Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

11

Swing : Timers & Threads

Jacques Bapst

jacques.bapst@hefr.ch

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Utilisation de Threads avec Swing

- La conception de la librairie Swing impose que les instructions qui accèdent et manipulent des composants de l'interface graphique soient effectuées dans un seul *Thread*.
- Autrement dit, la librairie **Swing n'est pas *Thread-Safe*** (à l'exception de quelques rares méthodes mentionnées plus loin).
- C'est par souci d'efficacité et de simplicité que ce modèle de conception a été utilisé pour l'élaboration de la librairie Swing.
  Le coût de la gestion des verrous *(locks)* a été jugé prohibitif en terme de performance et de complexité.
- Le *Thread* dans lequel s'exécute le code qui gère l'interface graphique est appelé ***Event Dispatch Thread*** ou ***EDT*** (ce n'est pas le *Thread* principal de l'application).
- Ce *Event Dispatch Thread* est démarré automatiquement lorsque l'on utilise des composants de la librairie Swing.

Swing Event-Dispatch Model

Operating System

AWT Event Queue

Mouse Press
Mouse Move
Key Press
. . .
Mouse Release
Mouse Move
Mouse Press

Critical thread

Time-consuming code freeze the GUI !

Event Listeners

EDT

AWT / Swing Repainting

Event Dispatch Thread

Règles à suivre

- La librairie Swing impose la règle suivante :

> **Dès qu'un composant a été rendu visible *(realized)*, tout le code qui dépend ou qui pourrait affecter l'état du composant doit être exécuté dans le *Event Dispatch Thread (EDT).***

- Un conteneur de premier niveau *(JFrame, JDialog, JApplet, JWindow)* est rendu visible *(realized)* dès qu'une des méthodes suivantes a été invoquée : `setVisible(true)`, `pack()` ou `show()` *[deprecated]*.
- Cette règle permet cependant, dans la plupart des cas, de construire l'interface graphique dans le *Thread* principal avant de la rendre visible.
- Il est donc possible (même si *Sun* le déconseille maintenant) de créer les composants visuels, de les configurer et de les placer dans des conteneurs dans le cadre du *Thread* principal de l'application mais ensuite (après l'invocation de l'une des méthodes `setVisible(true)` ou `pack()`), toutes les instructions qui accèdent ou manipulent les composants Swing doivent être exécutées dans le *EDT* (sous peine d'effets imprévisibles).

Event Dispatch Thread (EDT)

- Le ***Event Dispatch Thread*** *(**EDT**)* est le *Thread* qui exécute les instructions de dessin *(Drawing)* des composants et qui se charge de la gestion des événements *(Event Handling*) liés à ces composants.
- Toutes les instructions de **dessin** et de **réaffichage** ainsi que celles qui se trouvent dans les **récepteurs d'événements** *(Event Listener)* sont implicitement exécutées dans le *EDT*.
- L'utilitaire représenté par la classe **`javax.swing.Timer`** permet également d'exécuter des instructions (après un certain délai ou à intervalles périodiques) dans le *EDT*.
- Deux méthodes statiques de la classe **`EventQueue`** permettent également d'insérer des instructions dans le *EDT* :

  `invokeLater()`

  `invokeAndWait()`

- Ces deux méthodes prennent pour argument un objet qui implémente l'interface **`Runnable`**.

Timer [1]

- La classe **`javax.swing.Timer`** permet d'exécuter automatiquement certaines instructions sur la base d'une horloge (temporisateur).
- Un timer peut être utilisé pour :
  - **Exécuter** une tâche **après un certain délai**
  - **Répéter à intervalle périodique** une certaine tâche
- Attention : Ne pas confondre la classe **`javax.swing.Timer`** avec la classe plus générale **`java.util.Timer`**. La première doit être utilisée de préférence pour des activités liées à une interface utilisateur (GUI).
- La classe **`Timer`** permet d'effectuer des animations ou des mises à jour d'informations permettant de refléter la progression d'une activité.
- Les instructions exécutées par la classe **`Timer`** s'exécutent dans le *EDT* et peuvent donc accéder et modifier sans risque les propriétés des composants affichés.
- Pour ne pas perturber la réactivité de l'application, les instructions effectuées par la classe **`Timer`** **doivent s'exécuter rapidement** (on risque sinon de figer l'interface utilisateur).

