Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2021 – 2022
MOBG5
JLC - SDR

MOBG5 – Développement mobile

TD1 – Kotlin

Introduction à un nouveau langage

> **Consignes**
> Ce TD se concentre sur l'apprentissage des bases du langage `Kotlin`. Les exercices sont à réaliser avec l'outil `IntelliJ`.

1 Kotlin

`Kotlin` est un langage de programmation orienté objet et fonctionnel, avec un typage statique qui permet de compiler pour la machine virtuelle Java.

Son développement provient principalement d'une équipe de programmeurs chez `JetBrains` basée à Saint-Pétersbourg en Russie (son nom vient de l'île de Kotline, près de Saint-Pétersbourg).

Google et Kotlin

Le **8 mai 2019**, lors de la conférence Google I/O, **Kotlin devient officiellement le langage de programmation voulu et recommandé** par Google pour le développement des applications Android.

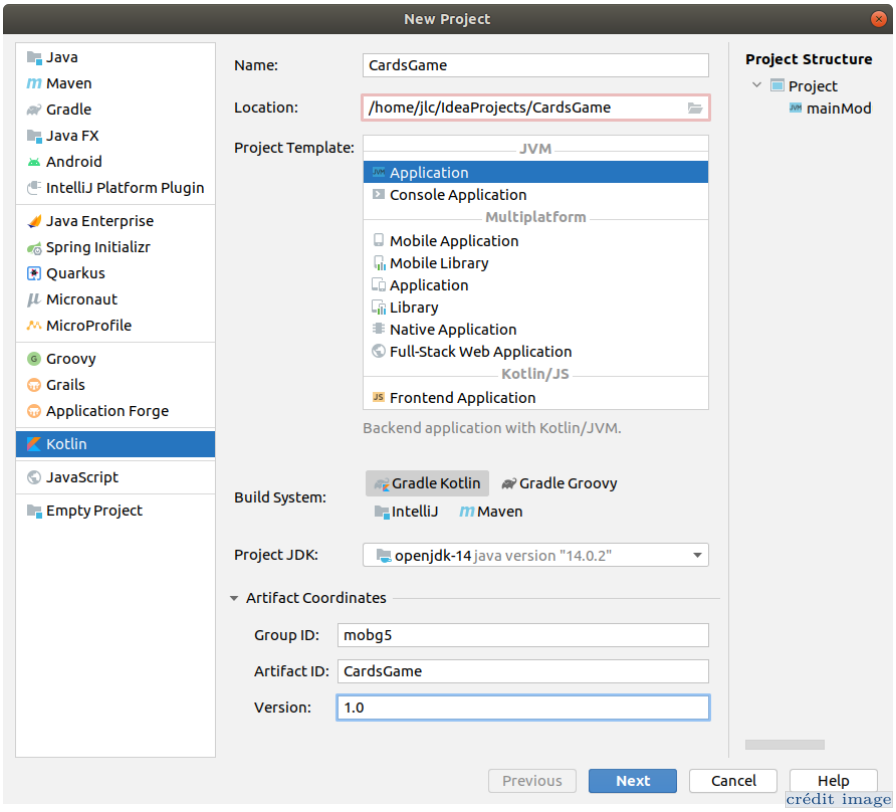
2 IntelliJ

Si ce n'est déjà fait commencez par installer IntelliJ [1] .

Démarrez l'IDE et créez un nouveau projet Kotlin intitulé `CardsGame` comme le montre la figure ci-dessous :

