Introduction Scala : première partie du cavalier d'Euler.

Notions Scala : classes, traits, update, apply, object, Tuple, déconstructeur, Option

Vérifiez les conventions à suivre pour l'écriture du code (cf Moodle).

Vous allez réaliser ce TD en Scala. Le programme doit fonctionner avec Scala 2.11 au minimum. Commencez par vérifier que vous avez l'environnement adéquat pour programmer en Scala (réalisez, compilez et exécutez un petit programme en Scala).

Avertissement : pour *commencer* à programmer en Scala, vous devez être à jour en développement Java 5 (généricité) a minima + être à l'aise avec les notions de classe abstraite, d'héritage, de polymorphisme, de généricité contrainte – en particulier, vous devez connaître le problème de 'type erasure'.

Tout le TD peut tenir en un seul fichier : vous pouvez définir plusieurs classes et objets dans un même fichier scala.

Vous indiquerez EXPLICITEMENT les types de retours des méthodes et des fonctions dans les déclarations systématiquement.

1. On souhaite dessiner l'état d'un échiquier en utilisant des couleurs. On peut utiliser une interface graphique (avec JavaFx), mais ici, histoire de simplifier le code, vous allez utiliser les commandes de séquences d'échappement de la console texte.

   Une séquence d'échappement commence généralement par **Esc** donc par le caractère \27 suivi par un code représenté par une série de caractères.

   Par exemple, l'affichage de \27[41m demande à la console d'afficher les caractères suivants en blanc sur fond rouge. La sortie de ce mode d'affichage se fait par le code \27[0m. Vérifiez avec un petit programme Scala pour afficher une ligne de texte coloré dans la console.

   Consultez par exemple http://www.termsys.demon.co.uk/vtansi.htm pour les codes les plus courants.

   Attention, la console windows ne suit pas nécessairement ces normes – il est possible d'installer une autre console pour windows.

2. Fabriquez un objet **Ansi** ainsi que ses méthodes **code(code_ : String) :String** et **reset:String** qui permettent d'envoyer le code passé en paramètre et de terminer une séquence d'échappement. Testez vos méthodes. Créez les méthodes de changement de couleur pour le fond et les caractères et vérifiez par exemple que :

   ***println*("coucou"+Ansi.*fBlue*+Ansi.*red*+" abcdef "+Ansi.*reset*)**

   permet d'afficher du texte rouge sur fond bleu dans la console.

3. Vous allez maintenant fabriquer la classe **class Echiquier[Piece](cote_ : Int)**. Normalement, un échiquier fait 8 cases de côté. Commencez par rajouter un constructeur secondaire dans le cas où le nombre de cases de cotés n'est pas spécifié.

   Une remarque : il est possible que le compilateur indique un message du type '**no classtag available**'. Ce message est lié aux limitations de la JVM sur laquelle tourne Scala, mais qui n'est pas conçue pour Scala. Ce message indique que le compilateur doit connaître la classe paramétrée générique et donc, doit avoir l'information à la compilation. Renseignez-vous sur ce qu'on appelle 'Type erasure' en Java (c'est une notion que vous devez absolument connaître). Vous pouvez explicitement demander au compilateur Scala de fournir cette information de la manière suivante :

   Rajoutez **import scala.reflect.ClassTag** en début de programme

   Modifiez : class Echiquier[Piece**: ClassTag**](cote_ : Int)

4. Un échiquier contient des pièces dans une matrice de **cote_ x cote_** de cases. Fabriquez une variable d'instance **plateau** pour l'échiquier. Déclarez la classe abstraite **Piece** de la manière suivante : **trait Piece** (pour l'instant, vous pouvez considérer que **trait** est équivalent à **interface**)

5. On pourra donc définir des échiquiers qui contiennent des instances de classes de **Piece**.

   Définissez pour commencer la class **PieceCol(etiquette_ : String, codeAnsi_ : String)** qui permet de créer une pièce associée à une étiquette et à un code Ansi de couleur (par défaut de couleur blanche sur fond noir).

