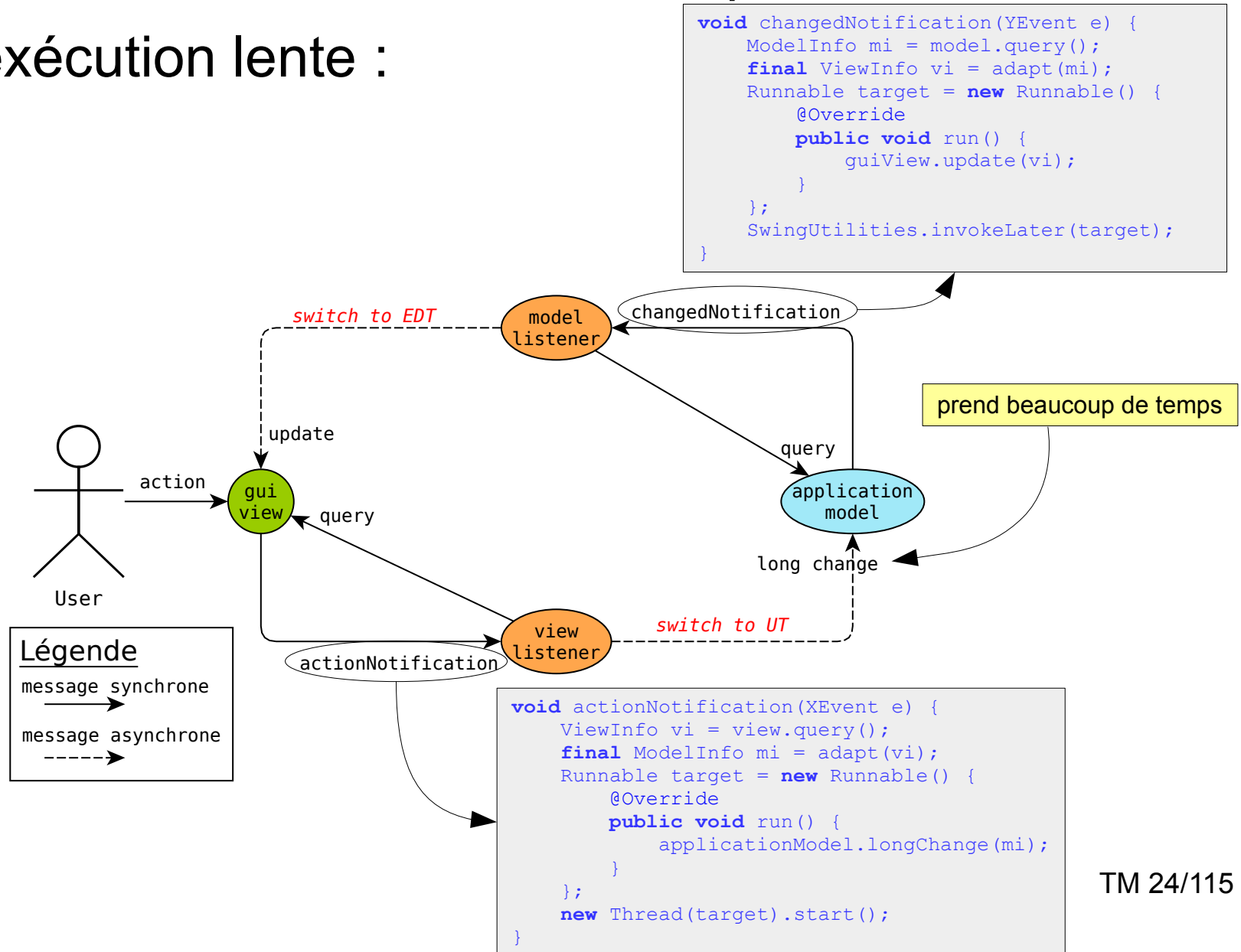


Swing et les threads

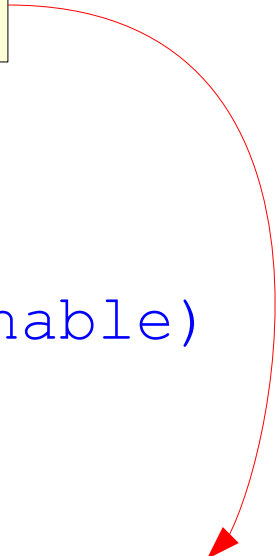
- Swing n'est pas *thread-safe*
 - accès aux composants → sur un seul thread
 - sinon
 - interférence entre threads
 - inconsistance de la mémoire
- EDT
 - *toutes* les opérations de dessin (`PaintEvent`)
 - *toutes* les fonctions réflexes de *tous* les écouteurs qui accèdent à des composants Swing (ou à leurs modèles)