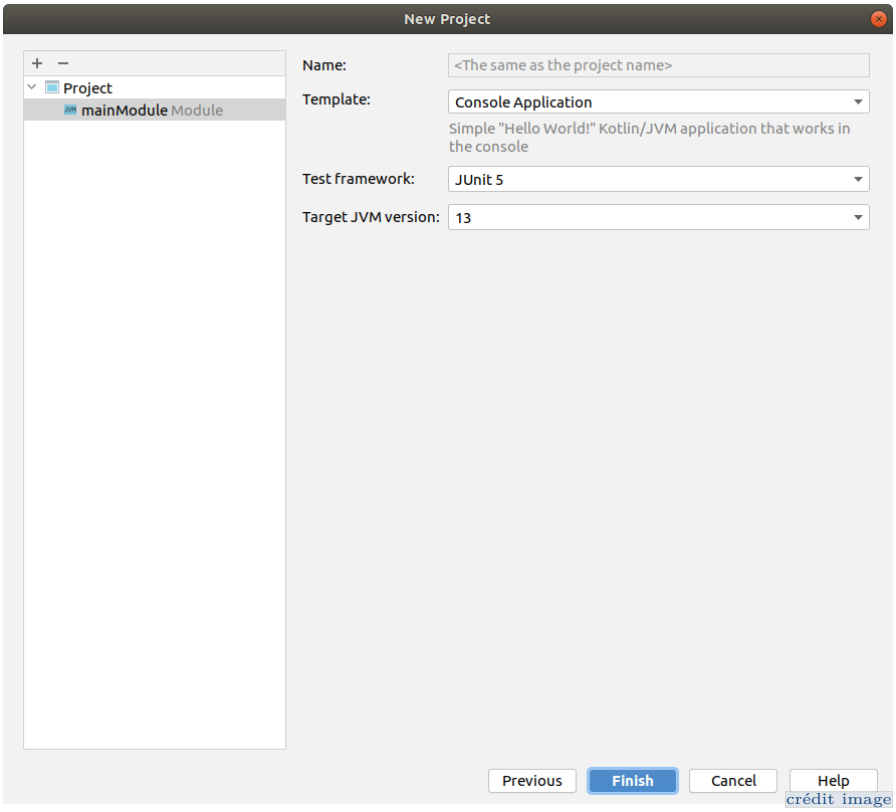
1. https://www.jetbrains.com/idea/

Version du document
8 septembre 2021



crédit image

Su le second écran de création de l'application, sélectionnez le template `Console Application`



crédit image

Consultez le fichier `main.kt` généré qui contient le traditionnel `Hello World`.

```
fun main(args: Array<String>) {
    println("Hello world")
}
```

Vous pouvez sur ce simple exemple constater les points suivants :

Premiers constats

- La visibilité public est celle par défaut ;
- Vous pouvez écrire des fonctions non attachées à une classe ;
- Les points virgules ne sont plus nécessaires.

3 Les bases

Dans cette section, vous allez explorer la syntaxe de `Kotlin`. Une liste exhaustive de tous les mots-clés et opérateurs est disponible dans la documentation officielle de Kotlin [2] .

3.1 Packages et imports

La première ligne d'un fichier `Kotlin` définit le package de travail. Si aucun package n'est définit, le contenu du fichier appartient au **package anonyme** par défaut. Ensuite le fichier renseigne les **imports** nécessaires. Ces imports sont des classes **et des fonctions**.

```
package be.esi.mobg5
    import foo.bar
    import footoo.bar as footoo
```

Afin d'éviter tout problème de noms identiques, Kotlin vous permet de renommer les packages avec le mot clé **as**.

Dans la documentation officielle sur les packages [a] , vous trouverez la liste des packages chargés par défaut.

a. https://kotlinlang.org/docs/reference/packages.html#default-imports

Contrairement à Java, Kotlin vous permet d'importer des fonctions individuelles depuis d'autres packages. Pour ce faire, on indique le chemin complet de la fonction :

```
import foo.bar.myFunction
```

Exercice 1

Création de package

Créez dans votre projet `CardsGame` les packages suivant :

- `blackjack.model`
- `blackjack.view`
- `blackjack.controller`

Déplacez ensuite votre fichier `main.kt` dans le package `blackjack`. Vérifiez que la première ligne du fichier mentionne correctement le package du fichier et exécutez votre application.

2. https://kotlinlang.org/docs/reference/keyword-reference.html

3.2 Variables

`Kotlin` connaît deux types de variables :

- ▷ les variables immuables en lecture seule : `val`
- ▷ les variables dont la valeur est modifiable : `var`

Exercice 2 **Les immuables**

Modifiez la fonction main du fichier main.kt avec le code ci-dessous.

```
fun main(args: Array<String>) {
    val name1 = "Clark Kent"
    name1 = "Superman"      // Erreur de compilation car immuable

    var name2 = "Clark Kent"
    name2 = "Superman"
}
```

Quelle erreur la compilation du code provoque-t-elle ?

Dans l'exercice précédent, `Kotlin` a déterminé seul le type de valeur des variables. Il est toutefois possible d'indiquer individuellement ces types de base.

3.3 Types de base

`Kotlin` travaille avec certains types de variables et de classes. Contrairement à Java **chaque type est un objet**.

Nombres

Il n'est pas obligatoire de renseigner le type d'un nombre dans votre programme. Le compilateur infère le type, comme on le voit avec les variables `aNumber` ou `myInt` ci-dessous. Si vous souhaitez préciser le type, il suffit de l'indiquer après le nom de la variable comme pour `myShort` ou `myDouble`.

```
fun main(args: Array<String>) {
    val aNumber = 48.657
    val myLong  = 3_000_000_000
    val myInt   = 42
    val myShort : Short = 12
    val myByte : Byte = 1
    val myDouble : Double = 3.14159
    val myFloat = 2.7182818284f
}
```

`Kotlin` connaît les types de nombres suivants, ces nombres pouvant avoir une taille maximale différente :

Les types de nombres

- Long : 64 bits
- Int : 32 bits
- Short : 16 bits
- Byte : 8 bits
- Double : 64 bits
- Float : 32 bits

N'hésitez pas à consulter la documentation[3] pour connaître les différentes syntaxes envisageables. Afin de permettre une meilleure lisibilité, les séparateurs de milliers sont représentés à l'aide de tiret bas (underscore) : `val myLong = 3_000_000_000`.