   Redéfinissez la méthode **toString** qui donne la représentation textuelle de la pièce (avec son code ANSI).

   Vérifiez que :

   val cavalier = new PieceCol("caval", Ansi.red)

   println(cavalier)

   affiche bien le texte 'caval' en rouge.

6. Modifiez votre code pour qu'on puisse créer un cavalier de cette manière :

   val cavalierRouge = PieceCol("caval", Ansi.red)

   val cavalier = PieceCol("caval") // ecriture normale

7. On souhaite pouvoir modifier une instance d'échiquier pour lui rajouter des pièces. Rajoutez la méthode : **def placerEn(piece_ : Piece, x_ : Int, y_ : Int)**

8. La méthode précédente peut s'appeler avec **echiquier.placerEn(piece,x,y)** mais on souhaite trouver une formulation plus naturelle, du type **echiquier(x,y) = p**.

Apply ne peut pas être utilisé, à cause de '='. **Update** non plus, puisque **update** ne prend qu'un seul argument (la signature de **update** est **update(x, piece)**).

Vous allez donc transformer les deux arguments en un seul en les réunissant à l'aide d'un couple de valeurs. Un **TupleN** est une association de N valeurs. Un **Tuple2** est donc un couple de valeurs. Sa déclaration est **coupleXY :Tuple2[Int, Int].** La première valeur du couple est **coupleXY._1** et la seconde **coupleXY._2**.

Scala permet d'instancier un couple de syntaxe **(9,3)** directement en **new Tuple2(9,3)**.

Utilisez cette construction et **update** pour permettre l'écriture de :

echiquier((x,y)) = piece // notez le nombre de parenthèses

9. Fabriquez de la même manière une méthode qui vous permet d'accéder en lecture à une case x,y :

   val piece = echiquier(x,y) ou bien **val piece = echiquier((x,y))** comme vous préférez.

10. Normalement, une case de l'échiquier contient ou bien une **pièce**, ou bien **rien**. Ce n'est pas satisfaisant d'inventer une pièce 'bidon' (ou NULL) et d'utiliser cette pièce pour initialiser l'échiquier.

    Il existe un type particulier en Scala. Ce type se nomme **Option**[T]. Une variable de type **Option**[T] contient, ou bien un objet de la forme **Some(**), ou bien rien (**None**). Vous allez donc transformer votre code pour que la variable d'instance **this.plateau** soit un tableau de **Option[Piece]** plutôt qu'un tableau de **Piece**.

    Histoire de bien comprendre cette notion de conteneur de type, expérimentez à partir du code suivant :

    var valeur: Option[Int] = Some(2)

    println(valeur) // donne Some(2)

    val Some(combien) = valeur

    println(combien) // donne 2

    valeur = None

    println(valeur) // donne None

    Vous noterez la syntaxe **val Some(combien) = valeur** qui exploite ce qu'on appelle un *déconstructeur* (le contraire du **new** classique). On essaye de savoir à quoi doit être égale la valeur **combien** pour que valeur soit identifiable à **Some(combien).**

    Profitez-en pour créer une méthode qui vide une des cases de l'échiquier.

11. Et donc …

A ce stade, vous devriez avoir les signatures des méthodes suivantes dans la classe Echiquier :

def placerEn(piece_ : Piece, x_ : Int, y_ : Int) : Unit

def update(coupleXY : Tuple2[Int,Int], piece_ : Piece) : Unit

def vider(x_ : Int, y_ : Int) :Unit

def apply(x_ : Int, y_ : Int): Option[Piece] // ou avec un Tuple2

12. Maintenant, vous pouvez faire en sorte d'initialiser l'échiquier à vide à la création de l'instance. Avant, on ne pouvait pas car on n'avait pas d'élément neutre pour **Piece**.

