Les threads en javaFX

10 février 2021

Les threads de JavaFX

Par défaut dans une application javaFX, il existe plusieurs threads :

- Prism render thread :
  calcule le rendu $n+1$, pendant que la frame n est affichée.
- Media thread
- JavaFX Application Thread :
  construit l'IHM, gère le scène graph, dessins, évènements, callbacks.
  Je vais dire souvent JaFXAT *pour faire plus court*.
- `main` s'il y a lieu.

Remarque :
Tout ce qui modifie l'interface (par exemple afficher un message dans un label du graphe de scene) **doit être exécuté** dans le *JaFXAT*

Les threads de JavaFX :

Le *JaFXAT* s'occupe de :

- gérer les événements :
  - gère la file d'attente des évènements.
  - exécute le code des "handler" correspondant à chaque événement, l'un après l'autre.

  Cela comprend les évènements gérés explicitement (par nous même), mais aussi la réaction de l'interface aux menus, etc
- la construction de la fenêtre et de l'IHM en général : `add(..)`, `setLayoutX`, `setWidth`... sont exécutés dans JaFXAT

Les threads de JavaFX : blocage de l'EDT ?

Souci :

Temps de calcul long, chargement de fichier, calcul d'image pour un fond d'écran, connection à une base de donnée, fait dans un handler ?

$\Rightarrow$

- lorsqu'un handler prend du temps, cela bloque l'interface.
- si lors de la construction d'une fenêtre, une manip prend du temps, cela bloque l'interface.

Exemple : TestFigeage.java

Les threads de JavaFX : blocage de l'EDT ?

Solution :

1) Déporter les gros calculs dans un thread séparé.

2) Si besoin, lors du calcul, demander à l'interface (i.e. JaFXAT) de montrer l'avancement du calcul

3) Si besoin, à la fin du calcul, demander à l'interface (i.e. JaFXAT) de prendre en compte le changement : changement d'étiquette, dessin, etc...

Les threads de JavaFX : blocage de l'EDT ?

On peut faire cela en utilisant :

- les outils standards de Java :
  Interface `Runnable` et classe `Thread`.
- les outils spécialisés de JavaFX :
  `Worker`, `Service`, `Task`

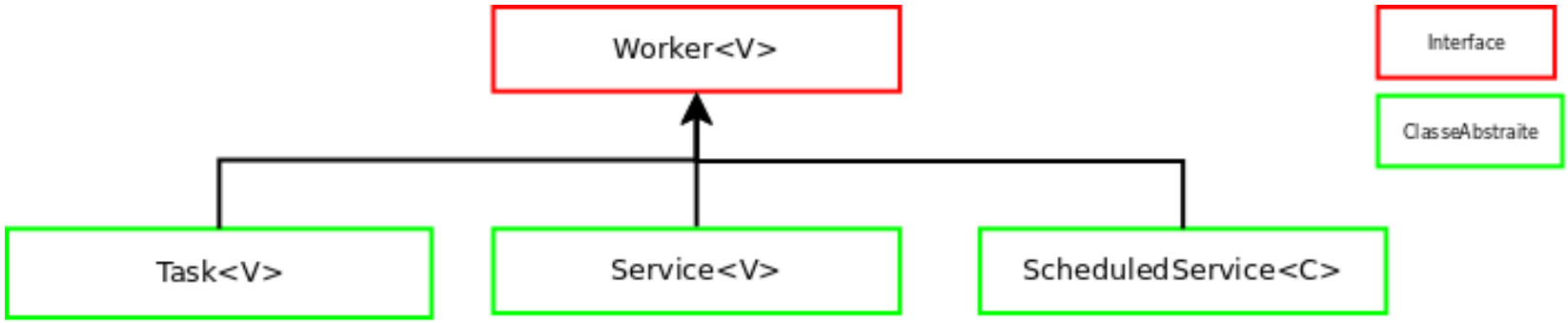
On voit ici les outils spécialisés de JavaFX.

Pour `Runnable` et `Thread`, se reporter au cours de P.O.O.

Worker and sons

Voici l'interface Worker et les classes abstraites qui l'implémentent.