Notez que les conversions entre types de nombre sont possibles.

```
val myInt = 42
val myLong= myInt.toLong()
```

String et Caractères

Pour utiliser une chaîne de caractères dans `Kotlin`, vous pouvez placer le texte entre des guillemets doubles. Si vous souhaitez intégrer plusieurs lignes de texte, il est nécessaire d'utiliser trois guillemets doubles.

```
fun main(args: Array<String>) {
    val aString = "Ce string comporte une seule ligne."
    val aLongString = """Ce string s'étend
    sur plusieurs lignes."""

    val escapeString = "Ce string contient un caractère d'échappement \$."

    val aCar = 'A'

    println("aString $aString")
    println("name2 $aLongString")
    println("escapeString $escapeString")
    println("aCar $aCar")
}
```

Le caractère d'échappement est comme en Java la barre oblique. Le symbole dollar permet d'insérer la valeur d'une variable définie plus haut.

La définition d'une variable caractère se fait par l'utilisation des guillemets simples.

Booléen

Le type de base `Boolean` est également définit en `Kotlin` : `val myBoolean = true`. Les opérateurs logiques que vous connaissez (&&, ||, !) de Java sont également utilisés.

Exercice 3

Entrée utilisateur

Sachant que pour lire les entrées de l'utilisateur au clavier vous pouvez utiliser la fonction `readLine()`, modifiez la fonction `main` afin de demander à l'utilisateur son nom pour afficher ensuite un message de bienvenue dans la console

3. https://kotlinlang.org/docs/reference/basic-types.html#numbers

```
Veuillez entrer votre nom :
JLC
Bienvenue JLC
```

Le mot clé utilisé devant la variable qui enregistre le nom de l'utilisateur est-il `var` ou `val` ?

3.4 Arrays Kotlin

Dans `Kotlin`, un Array[4] est une collection de données. Vous pouvez en construire une instance via `arrayOf()` ou `Array()`. La première de ces fonctions s'utilise comme suit :

```
val myArray1 = arrayOf(0, 1, 2, 3, 4, 5)
```

On génère ainsi un tableau avec des chiffres de 1 à 5. Ces collections peuvent toutefois abriter d'autres types, comme des chaînes de caractères et des booléens, voire **un mélange des deux**. Si on souhaite limiter le tableau à un type, il suffit de l'indiquer dans la fonction.

```
val myArray2 = intArrayOf(10, 20, 30)
val myArray3 = booleanArrayOf(true, true, false)
val myArray4 = arrayOf(1, true, 3)
```

Une autre manière d'instancier un tableau est d'utiliser le constructeur `Array()`. Vous devez dans ce cas indiquer la taille du tableau et une fonction lambda qui génère les éléments.

```
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { println(it) }
```

Vous pouvez parcourir les éléments du tableau via la fonction `forEach()`.

Si vous souhaitez obtenir l'élément de position `i` au sein d'un tableau, la notation en crochet est utilisée. Le premier élément ayant l'index 0.

```
fun main() {
    val myArray5 = arrayOf("JLC", "SDR", "SRV")
    println(myArray5[2])
}
```

3.5 Les listes

`Kotlin` distingue les listes **immuables** qui se construisent via `listOf()` et les listes variables qui utilisent `mutableListOf()`. Vous retrouvez l'utilisation que vous connaissez de Java pour ces objets avec les méthodes `size`, `get`, `remove`,....

```
fun main(args: Array<String>) {
    val words = listOf("pen", "cup", "dog", "spectacles")
    println("The list words contains ${words.size} elements.")
    println("First element is ${words.get(0)}")
    println("First element starting with s is ${words.first { w -> w.startsWith('s') }}")

    val wordsOrder = words.sorted()
    wordsOrder.forEach { e -> println(e) }
```

4. https://kotlinlang.org/docs/reference/basic-types.html#arrays

```
    val nums = listOf(11, 5, 3, 8, -1, 9, -6, 2)
    // nums.add(7); //liste immuable

    val msg = """ List of numbers
    max: ${nums.max()}, min: ${nums.min()},
    count: ${nums.count()}, sum: ${nums.sum()},
    average: ${nums.average()}
    There are ${nums.count { e -> e < 0 }} negative values
    """
    println(msg.trimIndent())

    val numbers = mutableListOf(3, 4, 5)
    numbers.add(6)
    numbers.add(7)
    numbers.addAll(listOf(8, 9, 10))
    numbers.add(0, 0)
    numbers.add(1, 1)
    numbers.add(2, 2)

    println(numbers)

    numbers.shuffle()
    println(numbers)

    numbers.removeAt(0)
    numbers.remove(10)
    println(numbers)
    numbers.clear()

    if (numbers.isEmpty()) println("The list numbers is empty")
    else println("The list numbers is not epty")
}
```

3.6 Intervalles

On peut définir un Range[5] comme un type allant d'un point à un autre. Pour générer un intervalle, on utilise l'opérateur `..` ou les fonctions `rangeTo()` ou `downTo()`.

```
val range1 = 1..5
val range2 = 1.rangeTo(5)
val range3 = 5.downTo(1)
```