Timer [2]

- Le principe de la classe **`Timer`** est de générer à intervalles réguliers des événements de type **`ActionEvent`** qui seront traités dans un gestionnaire de type **`ActionListener`**.
- L'objet qui implémente **`ActionListener`** ainsi que l'intervalle entre deux exécutions (en [ms]) sont passés en paramètre du constructeur de la classe **`Timer`** .

```
// La classe MyController doit implémenter ActionListener et donc déclarer
// une méthode actionPerformed()
MyController c = new MyController();

// Création d'un timer qui exécutera la méthode actionPerformed()
// toutes les deux secondes
Timer horloge = new Timer(2000, c);   // Création du Timer (2 s)
horloge.setInitialDelay(500);         // 0.5 s avant 1ère exécution
horloge.start();                      // Démarrage du Timer
```

Timer [3]

- La classe **`Timer`** possède les **propriétés** principales suivantes :

initialDelay	Durée d'attente (en [ms]) entre le démarrage du timer et la première exécution Par défaut : identique à delay
delay	Intervalle entre deux exécutions (en [ms])
repeats	**`true`** si l'exécution doit se répéter périodiquement **`false`** si l'exécution est unique Par défaut : **`true`**
coalesce	Indication (**`true`** / **`false`**) si le système doit grouper des exécutions multiples en attente (et n'exécuter qu'une seule fois l'action pour tous les événements en attente) Cela peut se produire si des événements n'ont pas pu être traités à temps en raison de la charge du système Par défaut : **`true`**
actionCommand	Définit la chaîne de caractères associée à la propriété *actionCommand* de l'évènement du timer

Timer [4]

- La classe **`Timer`** possède les **méthodes** principales suivantes :

start()	Démarre le timer après le délai initial *(initialDelay)*
restart()	Annule toutes les actions en attente et redémarre le timer après le délai initial *(initialDelay)*
stop()	Annule toutes les actions en attente et stoppe le timer
isRunning()	Retourne **`true`** si le timer est démarré

- Il est possible de **modifier la fréquence d'exécution** même si le *timer* est en train de tourner (en utilisant la méthode **`setDelay()`**).
- La classe **`javax.swing.Timer`** peut également être utilisée pour temporiser des activités qui n'ont rien à voir avec l'interface utilisateur.
- La classe **`java.util.Timer`** est un peu plus générale (elle contient plus de fonctionnalités) mais les activités ne se déroulent pas automatiquement dans le *EDT*. Elle est donc un peu plus complexe à utiliser correctement en relation avec une interface utilisateur.

Swing et les Threads [1]

- On l'a vu, les méthodes de la librairie Swing ne sont pas *Thread-Safe* et ne doivent pas être invoquées en dehors du *EDT*.
- Il y a cependant quelques exceptions et les méthodes suivantes peuvent être utilisées depuis n'importe quel *Thread* de l'application :

  `add`*`event`***`Listener()`**
  `remove`*`event`***`Listener()`**
  `repaint()`
  `invalidate()` (rarement nécessaire, appelé automatiquement)
  `revalidate()` (rarement nécessaire, appelé automatiquement)

  (Il y en a encore quelques autres qui sont mentionnées dans la documentation)

- Lorsqu'une interface graphique a été construite et affichée, la plupart des applications réagissent aux événements par invocation du code inséré dans les récepteurs d'événements *(Event Listener)*.
- Ces instructions ne posent pas de problèmes particuliers car elles sont implicitement exécutées dans le *EDT*.

Swing et les Threads [2]

- Il y a cependant des situations où l'exécution d'instructions liées à l'interface graphique ne peut pas s'effectuer dans les *Event Listeners*.
- Par exemple :
  - Si certaines instructions durent longtemps, elles ne devraient pas être insérées dans des *Event Listener* car elles bloqueront toute la gestion de l'interface graphique qui ne réagira plus aux sollicitations de l'utilisateur.
  - Si la mise à jour de l'interface graphique résulte d'un événement qui n'est pas lié à un composant visuel aucun gestionnaire d'événement ne pourra être enregistré pour le traiter (par exemple si l'événement est provoqué par un signal provenant d'une autre machine, par la synchronisation avec un autre processus ou par un signal provenant d'un périphérique particulier).
- Dans certaines situations, il peut donc être indispensable de créer un ou plusieurs *Threads* (pour effectuer les opérations qui prennent beaucoup de temps) et de mettre à jour l'interface graphique en invoquant l'une des deux méthodes utilitaires **`invokeLater()`** ou éventuellement **`invokeAndWait()`** qui permettront d'insérer les instructions nécessaires dans le *EDT*.