`javafx.concurrent.`**Worker**



- `Worker` : l'interface avec toutes les méthodes et outils fondamentaux
- et les classes permettant d'exécuter :
  - `Task` : une tâche *une seule fois*
  - `Service` : une tâche *plusieurs fois*
  - `ScheduledService` : une tâche *à intervalles réguliers*.

Worker and sons

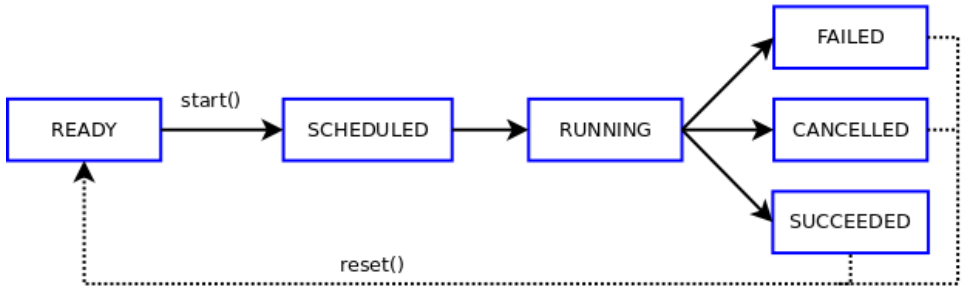
L'interface Worker<V>

- méthode abstraite V call(), contiendra le code que doit exécuter la tâche.
- `V` est le type de retour de la méthode `call()`.

`Void` si la tache ne calcule rien (`null` est la seule valeur possible pour un `Void`).

Worker contient des **property**, entre autres :

- `state` : l'état et l'évolution d'une tâche ou d'un service.



Worker and sons

- `value` : la valeur calculée par la tâche,

`running` : la tâche est en train de s'executer,

- `totalWork`, `progress`, `workDone` permettant de gérer l'avancement de la tâche

$\Rightarrow$ possibilité de binder ces propriétés et de mettre des Listener dessus.
Cela permet de surveiller/réagir à l'évolution du Worker.

- Et des méthodes, en particulier :
  - `getState()`, `isRunning()`
  - `cancel()`
  - `getValue()`

Task

La classe abstraite Task contient les méthodes et propriétés pour gérer :

- les événements :
  `addEventFilter`, `addEventHandler`, `setOnXXX`...,
- la communication avec le *JAFXAT* : en modifiant `valueProperty`, `progressProperty` et `messageProperty` de la Task
  - `updateValue(V value)`,
  - `updateProgress(workdone, max)`,
  - `updateMessage(String message)`

  Il faut écrire dans l'IHM (dans le JAFXAT) les instructions qui écoutent ces property .
- l'annulation :
  cancel(), `iscancelled()`

  Attention : `cancel()` n'arrête pas la tâche, c'est un simple signalement. À nous d'écrire dans `call()` les appels à `iscancelled()` et terminer s'il faut.

Task : utilisation de base

1. Écrire la tâche

```
Task<Void > calculateTask = new Task<Void>(){
    protected Void call() throws Exception {
        ...  le code long à executer
        return null ;
    }
};
```

2. Déclarer un thread avec cette tache

```
Thread th = new Thread(calculateTask) ;
```

3. Le lancer le thread dans le `handle`

```
th.start();
```

Task : traitement fin de tâche

Réagir à la fin de la tâche ?

- Abonnement à un ChangeListener (≈ *écouteur de Changement*) pour traiter tous les cas de terminaison possible.

```
calculateTask.stateProperty().addListener(new ChangeListener () {
    @Override
    public void changed(ObservableValue observable,
                        Worker.State oldValue,
                        Worker.State newValue) {
        switch (newValue) {
            case FAILED:
            case CANCELLED: System.out.println("on m'a annulé");
            case SUCCEEDED: scene.setCursor(oldCursor);
                calculateItem.setDisable(false); calculateButton.setDisable(false);
                message.setText("-> J'ai fini "); break;
                } }
});
```

- ou avec `setOnXX(EventHandler e)` pour chaque cas de terminaison.

```
calculateTask.setOnSucceeded(EventHandler<WorkStateEvent> eh)
```

Remarque : ceci est fait appelé dans le `JaFXAT`

Task : communication avec l'IHM

Interdit depuis un autre thread de manipuler l'interface.

