Université de Bordeaux
Année 2018-2019

Mastère 2
T.A.P

TD n°7 - Contrôle de l'évaluation

▷ En C#, le mot-clé `yield` permet de transformer un calcul produisant des valeurs de type `T` en une valeur de type `Enumerable<T>` :

```
static IEnumerable<int> GetPrimes() {
    var index = 2;
    while (true) {
        if (IsPrime(index))
            yield return index;
        index++;
    }
}
```

```
foreach(int prime in GetPrimes()) {
    Console.WriteLine(prime);
    if (prime > 20) return;
}
// Displays "2 3 5 7 11 13 17 19 23"
```

L'expression `GetPrimes().GetEnumerator()` retourne une valeur de type `Enumerator<int>`, sur laquelle il est possible d'appeler les fonctions `Current` et `MoveNext()`. Cela permet de générer les valeurs de la liste *de manière paresseuse*. Noter qu'il est tout à fait possible d'itérer sur cette liste avec un `foreach`.

Exercice 1: Ordonnancement équitable

Supposons vouloir écrire un système d'ordonnancement équitable (*fair scheduling*), dans lequel un système d'exploitation alloue à un ensemble de threads du temps de calcul de manière à ce que chaque thread ait droit à une part équilibrée du temps. Dans cet exemple, un thread représente un code dont l'exécution est découpé en morceaux. Chacun de ces morceaux a un coût en temps entier. Le système ordonnance les threads et les laisse exécuter leur code par morceaux. Afin de choisir le thread à exécuter, il sélectionne celui ayant coûté le moins cher en temps de travail jusqu'à présent.

Un `Thread` contient un entier nommé *pid* (process id), et un entier *workload* comptant la quantité de travail déjà effectuée. Les morceaux de travail du thread sont représentés à l'aide d'une méthode `TaskLengths()` utilisant le mot-clé `yield` et renvoyant une valeur de type `IEnumerable<int>`. Le morceau de travail est censé être effectué entre deux appels à `yield`. Pour simplifier, le thread dispose d'une méthode `Step()` qui force le thread à effectuer un morceau de travail, et ajoute sa charge de travail au `Workload`.

1. Compléter la méthode `TaskLengths()` de `Thread` en utilisant le mot-clé `yield` et en renvoyant un `IEnumerable<int>` contenant au moins 10 valeurs entières tirées aléatoirement.
   Remarque : il est recommandé d'effectuer un affichage avant chaque `yield` afin de visualiser à quel moment la fonction est appelée.

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Thread {
    public int Workload {get; private set;}
    public int Pid      {get; private set;}
    private IEnumerator<int> Worker;  // An enumerator of the tasks of the
                                      // thread (represented by their lengths)
    static Random rng = new Random(); // Number generator in the class
    public Thread() {
        Workload = 0;
        Pid      = rng.Next() % 100 + 100;  // Random pid number
        Worker   = this.TaskLengths().GetEnumerator();
    }
    public void Step() { // Executes one task of the thread
        Workload += Worker.Current;
        Worker.MoveNext();
    }
    IEnumerable<int> TaskLengths() {
       // To be filled
        yield return 1;
    }
}

class Program {
    public static void Main() {
        Console.WriteLine("Yield !");
        var ListOfThreads = new List<Thread>();
        ListOfThreads.Add(new Thread());
        ListOfThreads.First().Step();
    }
}
```

2. Écrire le code nécessaire pour créer une liste de 10 threads et les faire travailler dans l'ordre chacun 10 fois.
3. Écrire le code ordonnançant ces threads en considérant qu'à tout moment, le thread qui peut travailler est celui dont la charge est la plus petite.

Exercice 2: Java Standard Widget Toolkit

La bibliothèque SWT (Standard Widget Toolkit, cf. `https://www.eclipse.org/swt/`) est une bibliothèque maintenue par Eclipse permettant de réaliser des interfaces graphiques en Java. La documentation de cette bibliothèque est accessible à l'adresse `https://www.eclipse.org/swt/javadoc.php`.

Considérons le problème d'implémenter un équivalent du *Game of Life* de Conway avec ces widgets graphiques. Pour cela, considérons les cellules de ce jeu comme représentées par la classe `Cell` suivante :

```java
class Cell {
    private Button button;        // Widget associated to the Cell
    private List<Cell> neighbors; // Neighboring cells
    static Random rng = new Random();

    public Cell(Composite shell) {
        button    = new Button(shell, SWT.CHECK);
        neighbors = new ArrayList<Cell>();
        this.setVisible(rng.nextInt(2) == 0);
    }
    public void setVisible(boolean b) {
        button.setSelection(b);
    }
    public boolean getVisible() {
        return button.getSelection();
    }
    public void addNeighbor(Cell c) {
        this.neighbors.add(c);
    }
    public int livingNeighbors() {
        return 0; // Fill here ...
    }
}
```

Le code fourni construit une classe `Board` construisant un tableau à deux dimensions de `Cell` et initialisant les voisins de chaque cellule. Dans les questions qui suivent, le code doit utiliser les méthodes situées dans la classe `Stream<T>` (cf. `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`).

1. Écrire dans la classe `Cell` une méthode `livingNeighbors()` qui calcule le nombre de voisins visibles.
2. Écrire dans le constructeur de la classe `Board` le code construisant la liste `cells` de toutes les `Cell`.
3. Écrire dans la classe `Board` une méthode `evolve()` qui effectue une étape du Game of Life et change la visibilité des cellules en fonction.

Supposons vouloir créer un thread qui mette à jour l'interface graphique à chaque seconde en la faisant évoluer avec `evolve()`. L'affichage peut se faire à l'intérieur d'un `Thread` à l'aide de la méthode `display.asyncExec()`, qui prend en paramètre un `Runnable` (que l'on peut passer sous la forme d'une lambda-expression).

4. Écrire le thread en question en utilisant uniquement des lambda-expressions Java.
   Note : ce thread doit être démarré juste avant la boucle d'événements de SWT.