BTS SIO – SLAM **NOM :** **LDV – 12 octobre 2022**

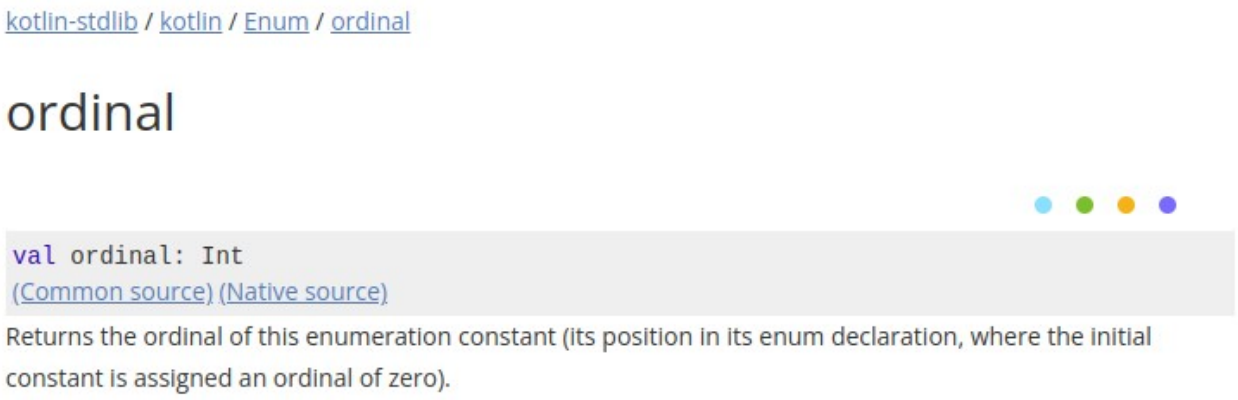
Durée 1h30 sur 50 points (arrondi)

A/ Un étudiant de la classe SIO22 à défini, pour le projet DevineLaCarte, la fonction de comparaison suivante, déclarée dans la classe *Carte* du projet : (**11 points**)

```
// classe Carte (extrait)

38 : override fun compareTo(other: Carte): Int {
39 :   if (this.nom.points.compareTo(other.nom.points) == 0)
40 :     this.couleur.ordinal.compareTo(other.couleur.ordinal)
41 :   else if (this.nom.points.compareTo(other.nom.points) != 0)
42 :      this.nom.points.compareTo(other.nom.points)
43 :   else 0
44 : }
```

Documentation technique :



A.1 Expliquez la logique mise en œuvre par les lignes 39 et 40. **N'hésitez pas à fournir des exemples ou à illustrer librement vos propos**. (**4 points**)

Si this et other on même nom (même valeur de points liés au nom) alors on les compare sur la valeur ordinal de la couleur (dans l'ordre de leur déclaration dans la classe enum, la couleur la plus forte en dernier).

A.2 Expliquez les conditions d'exécution de l'instruction placée en ligne 43 (**3 points**)

Aucune ! Car tous les cas on été exploités précédemment, à savoir == 0 et!= 0 : comment pourrait on avoir une autre valeur ?

Certains d'entre vous ont soulevé le cas de la valeur *null*. C'était une piste en effet, mais la fonction compare retourne une valeur non nullable, Int et non Int?

A.3 Proposez une implémentation plus optimisée de la méthode compareTo (**4 points**)

```
38 : override fun compareTo(other: Carte): Int {
39 :    val cmp: Int = this.nom.points.compareTo(other.nom.points)
40 :    if (cmp == 0)
41 :       this.couleur.ordinal.compareTo(other.couleur.ordinal)
42 :    else
43 :       cmp
44 : }
```

B/ Soit le test unitaire suivant (**2 points**)

```
@Test
fun compareCartesDeMemeCouleurMaisDeValeurDifferente()  {
  val dameDeCoeur : Carte = Carte(NomCarte.DAME, CouleurCarte.COEUR)
  val roiDeCoeur  : Carte = Carte(NomCarte.ROI, CouleurCarte.COEUR)

  assertTrue(dameDeCoeur.compareTo(roiDeCoeur) < 0 )
  assertTrue(dameDeCoeur < roiDeCoeur)
}
```