Swing et les Threads [3]

- En résumé, il y a **deux contraintes importantes à respecter** pour garantir la réactivité de l'interface graphique :

  - **Toutes les tâches qui prennent du temps doivent être exécutées en dehors du *EDT*.**
  - **Les composants de l'interface graphique ne doivent être accédés que depuis le *EDT*.**

- Il y a principalement trois types de *threads* à considérer dans une application comportant une interface-utilisateur graphique.
  - Le ***Thread* principal** *(Main Thread)* qui exécute les instructions au lancement de l'application
  - Le ***Event Dispatch Thread*** qui exécute le code de tous les gestionnaires d'événement *(Event Listeners)*
  - Les ***Worker Threads*** *(*ou *Background Threads)* qui sont les *threads* dans lesquels sont exécutés les tâches qui prennent du temps (ou qui sont en attente d'un signal asynchrone quelconque)

Utilisation de `invokeLater()`

- La méthode **`invokeLater()`** peut être invoquée depuis n'importe quel *Thread* pour exécuter certaines instructions dans le *EDT*.
- Le code à exécuter doit être inséré dans la méthode **`run()`** d'un objet qui implémente l'interface **`Runnable`**.
- Lors de son invocation, la méthode **`invokeLater()`** retourne immédiatement, sans attendre que les instructions de la méthode **`run()`** aient été exécutées dans le *EDT*.
- Exemple :

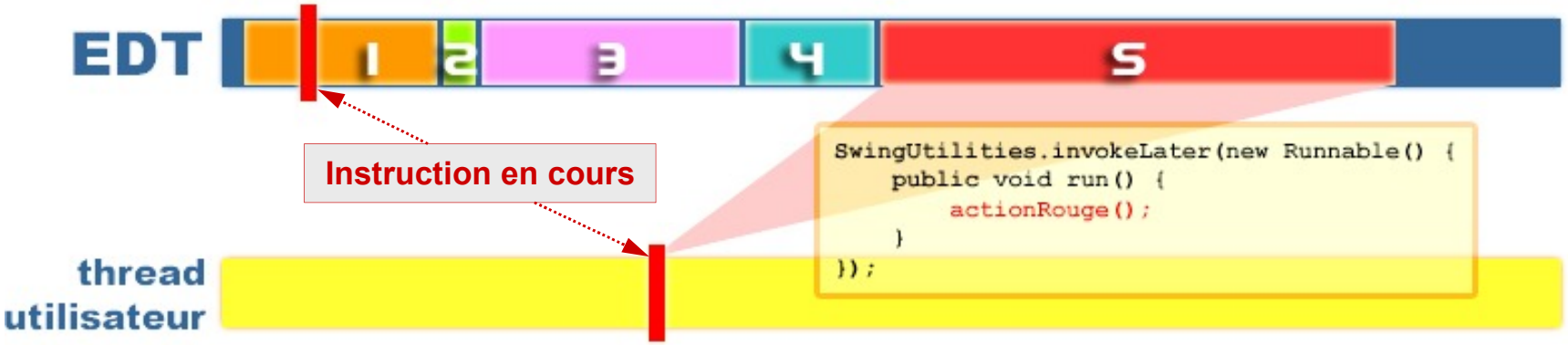
```
Runnable doWorkRunnable = new Runnable() {

    public void run() {
      doWork();       // Code à exécuter dans le Event Dispatch Thread
    }
};

EventQueue.invokeLater(doWorkRunnable);
```

Séquencement des Threads

- L'objet **`Runnable`** s'exécutera de manière asynchrone par rapport au *Thread* utilisateur (*Main Thread* par exemple).
- Séquencement avec la méthode **`invokeLater()`** :



- Si l'on veut impérativement attendre la terminaison de l'exécution de l'objet **`Runnable`** avant de continuer, il faut utiliser la méthode **`invokeAndWait()`** qui est bloquante.

