   b. Une même application *Android* peut combiner aisément du code *Kotlin* et du code *Java*

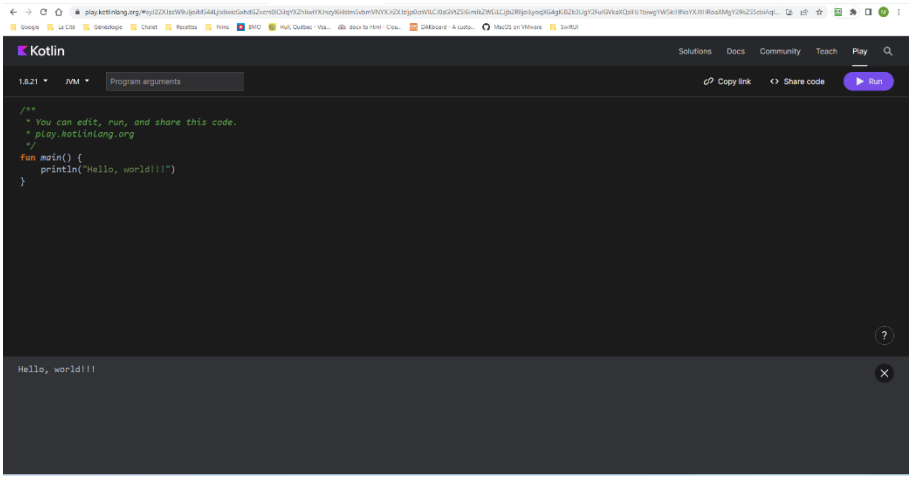
   c. Lors d'un copier-coller de code *Java* dans un projet *Kotlin*, *Android Studio* convertit automatiquement le code *Java* copié en *Kotlin*

2. Il est toujours possible d'exploiter *Java* à cause du code fossile (*legacy code*).

3. Il n'y a pas encore de « standard » *Kotlin*, donc la syntaxe évolue continuellement.

4. On va utiliser un interpréteur Web accessible à https://play.kotlinlang.org

   a. Ça nous évite d'avoir à concevoir une activité dans un projet *Android Studio*



3.2. Types de données, variables et constantes

5. Accédez à l'interpréteur Web *Kolín*

6. Déclarez des variables :

```
fun main() {
    var note = 75
    var message = "Note de Gustave: "
    println(message + note)
}
```

```
Note de Gustave: 75
```

   a. *Kotlin* infère le type de variable selon la valeur lui étant assignée.

   b. Attribuez une valeur de type différente à une même variable, puis afficher le message d'erreur produit :

```
fun main() {
    var note = 75
    var message = "Note de Gustave: "
    println(message + note)

    note = "B-"
}
```

```
Type mismatch: inferred type is String but
Int was expected
```

7. Déclarez une variable explicitement d'un type :

```
fun main() {
    var note = 75
    var message = "Note de Gustave: "
    println(message + note)

    var entier: Int
    entier = 20
}
```

```
Note de Gustave: 75
```

8. Les types fondamentaux sont en fait des classes :

```
fun main() {
    var note = 75
    var message = "Note de Gustave: "
    println(message + note)

    var entier: Int
    entier = 20

    println(Int.MAX_VALUE)
}
```

```
Note de Gustave: 75
2147483647
```

9. Déclarez des variables des autres types communs :

```
fun main() {
    ...

    println(Int.MAX_VALUE)

    var flottant: Double
    var booleen: Boolean
    var chaine: String
    chaine = "Bonjour"
}
```

```
Note de Gustave: 75
2147483647
```

10. *Kotlin* gère les accents, même dans les identificateurs :

```
fun main() {
    ...

    println(Int.MAX_VALUE)

    var flottant: Double
    var booleen: Boolean
    var chaine: String
    chaine = "Bonjour"

    var booléen: Boolean
    booléen = true
}
```

```
Note de Gustave: 75
2147483647
```

11. Des expressions peuvent être imbriquées dans une chaîne via **`${}`** :

```
fun main() {
    ...

    var booléen: Boolean
    booléen = true

    message = "La somme de 2+3 est ${2+3}."
    println(message)
}
```

```
Note de Gustave: 75
2147483647
La somme de 2+3 est 5.
```

12. Les constantes sont déclarées à l'aide du mot-clé **`val`**, et elles peuvent être assignées ultérieurement (une seule fois cependant) :

```
fun main() {
    ...

    message = "La somme de 2+3 est ${2+3}."
    println(message)

    val i = entier + 1
    val j: Int = 12
    val k: Int
    k = i + j
    println(k)
}
```

```
Note de Gustave: 75
2147483647
La somme de 2+3 est 5.
33
```

