# Systèmes d'acquisition - Partie "Java"

S. Reynal

2018-2019

Cette série de TP Java a pour objectif de vous initier au développement rigoureux d'applications, et en particulier d'IHM, dans le domaine de l'acquisition et de l'instrumentation. Seront abordés :

- la modularité, autour du paradigme model-view-controller
- la réalisation d'IHM avec le framework JavaFX
- les sockets et la communication en java sur un réseau
- la parallélisation avec les Threads

Il y a beaucoup de choses à réaliser : de fait, si vous êtes à cours de temps, vous pouvez sauter certaines parties et choisir celle pour laquelle vous avez le plus de compétences à acquérir.

## 1 Cahier des charges

Il vous est proposé de développer un oscilloscope virtuel en java, qui permet d'afficher des signaux provenant d'un capteur distant via un module XBee (cf. partie "XBee"). Ce cahier des charges vous donne quelques contraintes de développement de votre application. Celles-ci étant volontairement succinctes, il s'avère que ce qui est imposé dans la suite doit impérativement être respecté, et qu'a contrario tout ce qui n'est pas imposé constitue votre marge de liberté et de créativité.

Les classes seront organisées en trois packages :

- **model** : classes dédiées à la représentation et aux stockage des signaux ayant précédemment fait l'objet d'une acquisition. Dans un premier temps, on chargera les données depuis un fichier stocké sur le disque dur ;
- **view** : classes dédiées à la visualisation des signaux ; ici on trouvera les éléments habituels d'un panneau avant d'oscilloscope : écran, grilles, axes, boutons de réglage de calibre, etc.
- **controller** : classes dédiées aux évènements de contrôle du panneau avant de l'oscilloscope, c'est-à-dire gestion des évènements souris et clavier.

En outre, on créera une classe principale Oscilloscope : elle doit hériter de `javafx.application.Application`, implémente trois méthodes init(), start() et stop() (surchargées depuis la superclasse) qui sont destinées à initialiser toutes les autres classes de l'application (model, view et controller), ou bien (pour stop) à en libérer les ressources le cas échéant. Oscilloscope implémente également une méthode main() qui contient elle-même la méthode launch() lançant l'application JavaFX proprement dite. "init()" est plutôt destinée à initialiser ce qui ne relève pas du graphisme JavaFX (dans notre cas, ce seront par ex le port série sur lequel est connecté le XBee "coordinateur"), tandis que "start()" est habituellement chargée d'instancier les différents composants graphiques JavaFX.

# 2 package "model"

Ce package contient toutes les classes destinées à modéliser les données à acquérir. Il est totalement indépendant de l'interface graphique utilisée, et doit pouvoir être testé indépendamment.

## 2.1 Signal

- Représente un signal composé d'échantillons de type "double".
- Fields : tableau d'échantillon, fréquence d'échantillonage, nombre de bits de quantification, unité physique.
- Constructeurs : un signal peut être initialisé sous forme de signal prédéfini (prévoir une **enum** contenant les constantes NOISE, SINE, SQUARE) ou à partir d'un fichier sur le disque dur au format matlab binaire
- méthodes : outre les getters et setters habituel, la classe signal permet d'ajouter ou d'enlever des échantillons à la volée.

## 2.2 SignalTools

- Classe offrant des outils de traitement de signal (mean, variance, FFT)
- Initialisée à partir d'un Signal fournit en argument, d'un nom de fichier, ou d'un tableau de "doubles"

# 3 package "view"

Toutes les classes correspondant à des composants JavaFX héritent au moins de Node, voire un de ses descendants. Le package view contient une classe principale "FrontPane" qui représente le panneau avant de l'oscillocope et qui hérite

de BorderPane. Cette classe est un container pour les classes Screen (l'écran, à gauche) et ControlPane (les boutons, à droite) ci-dessous. Pour Screen, on pourra utiliser avec profit la classe `javafx.scene.chart.Chart` qui sait parfaitement dessiner des signaux sur un écran... Mon tutorial répertorie les autres classes JavaFX utiles.

### 3.1 Screen

- Représente un écran d'oscilloscope permettant d'afficher jusqu'à quatre signaux avec des couleurs différentes ;
- Implémente au moins une méthode permettra d'ajouter de nouveaux signaux ou d'enlever des anciens
- utilise la classe `javafx.scene.chart.Chart`.