Dans ces deux variantes, vous générez un intervalle avec un incrément de un. On peut parcourir l'intervalle via une boucle `for`. Afin de vérifier si une valeur fait partie de l'intervalle, on utilise l'opérateur `in`.

```
val range5 = 0..10
fun main() {
    for (n in range5) {
        println(n)
    }
    if (7 in range5) {
        println("yes")
    }
    if (12 !in range5) {
        println("no")
    }
}
```

5. https://kotlinlang.org/docs/reference/ranges.html

3.7 Fonctions

Les fonctions[6] sont créées avec le mot clé `fun`.

```
fun div(a: Int, b: Int): Int {
   return a/b
}
fun main() {
   println(div(100, 2))
}
```

Dans notre exemple l'en-tête de la fonction est composé :

- ▷ du nom de la fonction `div`
- ▷ de deux paramètres `Int a` et `Int b`
- ▷ du type retourné par la fonction sous la forme d'une variable `Int`

Les fonctions comportant une seule ligne de code sont écrites sans ouvrir d'accolade et en se passant du mot clé `return`.

```
fun div(a: Int, b: Int): Int = a/b
fun main() = println(div(100, 2))
```

Afin d'éviter une erreur due à des paramètres erronés, vous pouvez indiquer des valeurs standards lors de la définition de la fonction. Si les paramètres sont laissés libres lors de l'appel de la fonction, les valeurs par défauts seront utilisées.

```
fun div(a: Int = 10, b: Int = 5): Int = a/b
fun main() = println(div())
```

On peut également nommé les paramètres lors de l'appel.

```
fun div(dividend: Int = 10, divisor: Int = 5): Int = dividend/divisor
```

Dans ce cas, on peut appeler cette fonction de deux façons.

```
val quotient = div(100,20)
val other = div(divisor = 20,dividend = 100)
```

Si une fonction ne retourne aucun type on utile le mot clé Unit[7] :

```
fun printHello(name: String): Unit = println("Hello $name")
```

Exercice 4 Commencez la vue

Vous allez commencer à créer les fichiers nécessaires à l'implémentation du jeu de Black Jack dans votre projet.

Créez dans le package `view` un fichier `Kotlin` intitulé `ConsoleView`. Ajoutez à ce fichier trois fonctions :

- ▷ `initialize` : qui affiche le message "*Bienvenue au Black Jack*" ;
- ▷ `displayOver` : qui affiche le message "*La partie est terminée*" ;
- ▷ `askName` : qui demande à l'utilisateur son nom et le retourne.

6. https://kotlinlang.org/docs/reference/functions.html
7. https://kotlinlang.org/docs/reference/functions.html#unit-returning-functions

3.8 Lambdas

Une fonction lambda est une fonction qui n'appartient ni à une classe ni à un objet. Elle est appelée sans utiliser le mot-clé `fun`. En principe, les fonctions lambda s'utilisent comme les variables de type `val` et sont également générées de cette façon.

```
fun main() {
    val myMessage = { println("Coucou tout le monde !") }
    myMessage()
}
```

Les expressions lambda peuvent également recevoir des arguments. Ces arguments sont identifiés par une flèche qui sépare les paramètres du noyau de l'expression.

```
fun main() {
    val div = {a: Int, b: Int -> a/b}
    println(div(6,2))
}
```

3.9 Boucle

Trois types de boucles sont disponibles :

- ▷ `while`
- ▷ `do..while`
- ▷ `for`

Elles se comportent comme leurs équivalents dans les autres langages de programmation. Par exemple une boucle while peut s'écrire comme :

```
fun main() {
    var n = 1
    while (n <= 10) {
        println(n++)
    }
}
```

Une boucle for peut elle s'écrire comme : .

```
val myRange = 0..10
fun main() {
    for (n in myRange) {
        print("$n ")
    }
}
```

3.10 Condition

Deux possibilités de branchement sont disponibles :

- ▷ `if..else`
- ▷ `when`

La structure `if..else` suit la syntaxe Java.

```
fun main(args: Array<String>) {
    val number = 2
    if (number % 2 == 0) {
        println("Number is even")
    } else {
        println("Number is odd")
    }
}
```

Par contre l'expression `when` est une spécificité de `Kotlin`. Des actions différentes peuvent être réalisées en fonction de différents états. L'effet de l'expression `when` est assez similaire à celui de l'expression `switch` en Java,

```
var age = 17
fun main() {
    when {
        age > 18 -> println("Tu es trop vieux !")
        age == 18 -> println("Déjà adulte !")
        age == 17 -> println("Bienvenue !")
        age <= 16 -> println("Tu es trop jeune !")
    }
}
```

L'argument peut toutefois être transmis directement à `when` et ne doit pas être répété à chaque fois dans le corps. Par ailleurs, une condition unique peut déclencher plusieurs actions. Pour cela, il suffit de créer un nouveau corps avec des accolades.

```
fun multi(a: Int, b: Int, c: Int): Int {
    return a*b*c
}
fun main() {
    val d = "yes"
    when (d) {
        "no" -> println("Aucun calcul")
        "yes" -> {
            println("Démarrer le calcul")
            println(multi(5, 2, 100))
            println("Calcul terminé")
        }
        else -> println("Saisie erronée")
    }
}
```

Exercice 5 **Entrée robuste utilisateur**

Modifiez le fichier `ConsoleView` et ajoutez la fonction `askBoolean` qui prend en paramètre une question (une chaîne de caractères), par exemple "*Voulez-vous rejouer ?*" et qui :

- ▷ affiche la question ;
- ▷ demande à l'utilisateur d'entrer sa réponse (*oui* ou *non*) ;
- ▷ tant que l'utilisateur n'a pas répondu par *oui* ou par *non*, la question lui est demandée ;
- ▷ retourne `true` si la réponse est *oui* et `false` si la réponse est *non*.