    Contrairement à Java, vous pouvez exécuter du code dans le corps de la classe : ce code ne sera exécuté qu'une seule fois, au moment de la création de l'instance.

    Vous pouvez donc écrire :

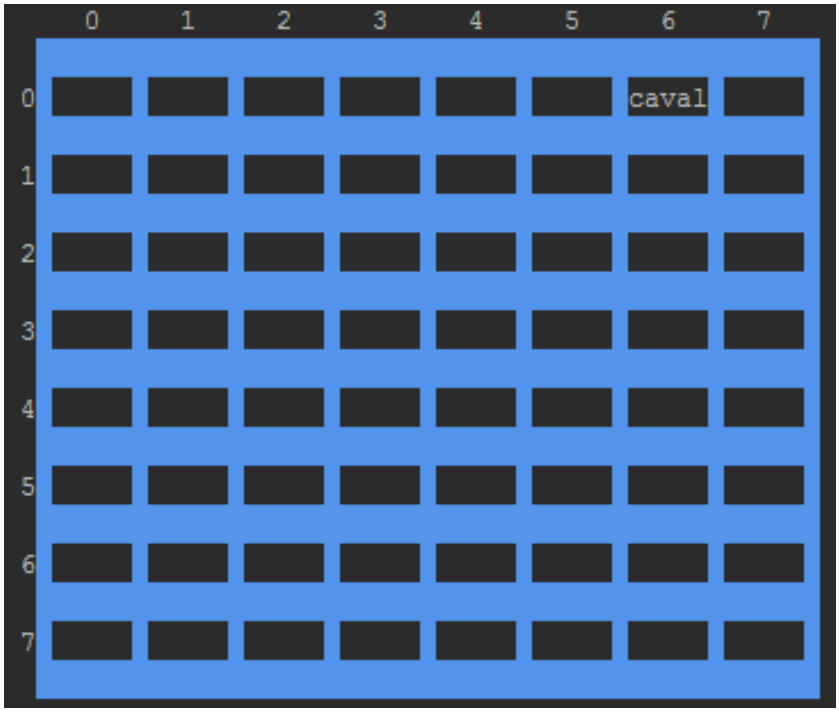
```
class Echiquier[Piece: ClassTag](cote_ : Int) {

 private var plateau = Array.ofDim[Option[Piece]](cote_, cote_)

 for( i <- 0 to cote_ -1 ; j <- 0 to cote_ -1) this.vider(i,j)

// suite de vos méthodes ici

// …

}
```

13. Enfin, vous allez écrire la méthode **toString** de la classe Echiquier pour récupérer une image textuelle de votre échiquier.

    Le code :

```
object Main {

 def main(args: Array[String]): Unit ={

  val cavalier = PieceCol("caval")

  val echiquier = new Echiquier[PieceCol]()

  echiquier((0, 6)) = cavalier

  print(echiquier)

 }}
```

Doit donner ceci dans la console :



Si nécessaire (à vous de voir si c'est le cas ou pas), vous pouvez étendre le trait **Piece** :

trait Piece {

def length: Int // donne la longueur du texte de la piece (hors codes ANSI)

}

Dans ce cas, vous aurez une erreur du compilateur qui concerne l'utilisation de la méthode **length** dans échiquier. Normalement, le type paramétrique est **Piece**, mais **Piece** est un **trait** (une espèce de class abstraite) qui n'implémente pas **length**. Donc, c'est normal que le compilateur indique que la méthode **length** n'existe pas (puisqu'elle est abstraite).