B.1 Expliquez la différence entre les deux assertions de ce test (**2 points**)

Aucune ! Comme expliqué dans le commentaire du projet initial.

voir dans CarteTest.kt, les commentaires placés dans le corps de

compareCartesDeMemeCouleurMaisDeValeurDifferente()

C/ Dans la fonction *main* du projet (*MainPlayConsole.kt*), en ligne 30 se trouve l'instruction suivante :

```
30 : val couleurCarteUserStr: String = readLine() + ""
```

C.1 Expliquez ce que fait cette ligne de code (**2 points**)

Déclaration d'une variable locale nommé couleurCarteUserStr de type String.

Initialisation de cette variable par une valeur issue d'une concaténation d'une « lecture clavier » avec une chaîne vide.

C.2 Quelle est la raison qui a poussé le développeur à effectuer une concaténation avec une chaine vide ? (**3 points**)

(vu en cours) Afin d'éviter la valeur *null* éventuelle, lorsque l'utilisateur tape ENTER directement (valeur null incompatible avec le type de la variable qui est String et non String?)

Admettons qu'une autre variable soit utilisée pour réceptionner une donnée saisie par l'utilisateur

```
25 : val dataInput: String? = readLine()
```

C.3 Donnez **deux façons différentes d'affecter** cette donnée (*dataInput*) à la variable *couleurCarteUserStr*, en réécrivant la ligne 30 précédente

C.3a (**2 points**)

```
// avec elvis operator (gérer les valeurs null est sa fonction !)
val couleurCarteUserStr: String =  dataInput ?: ""
```

C.3b (**2 points**)

```
// sans elvis operator (plus verbeux)
val couleurCarteUserStr: String =
    if (dataInput != null) dataInput else ""
```

```
Mauvaise idée : val couleurCarteUserStr: String =  dataInput!!
  // NPE assurée si dataInput vaut null

Très mauvaise idée :
 val couleurCarteUserStr: String =  dataInput.toString()
  // NPE non assumée
```

D/ Voici le code de la classe Jeu

```
1 : class Jeu constructor(
2 :   val avecAide: Boolean,
3 :   val paquet: Paquet,
4 :   paramCarteADeviner: Carte? = null)
5 :{
6 :    val carteADeviner: Carte
7 :     // le getter par défaut, inutile de le redéclarer
8 :     // (juste pour la démonstration)
9 :     // field est ici synonyme de carteADeviner (implicite backing memory
10 :    // de la propriété)
11 :   // REM : faire référence à carteADeviner au lieu de field entrainerait
12 :    // ici une récursion incontrôlée
13 :        get() = field
14 :
15 :        set(value) { field = value } // <== impossible car la propriété
16 :                                     // est en lecture seule
17 :   [autre code ici]
   :}
```

D.1/ Combien de propriétés sont déclarées dans cette classe, et lesquelles ? (**2 points**)

Réponse : 3.

2 déclarées en paramètre du constructeur (val ou var) et
1 dans le corps de la classe, en dehors des méthodes

Soit : avecAide, paquet et carteADeviner

D.2/ D'après le commentaire en lignes 11 et 12, l'absence du mot clé field entraînerait une récursion incontrôlée. Proposez une explication de ce phénomène (**3 points**)

(question difficile) Lorsque que l'on interroge une propriété d'un objet, c'est en fait son accesseur (méthode) get() qui est appelée

Ex : p.nom // appel p.getNom()

Donc, si l'implémentation du get() d'une propriété fait référence au nom de la propriété, c'est en fait get() qui sera appelé, qui lui-même appel get() etc. On arrive à une erreur de type StackOverFlow...