- par conséquent

- méthode réflexe → exécution rapide sur EDT
- si exécution lente :



- lève des exceptions
- ne peut pas être invoquée sur EDT



- Notification asynchrone sur EDT :
 - **void** `SwingUtilities.invokeLater(Runnable)`
- Notification synchrone sur EDT :
 - **void** `SwingUtilities.invokeAndWait(Runnable)`
- Rq : notification des `PropertyChangeListener` sur EDT possible avec
 - `SwingPropertyChangeSupport`
 - en passant **true** comme second paramètre au constructeur

Rappel : un composant est **réalisé** lorsque
il est placé sur une frame ayant exécuté
`setVisible(true)`, `show()` ou `pack()`

- *Urban legend*

- « *un composant Swing peut être manipulé par n'importe quel thread tant qu'il n'est pas réalisé* »

- Sous-entendu :

- on peut créer un composant Swing dans un thread
puis le manipuler avec EDT uniquement

- /! C'est (devenu) **FAUX**, lire par exemple :

- <http://web.archive.org/web/20090925191745/http://eppleton.sharedhost.de/blog/?p=806>

- Conclusion

```
SwingUtilities.invokeLater(new Runnable() {  
    @Override  
    public void run() {  
        new Appli(...).display();  
    }  
});
```

- Toutefois quelques méthodes sont *thread-safe*, par exemple
 - dans `Component` : **void** `repaint()`
 - dans `JComponent` : **void** `revalidate()`
 - dans `Container` :
 - `<T extends EventListener> T[]`
`getListeners(Class<T> listenerType)`
 - les méthodes :
 - **void** `addXListener(XListener)`
 - **void** `removeXListener(XListener)`

```
public void append(String str) {  
    Document doc = getDocument();  
    if (doc != null) {  
        try {  
            doc.insertString(doc.getLength(), ...);  
        } catch (BadLocationException e) {  
        }  
    }  
}
```

- Attention !
- Dans la doc Java 6 `JTextArea.append` est soit-disant *thread-safe*
 - c'est vrai dans un sens :
 - `AbstractDocument.insertString` utilise un verrou
 - mais `append` est une opération non atomique !
- (Rq : la doc Java 7 ne dit plus que `append` est *thread-safe*)
- Conclusion : ne pas considérer que `append` est *thread-safe*