Utilisation de `invokeAndWait()`

- La méthode **`invokeAndWait()`** s'utilise de la même manière que **`invokeLater()`** en lui passant un objet qui implémente l'interface **`Runnable`**.
- La seule différence est que lors de son invocation, la méthode **`invokeAndWait()`** attend que les instructions de la méthode **`run()`** aient été exécutées dans le *EDT* avant de retourner le contrôle au code qui l'a invoquée.
- D'une manière générale, la méthode **`invokeLater()`** doit être utilisée de préférence car la méthode **`invokeAndWait()`** peut poser certains problèmes d'interbloquage *(deadlock)* si le *Thread* dans lequel elle est invoquée maintient des verrous *(locks)* sur certains objets qui doivent être accédés par les instructions exécutées dans le *EDT*.
- *Remarque* : Si nécessaire, la méthode statique **`isDispatchThread()`** (de la classe **`EventQueue`**) permet de tester si un code s'exécute dans le *Event Dispatch Thread*.

SwingWorker Framework [1]

- Pour simplifier la gestion des applications qui comportent des activités dont la durée est indéterminée et qui affectent l'état de l'interface utilisateur, une classe utilitaire appelée **`SwingWorker`** est à disposition.
- Cette classe (qui se trouve dans le package *javax.swing.swingworker*) est un *framework* permettant de simplifier la séparation des tâches qui doivent être effectuées hors du *EDT* de celles qui touchent à l'interface et qui doivent donc obligatoirement être exécutées dans le *EDT*.
- Avant la version 1.6 du JDK, cette classe ne faisait pas partie de la plate-forme Java standard mais pouvait être téléchargée (site de *Sun*).
- A partir de la version 1.6, la classe **`SwingWorker`** a été sensiblement modifiée (avec généricité) et a été intégrée à la plate-forme Java.
- Pour utiliser cette classe abstraite, il faut créer une sous-classe et redéfinir la seule méthode abstraite **`doInBackground()`** dans laquelle seront placées toutes les instructions qui prennent du temps (sans interactions directes avec le GUI). La méthode **`doInBackground()`** retourne un objet de type **`T`** (paramètre générique de la classe).

SwingWorker Framework [2]

- La valeur retournée par la méthode `doInBackground()` peut être récupérée par la méthode `get()`.

  Attention : la méthode `get()` est bloquante et attendra que la méthode `doInBackground()` se termine. Un *timeout* fixant l'attente maximale peut cependant être passé en paramètre.

- La méthode `done()` peut également être redéfinie si nécessaire, elle sera invoquée lorsque la méthode `doInBackground()` se terminera (ou si `cancel()` est invoqué [voir plus loin]).
- Les instructions de la méthodes `done()` s'**exécutent dans le *EDT*** c'est donc là qu'il faut placer les instructions qui mettent à jour l'interface utilisateur (attention à ne pas y placer des instructions qui prennent beaucoup de temps !).
- Des mises à jour intermédiaires de l'interface graphique peuvent être effectuées en invoquant la méthode `publish()` dans la méthode `doInBackground()` et en récupérant les résultats dans la méthode `process()` (qui s'exécute dans le *EDT*).