D.3/ Expliquez pourquoi la déclaration en ligne 15 provoque une erreur de compilation (d'après le commentaire en lignes 15 et 16) (**3 points**)

Tout simplement parce que la variable est déclarée avec le mot clé **val**, qui signifie que la variable, après son initialisation, ne pourra pas changer de valeur, ce qui est **contradictoire** avec une déclaration d'un **set** dont la fonction est de permettre la modification de la valeur de la variable.

Cette erreur de logique est détecté par le compilateur kotlin.

D.4/ Après avoir mis en commentaire la ligne 15, quelle serait l'instruction (ou le type d'instructions) qu'il faudrait placer en ligne 17 ? (**3 points**)

On attend une instruction d'initialisation de la variable, car elle n'a pas encore de valeur.

Kotlin dispose d'un mot clé pour cela : **init** qui préfixe un bloc d'initialisation (*initializer blocks*)

voir : https://kotlinlang.org/docs/classes.html#constructors

E/ D'après la définition de la classe Paquet ci-dessous

```
class Paquet constructor(var cartes: List<Carte>) {
    init {
        if (this.cartes.isEmpty()) {
            this.cartes = createJeu32Cartes()
        }
    }

    fun cardinal(): Int = cartes.size
 }
```

E.1/ Poursuivre le scénario du test suivant, afin de vérifier que l'instanciation se soit bien déroulée. (**4 points**)

D'après la logique du constructeur, le code source nous montre que si l'argument reçu est une liste vide, alors un jeu de 32 cartes prendra sa place)

On vise ici à mesurer votre capacité à identifier du sens dans du code existant (**fondamental**)

```
@Test
fun paquetCreationAvecListeDeCartesVide()  {

    val paquet = Paquet(emptyList<Carte>())
    // TODO tester l'état de l'objet référencé par la variable paquet
    assertEquals(32, paquet.cardinal())

}
```

//

F/ On souhaite tester la méthode *rebattreLesCartes* de la classe Paquet.

Le test actuel n'est pas suffisant. On vous demande de tester de façon plus qualitative, en interrogeant des cartes. Idée et algorithme de votre choix(**6 points**)

```
@Test
fun paquetRebattreLesCartes()  {
   val paquet = Paquet(createJeu32Cartes())
   assertEquals(32, paquet.cardinal())
   paquet.rebattreLesCartes()
   assertEquals(32, paquet.cardinal())

}
```

De nombreuses solutions possibles, mais toutes interrogent des cartes dans le paquet avant et après avoir rebattu les cartes.
Remarque => une étape attendue dans le projet...

G/ D'après le code de Paquet, notez les assertions justes

```
class Paquet constructor(var cartes: List<Carte>) {
    init {
        if (this.cartes.isEmpty()) {
            this.cartes = createJeu32Cartes()
        }
    }

    /**
     * Donne le nombre de cartes dans le paquet
     */
    fun cardinal(): Int = cartes.size

    /**
     * Représentation textuelle de l'état du paquet
     */
    override fun toString(): String {
        return "Paquet de ${cardinal()} cartes"
    }

    /**
     * C'est le paquet qui décide quelle sera la carte à deviner
     * @see [org.sio.slam.Jeu]
     */
    fun getCarteADeviner(): Carte {
        // TODO implémenter une solution moins prédictive !!
        return this.cartes[0]
    }
}
```

G.a Le type abstrait de la méthode **cardinal** est (**2 points**)

[ ] Unit -> Unit
[ ] Unit -> Int
[ ] Int -> Unit
[ ] Paquet -> Unit
[x] Paquet -> Int // Paquet (sa classe), Int son type de retour
[ ] Int -> Int
[ ] Int -> Paquet

G.b Le type abstrait de la méthode **getCarteADeviner** est (**2 points**)

[ ] Unit -> Carte
[ ] Unit -> Int
[x] Paquet -> Carte // Paquet (sa classe), Carte (son type de retour)
[ ] Carte -> Paquet
[ ] Carte -> Unit

H/ Sachant que l'instruction ci-dessous respecte les conventions de nommage usuelles, identifier chacune de ses parties – (évaluation du sens et du vocabulaire - à compléter sur la feuille – **6 points**)