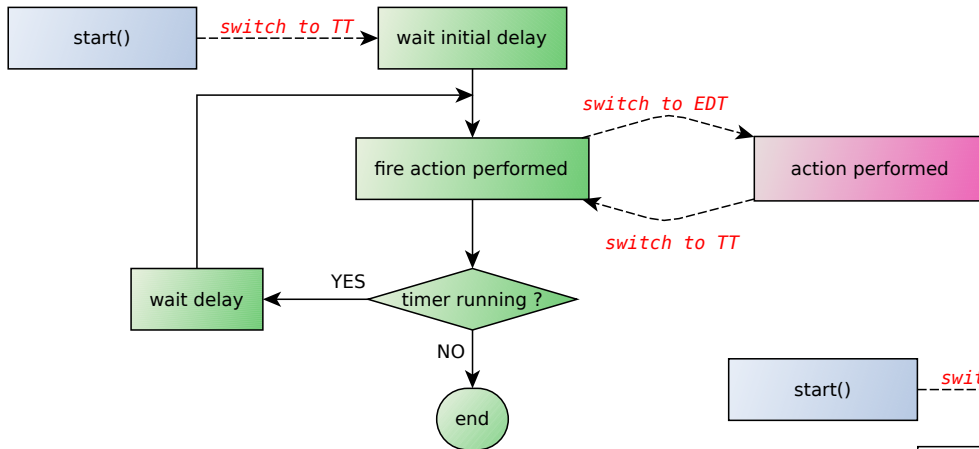
Utilisation d'un `Timer` Swing

!! ne fonctionne **que** dans une application graphique

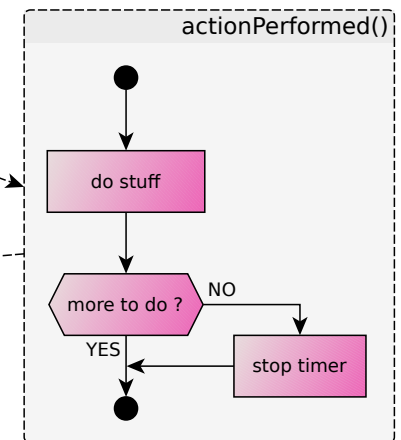
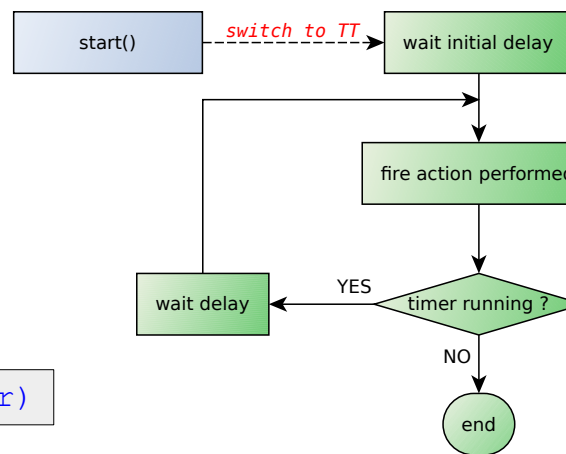
- Principe
 - découper une tâche répétitive en sous-tâches qui s'exécutent sur EDT à intervalles réguliers
- `new Timer(int d, ActionListener al)`
 - `d` : périodicité de la tâche en ms
 - `al.actionPerformed` tâche exécutée sur EDT
- `void {start,stop}()`
 - contrôlent l'exécution des tâches
- `boolean isRunning()`
 - indique si le timer est en cours d'exécution

on peut ajouter d'autres tâches avec `addActionListener`

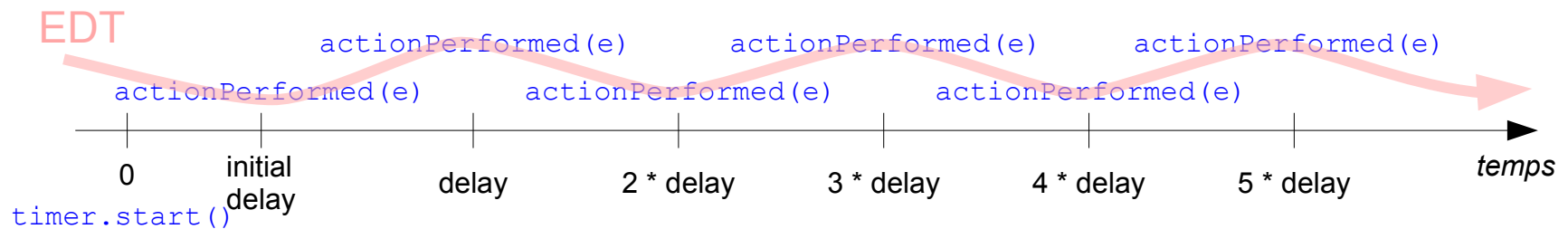
javax.swing.Timer



Timer qui s'arrête de lui-même

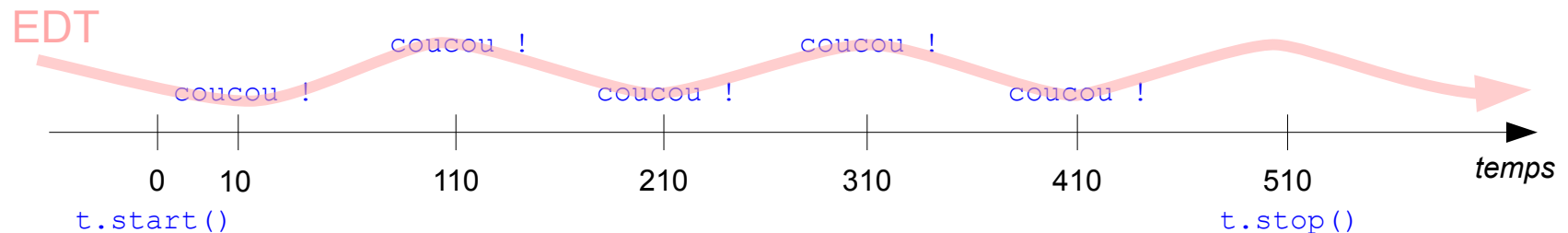


```
new Timer(delay, anActionListener)
```

