

Interfaces Graphiques

« La concurrence est rude mais nécessaire,
nul ne doit y échapper... »

Proverbe ultra-libéral

Jean-Baptiste.Yunes@univ-paris-diderot.fr

Université Paris Diderot

©2015



- Swing n'est pas thread-safe!!!
- pour en savoir plus, lisez « Swing's Threading Policy »
- In general **Swing is not thread safe**. All Swing components and related classes, unless otherwise documented, must be accessed on the event dispatching thread.

- Réactivité
- Concurrency
 - threads *initiaux*
 - thread principal de l'interface (distribution)
 - threads annexes de travail
- Ces threads sont automatiquement créés...

- thread principal (event dispatching thread)
 - contrôle la distribution des événements
 - **exécute sous son contrôle** le code de traitement associé aux événements (réaction)
 - **exécute sous son contrôle** de nombreuses sollicitations externes (commandes)

- interface réactive
 - à réception d'événements le thread principal ne doit pas exécuter de code trop long (en temps)
 - sinon, il faut utiliser des threads de travail

- Threads initiaux
 - Application
 - un seul thread appelant `main()`
 - Applets
 - éventuellement plusieurs threads appelant
 - `init()`
 - `start()`

- dans ces threads initiaux
 - on initialise l'application
- l'initialisation de l'interface doit être réalisée sous le contrôle du thread principal
 - créer un Runnable
 - demander à l'exécuter par le thread principal
 - `SwingUtilities.invokeLater(Runnable)`
 - `SwingUtilities.invokeAndWait(Runnable)`
 - une simple encapsulation des méthodes correspondantes de `java.awt.EventQueue`

- Attention pour une `Applet`
- la création de l'interface doit s'effectuer dans la méthode `init`
- elle doit être terminée avant le retour de celle-ci
- nécessaire d'utiliser `invokeAndWait`

- Threads de travail
 - s'occupent de réaliser des tâches longues
 - sont en arrière-plan
 - une telle tâche peut être représentée par une instance de `SwingWorker`

- mécanismes
 - exécution d'une méthode à la terminaison
 - fournit un résultat asynchrone (un futur)
 - peut exécuter du code sous contrôle du thread principal

- Comment intégrer correctement une longue tâche à une interface graphique ?
- Rappel : les `Listeners` doivent répondre rapidement
 - nécessité de passer par des threads annexes
 - la classe `SwingWorker` fournit tous les services utiles et la bonne abstraction

- Encapsule :
 - un thread de travail
 - des mécanismes de synchronisation vis-à-vis de l'exécution de la tâche
 - des mécanismes de synchronisation avec le thread d'interface

- la création du thread de travail est obtenue par
 - `execute()`
 - attention cette méthode ne peut être employée deux fois!
 - un thread ne s'exécute qu'une seule fois
- on peut obtenir le résultat du calcul par appel à
 - `T get()`
 - `T` est le type générique de la valeur de retour du `SwingWorker<T, V>`

- le calcul du thread annexe est implanté dans la méthode
 - `T doInBackground ()`
 - `T` est le type générique de la valeur de retour du calcul (`SwingWorker<T, V>`)
- lorsque `doInBackground ()` termine, la méthode
 - `void done ()` est appelée dans le contexte du thread de l'interface
 - on y place habituellement de quoi *nettoyer* l'interface

- `doInBackground()` peut avoir besoin d'effectuer mises-à-jour de l'interface, donc sous le contrôle du thread d'interface
- `publish(V... valeurs)` exécutée habituellement dans le thread de calcul
- permet d'envoyer des valeurs de type générique `V` (`SwingWorker<T, V>`) à la méthode
- `process(V... valeurs)` exécutée normalement dans le thread de l'interface
- permet de récupérer les valeurs envoyées par le thread de calcul

- `ProgressBarExample.java`
- un exemple illustrant l'usage d'un `SwingWorker` pour
 - obtenir la réalisation d'une tâche
 - l'affichage de sa progression dans l'interface