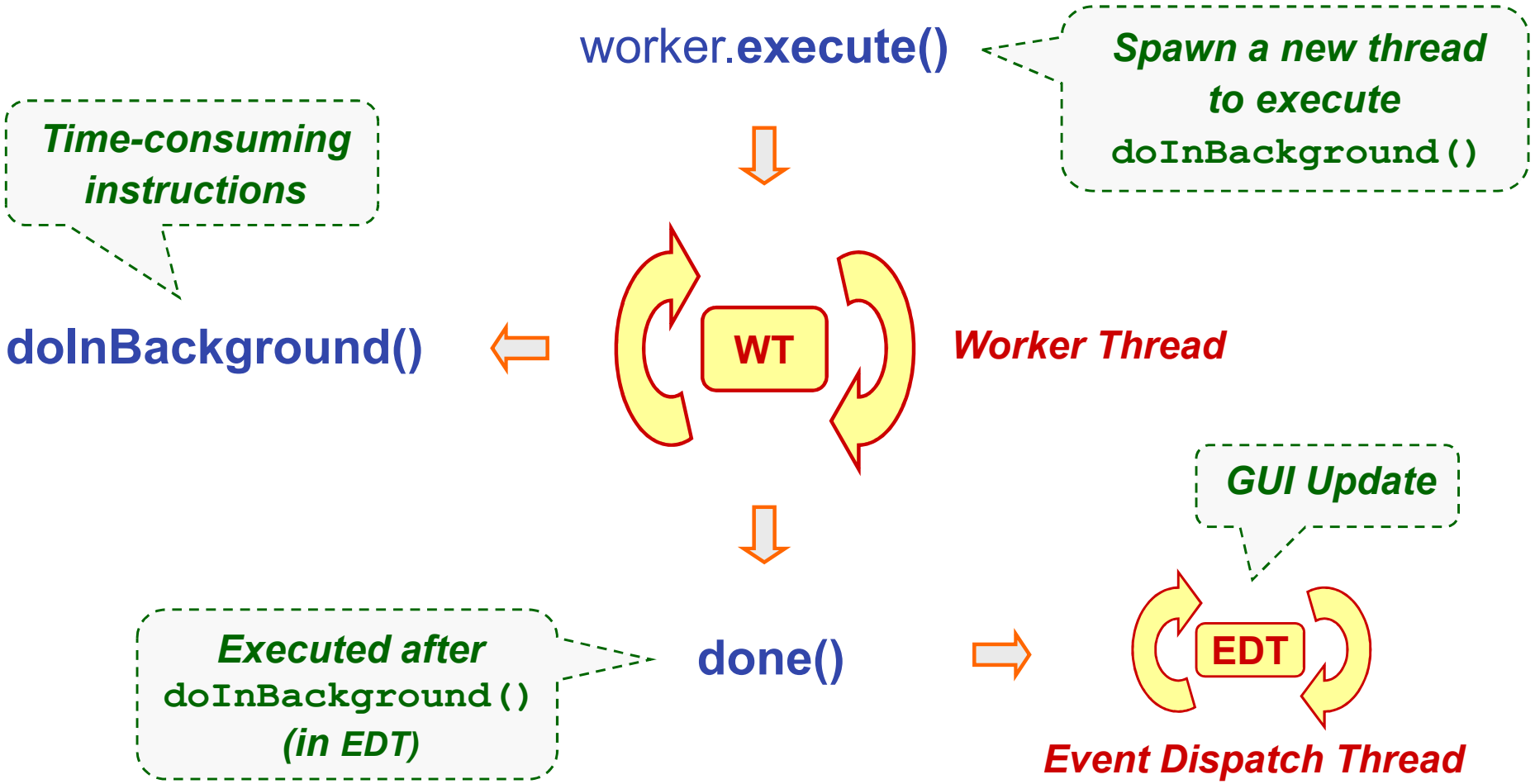
SwingWorker Framework [3]

- Les instructions de la méthode `doInBackground()` sont exécutées (dans un *Thread* indépendant) dès que la méthode `execute()` a été invoquée sur l'instance de `SwingWorker`.

- *Attention* : L'instance de `SwingWorker` ne peut pas être réutilisée. Un deuxième appel à `execute()` ne ré-exécutera pas les instructions de la méthode `doInBackground()`.