3.3. Variables optionnelles

13. Les variables *Kotlin* ne peuvent se voir attribuées `null` comme valeur

```
fun main() {
    var nomUsager: String = null
    println(nomUsager)
}
```

```
Null can not be a value of a non-null type
String
```

14. Expliquez qu'une variable optionnelle (*nullable* en anglais) est une variable pouvant se voir assigner la valeur `null`

```
fun main() {
    var nomUsager: String? = null
    println(nomUsager)
}
```

```
null
```

15. La manipulation d'une variable optionnelle exige une vérification qu'elle est non nulle

```
fun main() {
    var nomUsager: String? = "Gustace"
    println(nomUsager)

    println(nomUsager.toUpperCase())
}
```

```
Only safe (?.) or non-null asserted (!!.)
calls are allowed on a nullable receiver of
type String?
```

a. Alors que ceci fonctionne :

```
fun main() {
    var nomUsager: String? = "Gustace"
    println(nomUsager)

    if (nomUsager != null) {
        println(nomUsager.toUpperCase())
    }
}
```

```
Gustave
GUSTAVE
```

b. L'opérateur d'invocation sécurisé (`?.`) permet d'écourter la syntaxe précédente en interrompant l'évaluation si la variable optionnelle est nulle

```
fun main() {
    var nomUsager: String? = "Gustace"
    println(nomUsager)

    println(nomUsager?.toUpperCase())
}
```

```
Gustave
GUSTAVE
```

16. Si on est certain qu'une variable optionnelle est non nulle, on peut « forcer » l'invocation via l'opérateur d'assertion non-nulle (`!!.`)

```
fun main() {
    var nomUsager: String? = "Gustace"
    println(nomUsager)

    println(nomUsager!!.toUpperCase())
}
```

```
Gustave
GUSTAVE
```

a. Démontrez cependant le crash de l'application si l'assertion est infondée :

```
fun main() {
    var nomUsager: String? = null
    println(nomUsager)

    println(nomUsager!!.toUpperCase())
}
```

```
null
Exception in thread "main" java.lang.NullPointerException
at FileKt.main (File.kt:9)
at FileKt.main (File.kt:-1)
at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0
(:-2)
```

17. Une alternative à la vérification qu'une variable optionnelle est non nulle (via un `if` ou `?.`) est la fonction `let`

a. Effectuez une multiplication via une fonction membre de la classe `Int` :

```
fun main() {
    val premierNombre = 10
    val deuxièmeNombre = 20

    val résultat = premierNombre.times(deuxièmeNombre)
    println(résultat)
}
```

```
200
```

b. En revanche, il y a erreur de compilation si la seconde constante est optionnelle :

```
fun main() {
    val premierNombre = 10
    val deuxièmeNombre: Int? = 20

    val résultat = premierNombre.times(deuxièmeNombre)
    println(résultat)
}
```

```
Type mismatch: inferred type is Int? but Int was expected
```

c. La fonction let permet de transformer la variable optionnelle en équivalente non-optionnelle nommée `it` :

```
fun main() {
    val premierNombre = 10
    val deuxièmeNombre: Int? = 20

    deuxièmeNombre?.let {
        val résultat = premierNombre.times(it)
        println(résultat)
    }
}
```

```
200
```

18. Enfin, présentez l'opérateur Elvis (`?:`) permettant de substituer une valeur nulle par une valeur par défaut en une seule ligne de code :

```
fun main() {
    val premierNombre = 10
    val deuxièmeNombre: Int? = 20

    val résultat = premierNombre.times(deuxièmeNombre ?: 1)
    println(résultat)
}
```

```
200
```

a. Montrez que l'expression ci-dessus est équivalente au code suivant :

```
fun main() {
    val premierNombre = 10
    val deuxièmeNombre: Int? = 20

    val résultat: Int
    if (deuxièmeNombre != null) {
        résultat = premierNombre.times(deuxièmeNombre)
    }
    else {
        résultat = premierNombre.times(1)
    }
    println(résultat)
}
```

```
200
```

3.4. Opérateurs

19. Soulignez que la plupart des opérateurs standards de *Java* et *C#* sont les mêmes en *Kotlin*, tels que

a. Opérateurs arithmétiques : +, -, *, / et %

b. Opérateurs d'affectation : =, +=, -=, *=, …

c. Opérateurs relationnels : ==, !=, <, <=, > et >=

d. Opérateurs logiques : &&, || et !

e. Opérateurs d'incrémentation (++) et de décrémentation (--) en préfixe et postfixe

