420KBB – Activité de révision

En cette rentrée automnale, comme c'est souvent le cas, nous avons toutes et tous un peu de « rouille » dans le corps, alors essayons de nous remettre en forme.

Vos chics profs vous proposent cette petite activité formative, qui couvre quelques aspects clés de votre première année au DEC en Techniques de l'informatique, et vise à servir à la fois à une forme de révision, et à la fois à une amorce de réflexion sur la pratique de la programmation.

Note importante

Il arrive que le cours préalable à 420KBB soit réussi de manière « un peu limite » et que certains éléments clés requis pour passer du cours précédent à celui-ci soient… présumés acquis, mais en pratique très fragiles.

Si vous rencontrez des difficultés à réaliser cette activité, il est impératif de contacter votre enseignant(e) immédiatement! Vous n'êtes assurément pas seul(e) alors il n'y a pas de gêne à avoir, car l'important est de vous accompagner et de maximiser vos probabilités de réussite!

Prenez votre réussite entre vos mains. Il se trouve que 420KBB est un cours riche en contenu, un cours amusant mais où le rythme est rapide et soutenu. Aidez nous à vous aider et passons une belle session toutes et tous ensemble !

La situation

Vous êtes un(e) général Orque qui souhaite appeler ses troupes et les voir se réunir dans l'ordre. Pour les besoins de notre problème, un Orque se limite à son nom. Ce nom a certaines particularités :

- Il doit avoir entre `1` et `4` caractères seulement
- Il ne doit pas contenir plus d'une voyelle (ne pas tenir compte des accents)

Quand tous les Orques ont répondu à l'appel, vous (en tant que bon(ne) général d'Orque) les placez en ordre lexicographique (en gros : l'ordre alphabétique) de nom d'Orque, et vous comptez au passage le nombre de permutations que ce tri aura requis. Une permutation est une opération qui prend deux Orques et les intervertit (un *Swap*, en anglais).

Le programme principal, représentant ce que fait votre général, va comme suit :

```
using System;
using System.Collections.Generic;
List<Orque> orques = new();
try
{
   for(string s = Console.ReadLine(); "" != s; s = Console.ReadLine())
   {
      orques.Add(new Orque(s));
      Console.WriteLine($"Orque créé : {orques[orques.Count - 1].Nom}");
   }
}
catch(NomInvalideException nie)
{
   Console.WriteLine(nie.Message);
}
if(Trier(ref orques, out int nbPermutations))
   Console.WriteLine("Les orques ont été entrés en ordre alphabétique");
else
   Console.WriteLine($"Trier les orques a nécessité {nbPermutations} permutations");
Console.Write("La tribu d'orques est :");
foreach (Orque orque in orques)
   Console.Write($" {orque.Nom}");
```

Consigne particulière : votre fonction `Trier` doit faire un tri à bulles (un très mauvais tri, mais aussi très simple), dont l'algorithme général est :

```
Trier(tab)
   mainGauche ← 0
   Tant que mainGauche < tab.Length – 1
      mainDroite ← mainGauche + 1
      Tant que mainDroite < tab.Length
         Si tab[mainGauche] et tab[mainDroite] ne sont pas en ordre
           Permuter(tab[mainGauche], tab[mainDroite])
         ++mainDroite
      ++mainGauche
```

Il vous faudra donc écrire entre autres une fonction `Permuter` capable de permuter deux éléments de type `Orque`.

Vous devrez évidemment adapter cet algorithme aux besoins de ce programme et faire en sorte qu'en plus de trier les éléments de type `Orque`, il dépose dans le paramètre représentant le nombre de permutations le nombre de fois que `Permuter` aura été appelé (c'est à `Trier` de faire cette comptabilité, pas à `Permuter`) et retourne `true` seulement si aucune permutation ne fut requise, donc seulement si les éléments étaient déjà en ordre au moment de l'appel.

Pour réfléchir un peu…

Vous savez pertinemment que, dans un avenir rapproché, le général des Trolls, puis celui des Kobolds vont se pointer et vous donner leurs ordres. Comme programmeuse ou comme programmeur, vous anticipez ce moment, et vous comptez minimiser les modifications qui seront nécessaires ce jour-là.

Vous connaissez le concept de spécialisation, et vous voulez en profiter pour voir ce que ces types de créatures ont en commun, de même que ce qui les distingue. L'idée est à la fois d'avoir à la fois une *forte cohésion* (grouper ensemble ce qui va ensemble) et un *faible couplage* (faire en sorte que les changements apportés au code d'une classe aient un effet le plus local possible, pour en faciliter l'entretien).

Cela soulève la question des comportements : quels sont les comportements attendus d'un soldat, peu importe son espèce? Comment pourrait-on, par exemple, faire en sorte que le code suivant compile et donne un résultat cohérent?

```
Soldat [] soldats = new Soldat[]
{
   new Orque("Bill"), new Troll("Grr"), new Kobold("Piarkkk")
};
foreach(Soldat s in soldats)
   s.Saluer(); // chaque Soldat s se présente à sa façon
```

Votre tâche

Votre tâche pour cette activité formative est d'écrire la classe `Orque`, la fonction `Trier` et tout autre outil (classe, fonction) auxiliaire vous permettant d'en arriver à vos fins. Vous pouvez aussi vous amuser à ajouter les classes `Soldat`, `Kobold` et `Troll` suggérées à la section Pour réfléchir un peu… pour vous amuser.

L'objectif de cette activité n'est pas de réaliser le tout en un nombre minimal de fonctions ou en un nombre minimal de lignes; ce que nous visons est une réflexion sur la façon de programmer :

- Quelles sont les situations qui se modélisent mieux par une levée d'exceptions et quelles sont celles qui se représentent avantageusement autrement?
- Quelles sont les méthodes qui devraient être des méthodes de classe (`static`) et quelles sont celles qui devraient être des méthodes d'instances (non-`static`)? Notez que C# est limité, et que dans ce langage certains choix (parfois discutables) sont faits pour vous, mais dans d'autres cas, vous avez le choix et certaines options sont à privilégier
- En C#, quand devrait-on passer des paramètres par copie? Par référence (`ref` en C#)? En tant que sortants (`out` en C#)?
- Quelles classes auxiliaires pourrions-nous écrire pour nous aider? Quelles fonctions devrions-nous ajouter? Quelles devraient être leurs signatures?

Cette réflexion sur la pratique de la programmation (la *Praxis*) est au cœur de notre démarche dans ce cours. Nous reviendrons sur vos choix de design la semaine prochaine, et nous examinerons d'autres options.

Notez que cette activité est formative, pas sommative – elle ne compte pas au bulletin. Toutefois, il est *tout à fait* à votre avantage de vous y investir, car nous démarrerons bientôt la session sur les chapeaux de roues, et que nous programmerons *beaucoup*. L'accent sera mis sur la Praxis pendant toute la session, et réfléchir à vos façons de faire sera à votre avantage.

Amusez-vous bien!