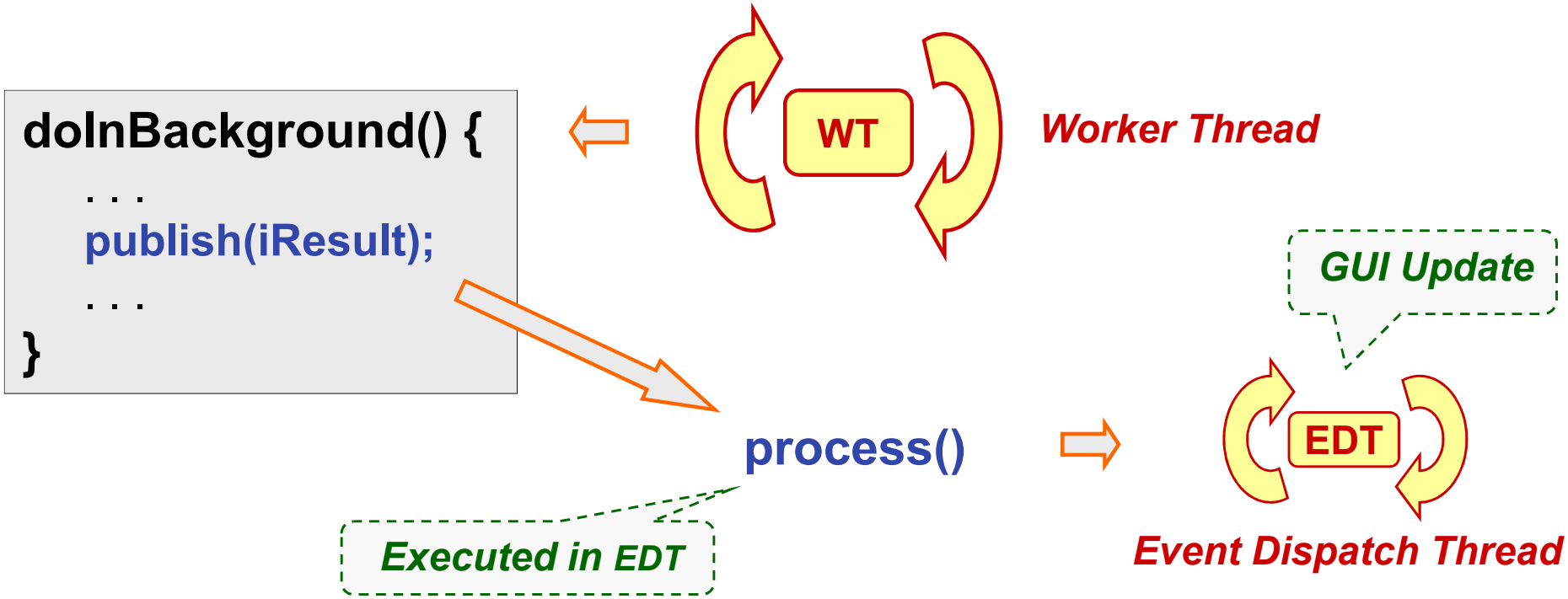
SwingWorker Model [1]

```
public class MyWorker extends SwingWorker <T, V> { … }
MyWorker worker = new MyWorker();
```



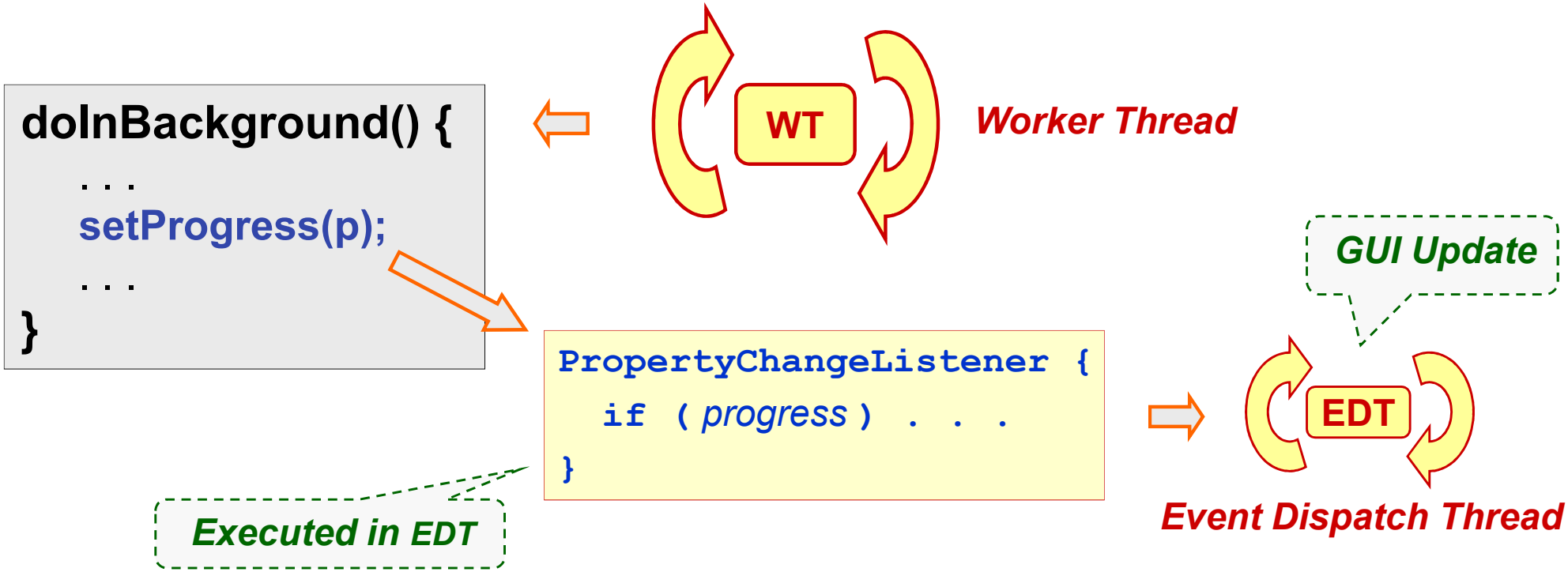
SwingWorker Model [2]