### 3.2 Grid

- Représente une grille graduée ; c'est ce qu'on appelle un helper class (indice : est-elle public ou package du coup ?) car elle sert uniquement à Screen
- Fields : le pas de la grille (vertical et horizontal) ; la couleur de la grille ; la présence ou non de labels et de graduation
- Indication : utiliser l'objet Line et une directive de style CSS pour ajouter des flèches, des pointillés, etc.

### 3.3 ControlPane

- Représente le panneau de contrôle de l'oscilloscope (boutons, potentiomètres, etc)
- Le traitement des évènements souris ou clavier reçus par cette classe est géré par une classe héritée de EventHandler (cf. la classe `javafx.scene.input.MouseEve` et située dans le package controller.

## 4 package "controller"

### 4.1 EventHandler

- Il s'agit d'une classe gérant les évènements souris (aka `javafx.scene.input.MouseEver` ou clavier (KeyEvent)
- Screen délègue à cette classe le soin de gérer ces évènements

# 5 sub-package "model.network", Threads et capteurs distants dans le cloud

Cette partie permet de s'initier aux sockets et aux Threads. L'objectif est de simuler la réception par un serveur (la machine sur laquelle vous développez) de données (température, etc.) émises par plusieurs capteurs intelligents connectés au serveur via le réseau (ethernet ou WiFi, par ex des xbee wifi). Ces données sont un autre moyen d'initialiser un objet Signal, donc.

- Commencer par implémenter la classe "Sensor" qui hérite de la classe "Thread". Cette classe se contente pour l'instant, dés que le thread est démarré, d'afficher régulièrement, chaque seconde, une donnée de mesure factice (un nombre aléatoire) à l'écran (via la console). Le code correspondant se trouvera dans la méthode "run()", et repose sur une boucle infinie et l'appel de la méthode "sleep()" (attention à bien gérer les exceptions associées). Les threads seront lancés depuis la méthode "main()". L'affichage souhaité pour chaque Sensor est à titre indicatif du type : *"Capteur numéro 4 : température = 28C"*.Dans un second temps, on créera un objet Signal à partir de ces data.
- Implémenter maintenant la classe Serveur. Cette classe hérite de ServerSocket et est chargé d'attendre les connections émanant de clients (les capteurs). Elle repose sur l'appel de la méthode "accept()" de la classe ServerSocket, qui renvoie un Socket représentant la connection active. Une fois la connection activée, récupérer le InputStream associé à la connection via la méthode ad hoc dans Socket, lire une donnée (un double) depuis la connection, fermer la connection, et passer au client suivant. Pour l'instant, le serveur ne peut gérer qu'une connection active avec un client à la fois. On devra utiliser la classe DataInputStream (paquet java.io).
- Le code n'est véritablement fonctionnel que s'il existe des clients ! Ajouter à la classe Sensor dans la méthode run(), la connection au serveur (via la classe Socket). Une fois la connection acceptée par le serveur, récupérer le OutputStream associé (via la méthode getOutputStream de la classe Socket), et écrire un double correspondant à la température.
- Ajouter à la classe Serveur la possibilité de gérer plusieurs connection simultanément, via le multithreading. Pour cela, il faut insérer le code qui attend un client (la méthode accept()), à l'intérieur de la méthode run() d'un thread (ou implémenter la méthode run() de l'interface Runnable, autre approche), puis lancer un nouveau thread à chaque nouveau client. Ensuite, il suffit de laisser les threads tourner en boucle infinie et communiquer avec leur capteur attaché.

Remarque : vous pouvez travailler en équipe, un binôme développant la classe

Sensor, un autre la classe Serveur. Ainsi les "faux" capteurs peuvent être disséminés sur plusieurs machines de la salle et émettre leur données pour le reste du groupe !

# 6 sub-package "model.xbee"

Cette partie est destinée à gérer l'acquisition de données distances via un module XBee. On utilisera pour cela la bibliothèque d'Andrew Rapp (tellement plus fiable que la bibliothèque officielle éditée par l'entreprise Digi — fabriquant du xbee — elle-même...). Elle est disponible sur Github : `https://github.com/andrewrapp/xbee-api`. Le package contiendra une classe XbeeListener qui implémente PacketListener et est chargée de dispatcher les informations envoyées par le module XBee vers l'interface graphique.