3.5. Structures conditionnelles

20. On a déjà vu le `if` :

```
fun main() {
    val x = 10
    val y = 20

    if (x > 0 && y > 0)
        println("Ok")
    else
        println("Bad")
}
```

```
Ok
```

a. Ne pas oublier les {} s'il y a plus d'une instruction dans la partie vraie ou fausse

21. Soulignez que le `switch` du *Java* ou *C#* est remplacé par le `when` en *Kotlin* :

```
fun main() {
    ...

    when (x) {
        10 -> println ("x = 10")
        8 -> println("x = 8")
        2,4,6 -> println("x = 2, 4 ou 6")
        else -> println("x n'est pas pair entre 2 et 10")
    }
}
```

```
Ok
x = 10
```

3.6. Structures répétitives

22. Soulignez que les boucles sont semblables à celles en *Java* et en C#

23. Présentez premièrement différentes saveurs du `for` :

```
fun main() {
    print("j =")
    for (j in 0..9) {
        print(" " + j)
    }

    val s = "bonjour"
    print("\nCaractères dans $s:")
    for (c in s) {
        print(" $c")
    }

    print("\nk =")
    for (k in 20 downTo 10 step 2)
        print(" " + k)
}
```

```
j = 0 1 2 3 4 5 6 7 8 9
Caractères dans bonjour: b o n j o u r
k = 20 18 16 14 12 10
```

24. Présenter les boucles `while` et do…`while` :

```
fun main() {
    var i = 0
    while (i < 10) {
        print("$i ")
        i += 1
    }

    do {
        print("$i ")
        i -= 2
    } while (i > 0)
}
```

```
0 1 2 3 4 5 6 7 8 9 10 8 6 4 2
```

3.7. Fonctions et lambdas

25. Présentez brièvement ces quelques exemples simples de fonctions

a. Première version :

```
fun main() {
    bonjour()
}

fun bonjour() {
    println("Bonjour!")
}
```

```
Bonjour!
```

b. Deuxième version :

```
fun main() {
    bonjour()
    bonjour("Gustave")
}

fun bonjour() {
    bonjour(null)
}

fun bonjour(nom: String?) {
    print("Bonjour")
    if (nom != null)
         print(" " + nom)
    println("!")
}
```

```
Bonjour!
Bonjour Gustave!
```

c. Troisième version :

```
fun main() {
    bonjour()
    bonjour("Gustave")
}

fun bonjour() {
    bonjour(null)
}

fun bonjour(nom: String?) {
    println(messageBonjour(nom))
}
```

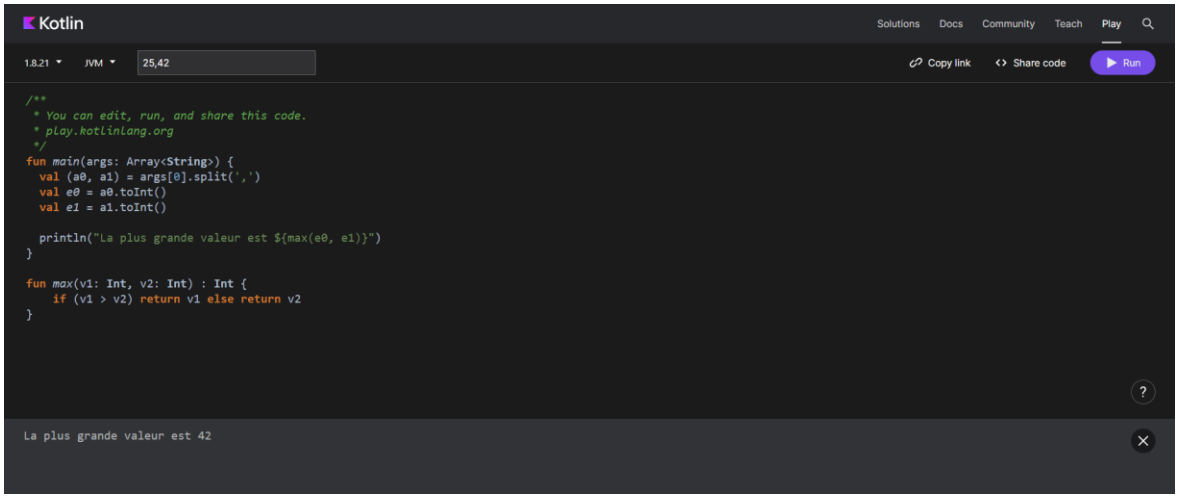
```
Bonjour!
Bonjour Gustave!
```

```
fun messageBonjour(nom: String?) : String {
    var msg = "Bonjour"

    if (nom != null)
         msg += " " + nom

    msg += "!"

    return msg
}
```