Il faut donc contraindre le type générique à être une sous-classe de Piece (qui implémente length), pas **Piece**. Vous pouvez faire cela en utilisant la syntaxe suivante :

class Echiquier[**P <: Piece: ClassTag**](cote_ : Int) {

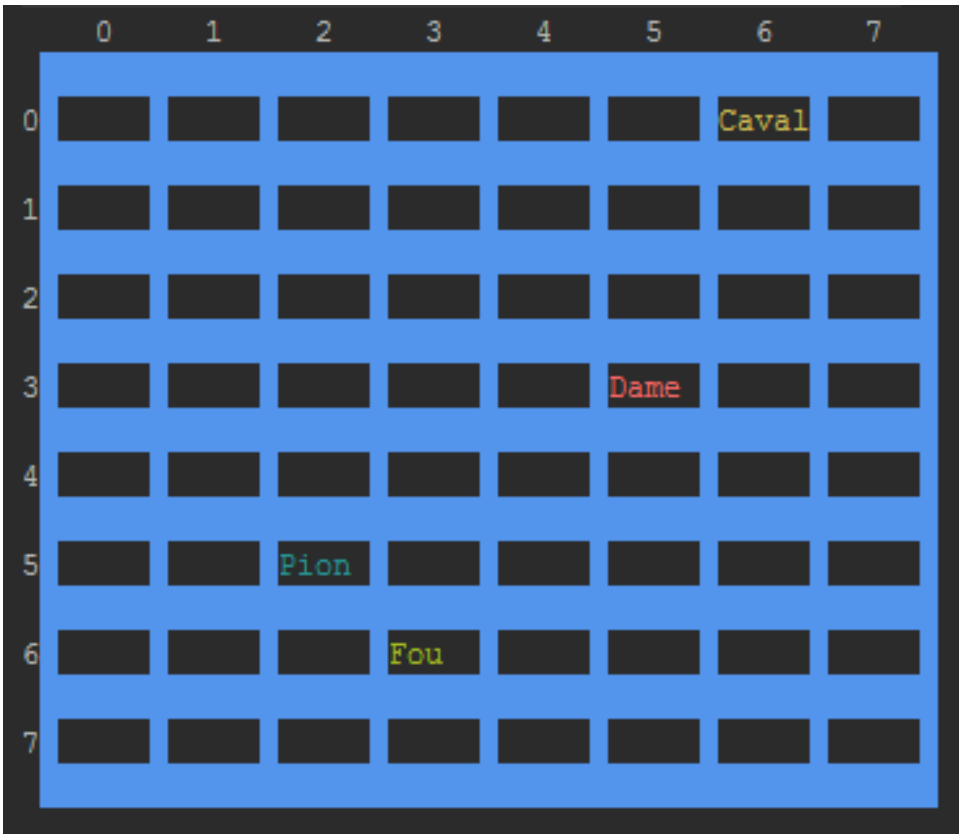
Le symbole **< :** indique que P doit être une classe qui étend **Piece**. Il faut donc également modifier dans les méthodes, le nom de la classe générique qui devient **P**.

14. Vérifiez que vous pouvez représenter un échiquier comme :

object Main {

def main(args: Array[String]): Unit ={

```
    val echiquier = new Echiquier[PieceCol]()
    echiquier((0, 6)) = PieceCol("caval")
    echiquier((3, 5)) = PieceCol("dame", Ansi.red)
    echiquier((6, 3)) = PieceCol("fou", Ansi.green)
    echiquier((5, 2)) = PieceCol("pion", Ansi.cyan)
    print(echiquier)
  }
}
```



15. Maintenant, vous allez modifier votre code pour permettre les accès suivants :

```
object Main {

  def main(args: Array[String]): Unit ={

    val echiquier = new Echiquier[PieceCol]()

    echiquier((0, 6)) = Cavalier()   // indice : utilisez un object Cavalier et un apply()

    echiquier((3, 5)) = Dame()

    echiquier((6, 3)) = Fou()

    echiquier((5, 2)) = Pion()

    echiquier((5, 2)) = Rien() // on vide la case

    print(echiquier)

    println(echiquier(5,2))

    println(echiquier(3,5))

  }

}
```

Ce qui donne :