3.11 Les énumérations

Les énumérations fonctionnent comme en Java. Par exemple, nous pouvons définir une énumération représentants les valeurs des cartes.

```
enum class Value() {
   ACE,TWO,THREE,FOUR,FIVE,
   SIX,SEVEN,EIGHT,NINE,TEN,
   JACK,QUEEN,KING
}
```

Ces énumérations peuvent posséder un constructeur afin de fixer la valeur de certains attributs, comme le score associé à chaque carte.

```
enum class Value(val score : Int) {
   ACE(11),TWO(2),THREE(3),FOUR(4),FIVE(5),
   SIX(6),SEVEN(7),EIGHT(8),NINE(9),TEN(10),
   JACK(10),QUEEN(10),KING(10)
}
```

Ce qui permet d'y accéder via

```
val score = Value.ACE.score
```

Exercice 6 **Caractéristiques des cartes**

Ajoutez dans le package `model` les deux énumérations suivantes :

- ▷ `Value` qui représente les valeurs des cartes avec leurs scores ;
- ▷ `Color` qui représente les 4 couleurs des cartes (*HEART*,*DIAMOND*,*CLUB*,*SPADE*).

3.12 Classe

Les classes[8] de Kotlin sont des collections de données et de fonctions. Pour définir une classe, il suffit d'utiliser le mot-clé `class`. Par exemple on peut définir une classe représentant une carte de notre jeu comme :

```
class Card {
   val value: Value
   val color: Color

   constructor(value: Value, color: Color) {
      this.value = value
      this.color = color
   }
}
```

Le constructeur reconnaissable au mot clé `constructor` permet de créer une instance d'une classe.

On distingue deux types de constructeurs :

- ▷ un *primary constructors* : un constructeur avec une syntaxe abrégée
- ▷ les *secondary constructors* : des constructeurs avec une syntaxe similaire à Java

Il est possible de se passer de *secondary constructors* et d'utiliser plutôt un *primary constructor*. Par exemple notre classe peut s'implémenter avec un unique *primary constructors* :

8. https://kotlinlang.org/docs/reference/classes.html

```
class Card constructor(val value: Value, val color: Color)
```

Remarquez que si vous ne souhaitez pas apporter d'informations complémentaires sur la visibilité de la classe autrement dit si vous conservez la visibilité de la classe comme `public`, vous pouvez vous passer entièrement du mot-clé `constructor`.

```
class Card(val value: Value, val color: Color)
```

Ces trois exemples de code génèrent le même résultat. Vous pouvez à présent utiliser cette classe dans le reste de votre code.

```
val card1 = Card(Value.ACE,Color.HEART)
val card2 = Card(Value.QUEEN,Color.SPADE)
```

La notation pointée permet d'accèder aux **propriétés** d'un objet.

```
class Card(val value: Value, val color: Color)

val card1 = Card(Value.ACE,Color.HEART)
val card2 = Card(Value.QUEEN,Color.SPADE)

fun main() {
   println(card1.value)
}
```

Nous reviendrons sur cette notion de propriété plus loin et essayerons les distinguer de la notion d'attribut et de champs.

Vous pouvez également définir des valeurs par défaut au paramètre du constructeur, par exemple :

```
class Card (val value: Value = Value.ACE, val color: Color = Color.SPADE)
```

Exercice 7 **La classe carte**

Ajoutez dans le package `model` la classe `Card`. Cette classe possède deux propriétés :

- ▷ `value` qui représente la valeur de la carte ;
- ▷ `color` qui représente la couleur de la carte.

Ajoutez à la classe `Card` la méthode `isAce()` qui retourne `true` si la carte est un AS et `false` dans le cas contraire.

Ensuite générez la méthode `equals` et `hashCode` via `IntelliJ`. Un clic droit au sein de la classe vous donne accès au menu `Generate...` qui permet de générer ces méthodes pour vous.

Dans la fonction `main` testez le code ci-dessous :

```
val card1 = Card(Value.ACE, Color.HEART)
val card2 = Card(Value.ACE, Color.HEART)

val isEquals = card1 == card2
val isSame = card1 === card2

println("isEquals $isEquals")
println("isSame $isSame")
```

Que déduisez-vous des opérateurs `==` et `===` ?

3.13 Null

L'exception `NullPointerException` survient en Java lorsque l'on tente d'accéder à la référence d'un objet dont la valeur est `null`.