26. *Kotlin* supporte aussi les expressions lambdas

a. Expliquez (sans trop de détails) que ce sont des blocs de code (généralement très courts) insérés dans le code afin d'éviter la définition de fonctions invoquées à un seul endroit dans le code

b. Soulignez que https://play.kotlinlang.org/ ne pouvant pas gérer les lectures via la console, les inputs doivent être fournies via ligne de commande



c. Remplacez la définition de `max()` par une expression lambda

```
fun main(args: Array<String>) {
  val (a0, a1) = args[0].split(',')
  val e0 = a0.toInt()
  val e1 = a1.toInt()

  val max = { v1: Int, v2: Int -> if (v1 > v2) v1 else v2 }
  println("La plus grande valeur est ${max(e0, e1)}")
}
```

```
La plus grande valeur est 42
```

Évaluation formative 03 Indiquez aux étudiants qu'ils peuvent récupérer l'énoncé de l'exercice (en format PDF) sur le portail éducatif du collège. Ils devraient pouvoir solutionner l'exercice sans aide en se référant aux chapitres 11 à 15 du livre de référence

3.8. Définition de classes

27. Expliquez qu'on va définir une classe simple représentant un compte bancaire :

```
fun main() {
    var compte = CompteBancaire()
    compte.afficherSolde()
}

class CompteBancaire {
    var solde: Double = 0.0
    var numéro: Int = 0

    fun afficherSolde() {
        val montant = String.format("%.2f$", solde)
        println("Solde du compte #$numéro: $montant")
    }
}
```

```
Solde du compte #0: 0.00$
```

28. Pour permettre de fournir des valeurs d'attributs lors de l'instanciation, la classe doit avoir un constructeur :

```
fun main() {
    var compte = CompteBancaire(123456, 2045.37)
    compte.afficherSolde()
}

class CompteBancaire {
    var solde: Double = 0.0
    var numéro: Int = 0

    constructor(numéro: Int, solde: Double) {
        this.numéro = numéro
        this.solde = solde
    }

    ...
}
```

```
Solde du compte #123456: 2045.37$
```

a. Soulignez qu'il est possible de déclarer les attributs ET le constructeur de façon abrégé :

```
fun main() {
    var compte = CompteBancaire(123456, 2045.37)
    compte.afficherSolde()
}

class CompteBancaire(var numéro: Int, var solde: Double) {
    fun afficherSolde() {
        val montant = String.format("%.2f$", solde)
        println("Solde du compte #$numéro: $montant")
    }
}
```

```
Solde du compte #123456: 2045.37$
```

b. Soulignez qu'une classe peut disposer de plus d'un constructeur, en autant qu'ils n'aient pas de signature commune :

```
fun main() {
    var compte = CompteBancaire(123456, 2045.37, "Tintin")
    compte.afficherSolde()
}

class CompteBancaire(var numéro: Int, var solde: Double) {
    var détenteur: String? = null
```

```
Solde du compte #123456 détenu par TinTin:
2045.37$
```

```
    constructor(numéro: Int, solde: Double, détenteur: String)
                    : this(numéro, solde) {
        this.détenteur = détenteur
    }

    fun afficherSolde() {
        val montant = String.format("%.2f$", solde)
        val détenteur = if (this.détenteur != null)
                            " détenu par ${détenteur}" else ""
        println("Solde du compte #$numéro$détenteur: $montant")
    }
}
```

29. En *Kotlin*, un constructeur ne peut que contenir des affectations initialisant les attributs membres. Pour exécuter du code lors de l'instanciation, il faut un initialiseur

```
class CompteBancaire(var numéro: Int, var solde: Double) {
    init {
        // code d'initialisation
    }
}
```

    a. L'initialiseur d'une classe est invoqué APRÈS ses constructeurs

30. Par défaut, les membres d'une classe sont d'accès `public`. Pour tout autre accès (`protected`, `private`, `internal`), il faut explicitement l'identifier pour chaque membre (comme en *C#*)

31. *Kotlin* supporte la définition d'accesseurs et mutateurs :

```
class CompteBancaire {
    protected var solde: Double = 0.0
    public var numéro: Int = 0

    public var Solde : Double
        get() {
            return solde
        }

        set(value) {
            solde = value
        }
    ...
}
```

32. *Kotlin* supporte l'héritage simple (comme *Java* et *C#*)

    a. Les membres peuvent être surchargés via le mot réservé `override`

    b. Le mot réservé `super` permet d'invoquer un membre de la classe de base plutôt que celui surchargé