- La méthode `doInBackground()` peut transmettre des résultats intermédiaires (de type `V`) en invoquant, dans ses instructions, la méthode `publish()`.
- Le traitement de ces résultats partiels est effectué dans la méthode `process()` qui reçoit un liste d'objets de type `V`.
  La méthode `process()` s'exécute dans le *EDT*.



SwingWorker Model [3]

- La méthode **`doInBackground()`** peut également transmettre des informations de progression (entier 0...100) en invoquant, dans ses instructions, la méthode **`setProgress()`**.
- Le traitement de la progression s'effectue en créant et enregistrant un récepteur d'événement de type **`PropertyChangeListener`** et en surveillant la propriété "**`progress`**".



SwingWorker : Principe d'utilisation [1]

```
public class WorkerTask extends SwingWorker<Double, String> {

  //-------------------------------------------------------------------
  // doInBackground() - Code executed in a separated thread
  //-------------------------------------------------------------------
  protected Double doInBackground() throws Exception {
     ...
     ... publish(result);              // Send intermediate results
     ... setProgress(percentDone);     // Progression [0..100]
     ...
  }

  //-------------------------------------------------------------------
  // done() - Code executed in EDT, after doInBackground() has finished
  //-------------------------------------------------------------------
  protected void done() {
     ...
  }

  //-------------------------------------------------------------------
  // process() - Code executed in EDT to treat intermediate results
  //-------------------------------------------------------------------
  protected void process(List<String> list) {
     ...
  }
}
```

SwingWorker : Principe d'utilisation [2]

- Exemple de tâche à exécuter dans un *Thread* séparé

```
//-----------------------------------------------------------------
// doInBackground() - Code executed in a separated thread
//-----------------------------------------------------------------
protected Double doInBackground() throws Exception {

  double result  = 0;

  //--- Long task
  for (long i=1; i<nbSteps; i=i+2) {
    result += 1.0/i;
    if ((i-1)%100000 == 0) publish(""+result);   // Send intermediate results
    setProgress((int)((100*i)/(nbSteps-1)));     // Progression [0..100]

    //--- Check if task cancelled
    if (isCancelled()) return null;

    //--- Give other threads (of same priority) a chance to run
    Thread.yield();
  }

  return new Double(result);   // Result ( can be recovered by get() )
}
```


SwingWorker : Principe d'utilisation [3]

- Récupération d'informations intermédiaires et mise à jour de l'interface graphique (les instructions de la méthode **`process()`** sont exécutées dans le *EDT*).

```
//-----------------------------------------------------------------
// process() - Code executed in EDT to treat intermediate results
//-----------------------------------------------------------------
protected void process(List<String> list) {

  //--- Get intermediate results sent by publish() and transmitted
  //    in a List (multiple results could be grouped)
  String s = list.get(list.size()-1);

  //--- Update view
  view.lbResult.setText(s);
}
```

SwingWorker : Principe d'utilisation [4]

- Récupération des informations de progression émises par la méthode **`setProgress()`**.
- Un gestionnaire d'événements de type **`PropertyChangeListener`** doit être créé et enregistré sur l'instance de **`SwingWorker`**.
- La propriété porte le nom "**`progress`**" (valeurs comprises entre [0..100]).
- Exemple sous forme de classe interne anonyme :

```
//-------------------------------------------------------------
// Controller to manage the progression (activated by setProgress())
// Update the state of the progress-bar
//-------------------------------------------------------------
worker.addPropertyChangeListener(new PropertyChangeListener() {

  public void propertyChange(PropertyChangeEvent evt) {
    if ("progress".equals(evt.getPropertyName())) {
      view.pbCurState.setValue((Integer)evt.getNewValue());
    }
  }

});
```

SwingWorker : Principe d'utilisation [5]