`Kotlin` contourne ce problème[9] en **refusant** d'emblée que les variables prennent la valeur `null`. Si le cas se présentait, le message *« Null can not be a value of a non-null type String »* apparaît à la compilation.

```
var name : String       // Erreur de compilation
var name : String = null  // Erreur de compilation
var name = "MOBG5"   // Valide
var name : String? = null // On utilise le type String? qui signifie String nullable
```

Il existe toutefois des situations où l'on souhaite utiliser la valeur `null` à dessein. Comme on le voit dans l'exemple précédent, `Kotlin` utilise dans ce cas l'opérateur `?`

3.14 Initialisation des instances de classes

Dans une classe une méthode `init` sensiblement équivalente à la méthode `initialize` de *JavaFX* est disponible. Son objectif est d'initialiser les propriétés et les attributs dont la classe a besoin lors de son instanciation.

Par exemple, si nous créons la classe `Deck` qui va contenir notre jeu de cartes, nous avons besoin d'une propriété qui contient la liste des cartes. Une fois cette classe instanciée il faut instancier les 52 cartes du jeu et les placer dans la liste des cartes. On peut écrire

```
class Deck {

    val cards = mutableListOf<Card>()

    init {
        println("DEBUG – Deck init() – Instanciation des cartes")
        for (color in Color.values()) {
            for (value in Value.values()) {
                val card = Card(value, color)
                cards.add(card)
            }
        }
    }
}
```

Remarquez

- ▷ l'utilisation du constructeur `mutableListOf<Card>()`
- ▷ le parcours des énumartions via `for (color in Color.values())`

Exercice 8 **Plusieurs méthodes d'initialisation**

Ajoutez dans le package `model` la classe `Deck`. Cette classe possède une propriété : la liste des cartes du jeu. Remplissez cette liste via la méthode `init()` décrite précédement.

Ajoutez les deux méthodes suivantes à la classe `Deck` :

- ▷ `shuffle()` : qui mélange les élements de la liste ;

9. https://kotlinlang.org/docs/reference/null-safety.html

▷ `hit()` : qui enlève un élément de la liste et le retourne.

Que ce passe-t-il si après la première méthode `init()`, vous ajoutez le code ci-dessous

```
init {
    println("DEBUG – Deck init() – 2")
}
```

Instanciez `Deck` dans votre `main` pour vérifier quelle méthode `init()` est appelée en premier.

Vous trouverez plus d'informations sur la méthode `init()` dans la documentation [10] .

3.15 Initialisation retardée : lateinit

Dans certaines situations on ne dispose pas des informations nécessaires à la création d'une instance pour alimenter toutes ses propriétés. Dans ce cas un mécanisme d'initialisation spécial est prévu via le mot clé : `lateinit`.

Prenons l'exemple de la classe `Game` qui va gérer notre jeu de black jack. Ce jeu possède pour l'instant deux propriétés : le deck et le nom du joueur.

Lors de l'instanciation de `Game`, on peut instancier le deck et le mélanger dans la méthode `init()`. Cependant dans notre jeu, nous décidons que le nom du joueur est demandé plus tard via une méthode `addName(name : String)`. L'objectif est de controller que le nom est cohérent (n'est pas vide, n'a pas déjà été donné,...).

Pour développer cette classe `Game` le mot clé `lateinit` va nous être utile.

```
class Game {

    val deck = Deck() //déclaration et instanciation
    lateinit var playerName : String

    init {
        deck.shuffle() // mélange des cartes
    }

    fun addName(name : String){
        playerName = name
    }
}
```

Regardez la documentation de latenint [11] pour en savoir plus.

Exercice 9 **La classe Game**

Ajoutez dans le package `model` la classe `Game` décrite ci-dessus.

3.16 Exception

Concernant les exceptions [12] , comme vous pouvez le voir dans la documentation la seule différence avec Java est que les *checked exceptions*(les exceptions qui doivent être mentionnées dans la signature des méthodes) n'existent pas.

Exercice 10 **Gérer les erreurs**

10. https://www.programiz.com/kotlin-programming/constructors#init
11. https://kotlinlang.org/docs/reference/properties.html#late-initialized-properties-and-variables
12. https://kotlinlang.org/docs/reference/exceptions.html

Modifiez la méthode `addName(name : String)` de la classe `Game` pour envoyer une `IllegalArgumentException` si le jom donné en paramètre est vide. pensez a utiliser la méthode `isBlank()` de `String`.

3.17 Visibilité

Les classes, les interfaces, les constructeurs, les fonctions et les propriétés peuvent voir leurs visibilités[13] modifiées. Les mots clés prévus pour cette modifications sont *private, protected, internal et public*. Le modificateur par défaut est contrairement à java *public*.

Exercice 11 Changer la visibilité

Modifiez la visibilité de la classe `Deck` et ses méthodes en `internal`.

