Prise en main : Téléchargez Scala ici (ou directement depuis votre gestionnaire de paquets). Pour ce TP, on travaillera dans un unique fichier `tp3.scala`, dans lequel on déclarera un `object Main`, doté d'une méthode

```
def main(args : Array[String]) = {
  ...
}
```

Pour exécuter le main, on commencera par le compiler en bytecode via `scalac tp3.scala`, avant de l'exécuter via `scala Main`.

Ce TP a pour objectif de représenter des formules propositionnelles (pour rappel, il s'agit de formules construites à partir de variables et des constantes `Vrai` et `Faux` à l'aides des opérateurs `Non`, `Et` et `Ou`), dans le but de les manipuler et de les évaluer.

Question 1 Déclarez une classe abstraite `Formule`, qu'on scellera.
Déclarez ensuite les `case class` suivantes, qui héritent de `Formule` :

- `Variable(nom)`, où `nom` est une chaîne de caractères
- `Non(form)`, où `form` est une `Formule`
- `Et(formg, formd)` et `Ou(formg,formd)`, où `formg` et `formd` sont des `Formule`

Pour terminer, déclarez deux `case object` (qui fonctionnent de la même manière que des `case class`, mais sont des singletons) `Vrai` et `Faux`.

Question 2 On va commencer par travailler l'esthétique des formules. Créez une formule utilisant au moins une fois tous les cas, et affichez-la. Cette affichage est-il satisfaisant ?
En réimplémentant la méthode `toString()` via

```
override def toString() = {
  ...
}
```

faites en sorte que

- une variable n'affiche que son nom,
- l'affichage des opérateurs binaires se fasse de manière infixe (c'est-à-dire que le "ou" et le "et" devront s'afficher entre leurs arguments, et pas avant eux).

On utilisera systématiquement des parenthèses pour désambiguïser les formules, et on ne s'inquiétera pas de laisser des parenthèses inutiles.

Par exemple,

```
Et(Non(Variable("a")),Ou(Faux,Variable("b")))
```

devra s'afficher comme

```
(non(a)) et ((faux) ou (b))
```

Notons que Scala implémente pour nous le `equals` des case class : par exemple,

```
Variable("a") == Variable("a")
```

est évalué à `true`.

Question 3 On cherche maintenant à opérer des substitutions sur des formules : étant donnée une formule `f` et un mapping qui associe des formules à certaines variables, on veut remplacer dans `f` ces variables par les formules associées.
Par exemple, si l'on applique à la formule de l'exemple précédent la substitution qui à `Variable("a")` associe la formule `Non(Variable("c"))`, on obtient la formule

```
Et(Non(Non(Variable("c"))),Ou(Faux,Variable("b")))
```

On remarque que quand une variable n'a pas de valeur spécifiée par le mapping, alors elle reste inchangée dans la formule de départ.

Pour représenter les mapping, on utilisera la classe `Map[Variable,Formule]`, qui ne représente rien d'autre qu'une table associative dont les clefs sont des variables, et les valeurs des formules. On se contentera d'utiliser la méthode

```
def get(clef:Variable) : Option[Formule]
```

des instances de `Map[Variable,Formule]`. Cette méthode renvoie `None` si `clef` n'est pas une clef du mapping, ou `Some(valeur)` si la valeur associée à `clef` est `valeur`.

Implémentez la méthode

```
def substituer(sub:Map[Variable,Formule]) : Formule
```

de `Formule`, qui renvoie la nouvelle formule obtenue après l'application de la substitution `sub`.

Pour tester ce code, on pourra créer un mapping comme suit :

```
Map(Variable("a") -> Ou(Vrai,Variable("c")), Variable("b") -> Faux)
```

Il s'agit ici de la substitution qui remplace `Variable("a")` par la formule `Ou(Vrai,Variable("c"))`, et `Variable("b")` par `Faux`.

Question 4 Comme on l'a vu dans l'exemple, le résultat d'une substitution peut encore contenir des variables (appelées variables libres). Dans ce cas, évaluer la formule n'a pas de sens ; l'évaluation ne peut se faire que si plus aucune variable libre n'est présente dans la formule. Les formules sans variables libres sont appelées *formules closes*.
Écrivez la méthode

```
def evaluer(sub:Map[Variable,Formule]) : Option[Boolean]
```

de `Formule`. Si après substitution, la formule est close, le résultat devra être `Some(b)`, où `b` est le résultat `Boolean` de l'évaluation. Si la formule n'est pas close après substitution, alors cette méthode renverra `None`.

Question 5 (☠) Ce qui a été dit dans la question précédente est inexact : il est parfois possible d'évaluer une formule même quand il reste des variables libres. C'est par exemple le cas de la formule `Ou(Vrai,Variable("a"))`, qui s'évaluera à `true` quelle que soit la valeur de la variable `a`.

Affinez votre code pour que la méthode `evaluer()` retourne un résultat `Some(_)` le plus souvent possible.