- Le code de la méthode **`done()`** est exécuté après la fin de la méthode **`doInBackground()`** ou après invocation de la méthode **`cancel()`**. La méthode **`get()`** permet de récupérer la valeur retournée par la méthode **`doInBackground()`**.
- *Attention* : Compte tenu de son fonctionnement asynchrone, le code de la méthode **`process()`** peut être exécuté après le code de **`done()`**.

```
//-------------------------------------------------------------------
// done() - Code executed in EDT, after doInBackground() has finished
//-------------------------------------------------------------------
protected void done() {
  view.btStart.setEnabled(true);
  view.btStop.setEnabled(false);

  if (isCancelled()) System.out.println("=== Cancelled ===");
  else {
    try {
      view.lbResult.setText(get().toString());
    } catch (Exception e) {
      System.out.println("=== Interrupted ===");
    }
  }
}
```

Interruption de SwingWorker [1]

- La méthode `cancel()` peut être utilisée pour demander l'interruption de l'activité du *Thread SwingWorker* (instructions contenues dans la méthode `doInBackbround()`).
- La méthode `cancel()` prend un paramètre de type booléen :
  - **true** : Le flag ***cancel*** et le flag ***interrupt*** sont mis à `true`
  - **false** : Seul le flag ***cancel*** est mis à `true`
- La prise en compte de l'interruption par le flag ***cancel*** est à la charge du programmeur (aucun effet sinon). Elle s'effectuera en invoquant la méthode `isCancelled()` dans la méthode `doInBackground()`.
- La prise en compte de l'interruption par le flag ***interrupt*** s'effectuera
  - Si la méthode `doInBackground()` invoque les méthodes `sleep()` ou `wait()` qui génèreront l'exception `InterruptedException` en cas d'interruption
  - Si la méthode `doInBackground()` interroge explicitement l'état du flag d'interruption en invoquant `Thread.interrupted()`
- L'interruption est donc **basée sur la coopération** !

Interruption de SwingWorker [2]

- Après exécution de **`cancel()`**, la méthode **`done()`** sera invoquée (même si la méthode **`doInBackground()`** ne s'est pas encore arrêtée).
- La technique utilisant la **`isCancelled()`** a été illustrée dans l'exemple donné pour la méthode **`doInBackbround()`** (voir pages précédentes).
- Exemple utilisant le flag ***interrupt*** :

```
protected String doInBackground() throws Exception {
  try {
    for(int i=0; i<NUMLOOPS; i++) {
      updateCounter(i);
      if (Thread.interrupted()) throw new InterruptedException();
      doWork(i);
      Thread.sleep(500);     // Can throw InterruptedException
    }
  }
  catch (InterruptedException e) {
    updateCounter(0);
    return "Interrupted";
  }
  return "All done";
}
```

SwingWorker et activités GUI

- Si la méthode `doInBackground()` doit effectuer des actions liées à l'interface utilisateur (une mise à jour par exemple) il faut impérativement que ces instructions s'exécutent dans le *EDT*.
- On utilisera de préférence la méthode `setProgress()` vue précédemment qui permet de transférer des résultats intermédiaires à un processus qui s'exécute dans le *EDT* (un `PropertyChangeListener`).
- Dans certaines situations particulières on peut placer ces instructions dans un `Runnable` et les injecter dans le *EDT* avec `invokeLater()`.

```
void updateCounter(final int i) {
    model.setCounter(i);

    Runnable updateProgressBarValue = new Runnable() {
      public void run() {
        progressBar.setValue(i);
      }
    };

    EventQueue.invokeLater(updateProgressBarValue);
}
```

Autres *Framework*

- Le modèle asynchrone de *SwingWorker* peut présenter certaines faiblesses dans des applications complexes.
- D'autres solutions *(Framework)* ont été proposées pour remédier à certains des inconvénients de *SwingWorker*.
- Parmi ces *Framework* on peut citer : ***Foxtrot*** (`foxtrot.sourceforge.net`) qui se base sur un modèle synchrone.
- Parmi les avantages avancés par les concepteurs on peut mentionner :
  - une API simple
  - une meilleure symétrie et lisibilité du code
  - un traitement facilité des exceptions
  - une maintenance plus simple
- ***Foxtrot*** est composé de trois classes principales (`Worker`, `Task`, `Job`)
- Autre *Framework* avec le même objectif : ***Spin*** (`spin.sourceforge.net`)
- Voir API, documentation et exemples sur les sites correspondants.