3.18 Data Class

Une data class[14] est une classe conçue pour enregistrer uniquement des données.

```
data class User (var username: String, var name: String, var age: Int)
```

Cette classe peut être utilisée de la même manière qu'une classe classique.

```
data class User (var name: String, var age: Int)

fun main() {
    val user1 = User ("SpongeBob", 24)

    println("""Name: ${user1.name}""")
    println("""Age: ${user1.age}""")
    println("""toString: $user1""")

    val user2 = User ( "SpongeBob", 24)

    val isEquals = user1 == user2
    val isSame = user1 === user2

    println("isEquals $isEquals")
    println("isSame $isSame")
}
```

> Les méthodes `equals`, `hashCode`, `toString` et `copy` sont réécrites automatiquement.

Exercice 12 Création d'un DTO

Ajoutez le package `blackjack.dto` et créez une data class `ScoreDto` qui possède deux propriétés :

- ▷ `bank` : un entier qui représente le score des cartes de la banque ;
- ▷ `player` : un entier qui représente le score des cartes du joueur.

13. https://kotlinlang.org/docs/reference/visibility-modifiers.html
14. https://kotlinlang.org/docs/reference/data-classes.html

3.19 Object

Pour créer un *singleton* il suffit d'utiliser le mot clé `object` dans l'en-tête de la classe. Dans ce cas il n'existera qu'une seule instance de cette classe.

Par exemple, on peut ajouter un singleton qui lit le contenu d'un fichier présent dans les ressources de l'application et qui peut ajouter des éléments à ce fichier : un `Data Access Object` ?

```
import blackjack.dto.ScoreDto
import java.io.File

object ScoreDao {

   private val file = File(ClassLoader.getSystemResource("myFile.txt").file)

   fun selectAll(): List<ScoreDto> {
      val scores = mutableListOf<ScoreDto>()
      val list = file.readLines(Charsets.UTF_8)
      for (text in list) {
         val line = text.split(" ")
         val current = ScoreDto(line[0].toInt(), line[1].toInt())
         scores.add(current)
      }
      return scores
   }

   fun insert(item: ScoreDto) = file.appendText("\n$item", Charsets.UTF_8)
}
```

Exercice 13 **Création d'un Dao et d'un Repository**

Ajoutez le package `blackjack.data` et créez la classe singleton `ScoreDao` présentée ci-dessus.

Ajoutez le package `blackjack.repository` et créez la classe singleton `Repository` avec les deux méthodes suivantes :

- ▷ getAll() : qui retourne tous les élements du fichier via un appel à la méthode `selectAll()` de la classe `ScoreDao` ;
- ▷ add(item : ScoreDto) : qui insère un élement dans le fichier via un appel à la méthode `insert(item: ScoreDto)` de la classe `ScoreDao`.

Placez un fichier texte vide intitulé `myFile.txt` dans le dossier ressources de votre projet, c'est à dire dans `src/main/resources`.

3.20 Static et Companion

Si vous avez besoin de définir une fonction qui n'est pas attachée à une instance en particulier mais à une classe vous pouvez utiliser les mots clés `companion object`.

Par exemple, nous aimerions définir dans la classe `Game` le score limite du jeu de black jack (21) et le score minimal que la banque essaera toujours d'obtenir au minimum (16). En Java nous utiliserions des variables `static`, mais en `Kotlin` nous écrirons :

```
class Game {
   companion object {
      val MAX = 21
      val MIN_IA = 17
   }
```

Vous pouvez accéder à ces variables via la notation pointées : `Game.MAX`.

Dans notre cas, ces variables sont des constantes du programme, ce que nous pouvons spécifier via le mot clé `const`.

```
class Game {
    companion object {
        const val MAX = 21
        const val MIN_IA = 17
    }
```

Exercice 14 **Game et son companion object**

Ajoutez `companion object` de la classe `Game` comme présenté ci-dessus.

3.21 Propriétés

En `Kotlin` on définit au sein des classes des propriétés [15] . Pour faire simple une propriété est un attribut dont l'accesseur et le mutateur sont générés par défaut. Par exemple ce code `Kotlin`,

```
class Person {
    var name: String = "SpongeBob"
}
```

est équivalent au code java suivant :

```
public class Person {
    private String name = "SpongeBob";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Lorsque l'on accède à une propriété d'une instance `person = Person()` via `person.name`, l'expression `person.name` appelle l'accesseur de la classe `Person`.

On peut s'en rendre compte en modifiant cet accesseur pour par exemple retourner le nom de la personne en majuscule. Il suffit après la déclaration de la propriété de déclarer une fonction `get()` comme ci-dessous :

```
class Person {
    var name: String = "SpongeBob"
        get() = field.capitalize()
}
```

L'expression `person.name` retourne dorénavant le nom de la personne en majuscule car elle passe automatiquement par l'accesseur.

La mise à jour de l'accesseur utilise un champ (`field`) pour conserver temporairement la valeur de la chaîne de caractères (la valeur *SpongeBob*). Mais ce champs `field` est utilisable **uniquement** au sein des accesseurs de la propriété. Le reste de la classe ne peut pas y accéder.

15. https://kotlinlang.org/docs/reference/properties.html

Si on souhaite toutefois accéder au champ de la propriété à l'extérieur des accesseurs, on peut utiliser le principe des `Backing Properties`. Il s'agit de créer une propriété privée au sein de la classe et de modifier l'accesseur de la propriété initiale pour qu'elle pointe sur cette propriété privée.