Si `labM` est un Label de l'IHM, `labM.setText("à plus");` lève une erreur.

Deux solutions :

1) Utiliser les property
`messageProperty`, `valueProperty`, `progressProperty`, `totalWorkProperty`

- *Dans le* `call()` *de la Task* `maTache`
  ```
  updateMessage("Bonjour j'ai presque fini");
  updateProgress(80, 120); // J'ai traité 80 sur 120.
  ```
- *Dans l'interface*
  - bind avec le label contenant le message.
    ```
    labM.textProperty().bind(maTache.messageProperty(),
    ```
  - ou avec un `Listener`
  - ou en appelant la methode `getMessage()` au moment désiré dans l'IHM
    ```
    labM.setText(maTache.getMessage())
    ```

Task : communication avec l'IHM

2) Utiliser un appel à `PlatForm.runLater(Runnable r);`

Dans le `call()` *de la Task* `maTache`

- On crée un `Runnable r`,
  dont la méthode `run()` contient le code à exécuter dans le *JaFXAT*
- Dans le `call()` de la Task `maTache`

```
Platform.runLater(r);
```

Exemple avec expression-lambda :

```
Platform.runLater(()-> { message.setText("j'ai presque fini");})
```

Task : est-on dans le JaFXAT ?

On peut écrire du code, ou une méthode qui sera utilisée à plusieurs endroits :
– dans une `Task`
– dans le corps de l'IHM par le JaFXAT.

Problème : on vient de voir qu'on doit avoir un traitement différencié...

`boolean Platform.isFxApplicationThread()`
indique si on est dans le JaFXAT.

Task : Annulation de la tache

Attention, la méthode cancel ne suffit pas, ce n'est qu'une demande.
$\Rightarrow$ la tâche doit tester et traiter la demande.

Soit une Task maTache

- *dans l'IHM (le JaFXAT)*

```
maTache.cancel()
```

- *dans le* `call()` *de* `maTache`

```
if (isCancelled()) {
   instructions pour terminer le calcul
}
```

Service

Une tâche ne peut être lancée qu'une fois.

alors un bouton ne marche qu'une seule fois ?

Un Service crée une nouvelle tache à chaque lancement.
L'essentiel du fonctionnement est le même :

- les différents états du service :
  mais on peut revenir à l'état `READY` par un `reset()`, ou `restart()`
- les écouteurs de changement d'état :
  mais on peut écouter le changement vers `READY`
- communication avec le JaFXAT
  – par les `Property` du `Service` et les méthodes `updateXXXX`
  – par l'appel à `Platform.runLater()`
- gestion de l'annulation par `cancel()` et `isCancelled`

Tout passe par le service qui gère la tâche sous-jacente.

Service : Utilisation

1) Écriture du service.

Il contient une méthode **createTask** chargée de créer une nouvelle tâche à chaque lancement.

```
Service<Void> calculateService = new Service<Void>(){

  @Override
     protected Task<Void> createTask(){

       return new Task<Void>(){
             @Override
             protected Void call() throws Exception {
               //
               // .. le calcul long..
               //
               return null ;
             }
       };
     }
};
```

Service : Utilisation

2) Lancement du service

par exemple dans le `handle` de l'écouteur.

`calculateService.start()` crée la tâche et la lance.

3) une fois terminé, ne pas oublier :

`calculateService.reset()` reinitialise le service

Remarque : `restart()` = `{ reset(); start();}`

ScheduledService:

ScheduledService

permet d'avoir des tâches répétitives, à intervalles programmés.

On dispose en particulier des méthodes :

- `service.setDelay(Duration.seconds(50));`
  délai d'attente avant le démarrage du service
- `service.setPeriod(Duration.seconds(10));`
  temps d'attente entre deux exécutions sans erreur
- traitement des échecs :
  - `service.setRestartOnFailure(true);`
    on redémarre si il y a echec (par défaut, on arrête le service)
  - `service.setMaxFailureCount(100);`
    nombre d'échecs maximum.