```
class Person {
    private var _name: String = "SpongeBob"

    var name: String
        get() = _name.capitalize()
```

De cette manière une méthode de la classe peut interagir avec le contenu de la propriété sans passer par l'accesseur.

> L'underscore présent devant le nom des `Backing Properties` est une convention d'écriture en `Kotlin`.

Backing property et encapsulation

L'utilisation des `Backing Properties` peut servir a encapsuler des propriétés dans une classe. Par exemple on peut :

- ▷ créer une propriété privée avec une liste variable ;
- ▷ créer une propriété publique immuable dont le champs pointe vers la propriété privée.

```
private val _hand = mutableListOf<Card>()

internal val hand: List<Card>
    get() = _hand
```

A l'intérieur de la classe, on peut mettre à jour la liste d'utilisateurs mais à l'extérieur de la classe on a uniquement accès à l'accesseur qui retourne une liste immuable.

Exercice 15 **La classe Player**

Ajoutez dans le package `model` la classe `Player`.

Cette classe a un constructeur avec une propriété name, qui est une chaine de caractères représentant le nom du joueur.

Ajoutez une propriété `hand` qui est une liste de carte que le joueur a en main. Utilisez le système de backing properties en créant une variable privée `_hand` comme expliqué ci-dessus.

Ajoutez une propriété entière nommée `score` qui retourne le score des cartes dans la main du joueur. Son accesseur va parcourir la liste `_hand` pour additionner le score de chaque carte.

Ajoutez finalement les méthodes :

- ▷ add(card : Card) : qui ajoute une carte à la main du joueur ;
- ▷ clear() ; vide la main du joueur.

3.22 Interface

Les interfaces[16] dans Kotlin ressemblent aux interfaces Java.

```
interface SimpleInterface {
    fun firstMethod(): String

    fun secondMethod(): String {
        return("Hello, World!")
    }
}
```

Cette interface contient deux fonctions. Une fonction abstraite et une fonction avec une implémentation par défaut. Nous pouvons également y ajouter des propriétés.

```
interface SimpleInterface {
    val firstProp: String

    val secondProp: String
        get() = "Second Property"

    fun firstMethod(): String

    fun secondMethod(): String {
        return("Hello, from: " + secondProp)
    }
}
```

Dans notre exemple une des deux propriétés possèdent une implémentation par défaut de son accesseur. Notez que les propriétés d'une interface ne peuvent pas conserver d'état. L'expression ci-desous est illégale en Kotlin :

```
interface SimpleInterface {
    val firstProp: String = "First Property"//Illegal declaration
}
```

Pour implémenter notre interface, nous pouvons écrire :

```
class SimpleClass: SimpleInterface {
    override val firstProp: String = "First Property"
    override fun firstMethod(): String {
        return("Hello, from: " + firstProp)
    }
}
```

Exercice 16 L'interface Model

Ajoutez dans la package `model`, l'interface ci-dessous.

```
interface Model {
    val bank: Player
    val winner: Player

    fun addName(name: String)
    fun start()
    fun canHit(): Boolean
    fun hit()
}
```

La classe `Game` doit au minimum implémenter cette interface.

16. https://kotlinlang.org/docs/reference/interfaces.html

3.23 Heritage

La classe `Any` est la classe mère par défaut de toutes les classes en Kotlin. Cette classe implémente les trois méthodes `equals()`, `hashCode()` et `toString()`.

L' hétitage[17] en Kotlin se déroule en deux étapes :

- ▷ la classe mère doit être modifiée avec le mot clé `open`, en effet de base les classes en Kotlin sont finales
- ▷ pour renseigner son parent la classe fille doit utiliser la notation :

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

class Circle() : Shape() {
   override fun draw() { /*...*/ }
}
```

On constate que la réécriture d'une fonction s'accompagne du mot clé `override`. Si on souhaite réécrire une propriété le mot clé est à nouveau `override` .

```
open class Shape {
   open val vertexCount: Int = 0
}

class Rectangle : Shape() {
   override val vertexCount = 4
}
```

Exercice 17 **Terminer le black jack**

Sur base des classes créées durant le TD, développez une version simplifiée du jeu Blackjack.

Le jeu se joue par manche jusqu'à ce que le joueur s'arrête. Une manche du jeu se déroule comme suit :

1. le joueur reçoit 2 cartes ;
2. le joueur choisit de demander une carte supplémentaire ou de stopper ;
3. si le joueur choisit une carte supplémentaire il la reçoit, si son score dépasse 21 il a perdu (et perd la mise), sinon il retourne au point précédent (2) ;
4. si le joueur choisit au point (2) de stopper c'est au tour de la banque de jouer ;
5. la banque joue, si elle obtient plus de 21 ou moins que le joueur le joueur gagne (et donc gagne la mise) et sinon il perd.

La stratégie de la banque est de tirer une carte jusqu'à atteindre au moins 16.

17. https://kotlinlang.org/docs/reference/classes.html#inheritance