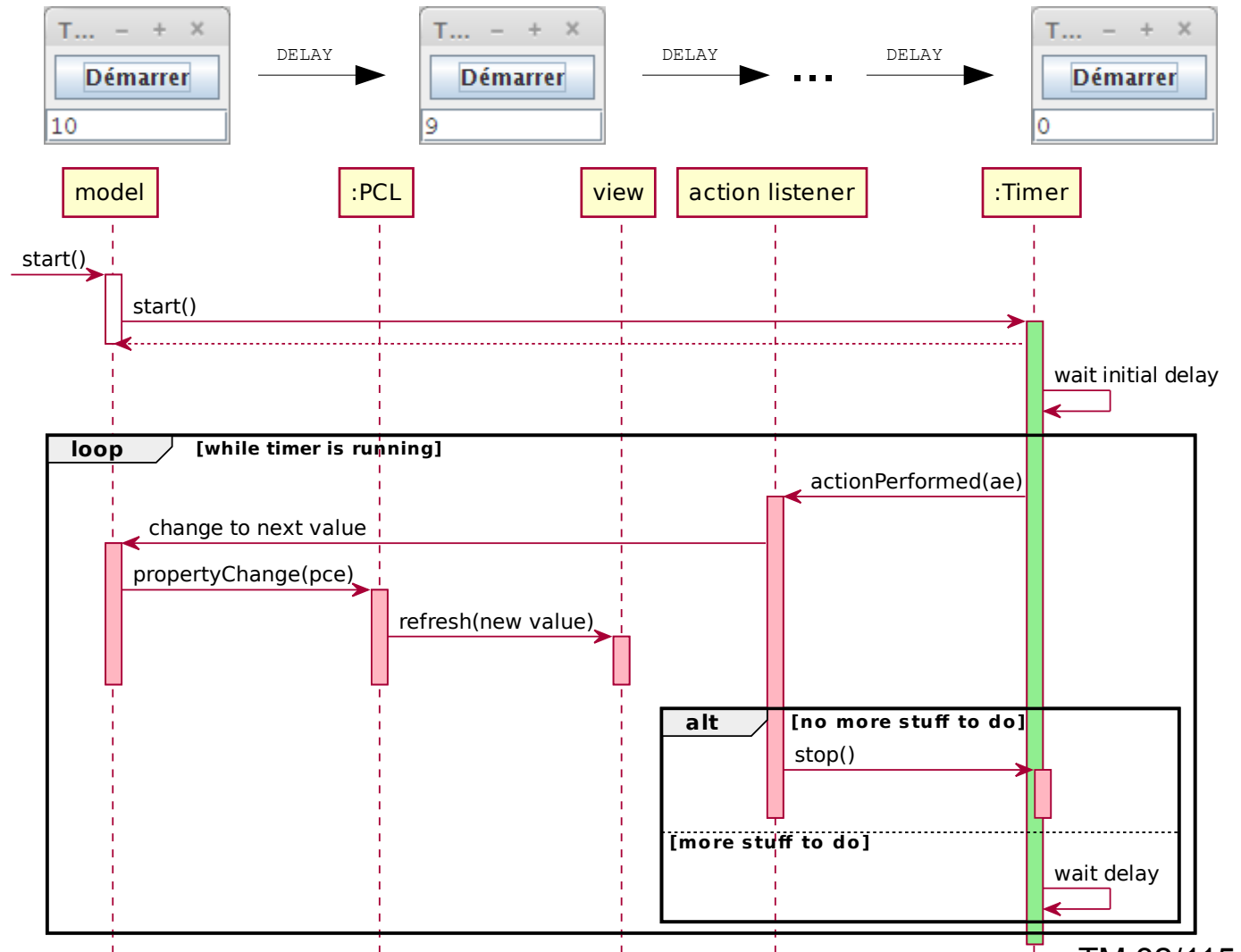
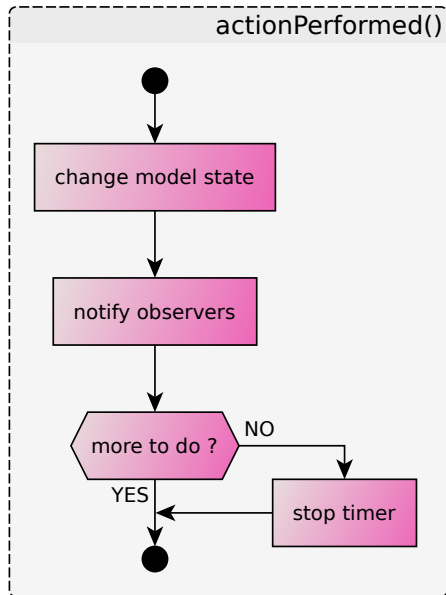


Utilisation triviale d'un `Timer`


```
ActionListener task = new ActionListener() {  
    private int i = 0;  
    public void actionPerformed(ActionEvent e) {  
        if (i < 5) {  
            System.out.println("coucou !");  
            i += 1;  
        } else {  
            t.stop();  
        }  
    }  
};  
t = new Timer(100, task);  
t.setInitialDelay(10);  
t.start();
```



Un *Timer* inscrit dans une architecture MVC



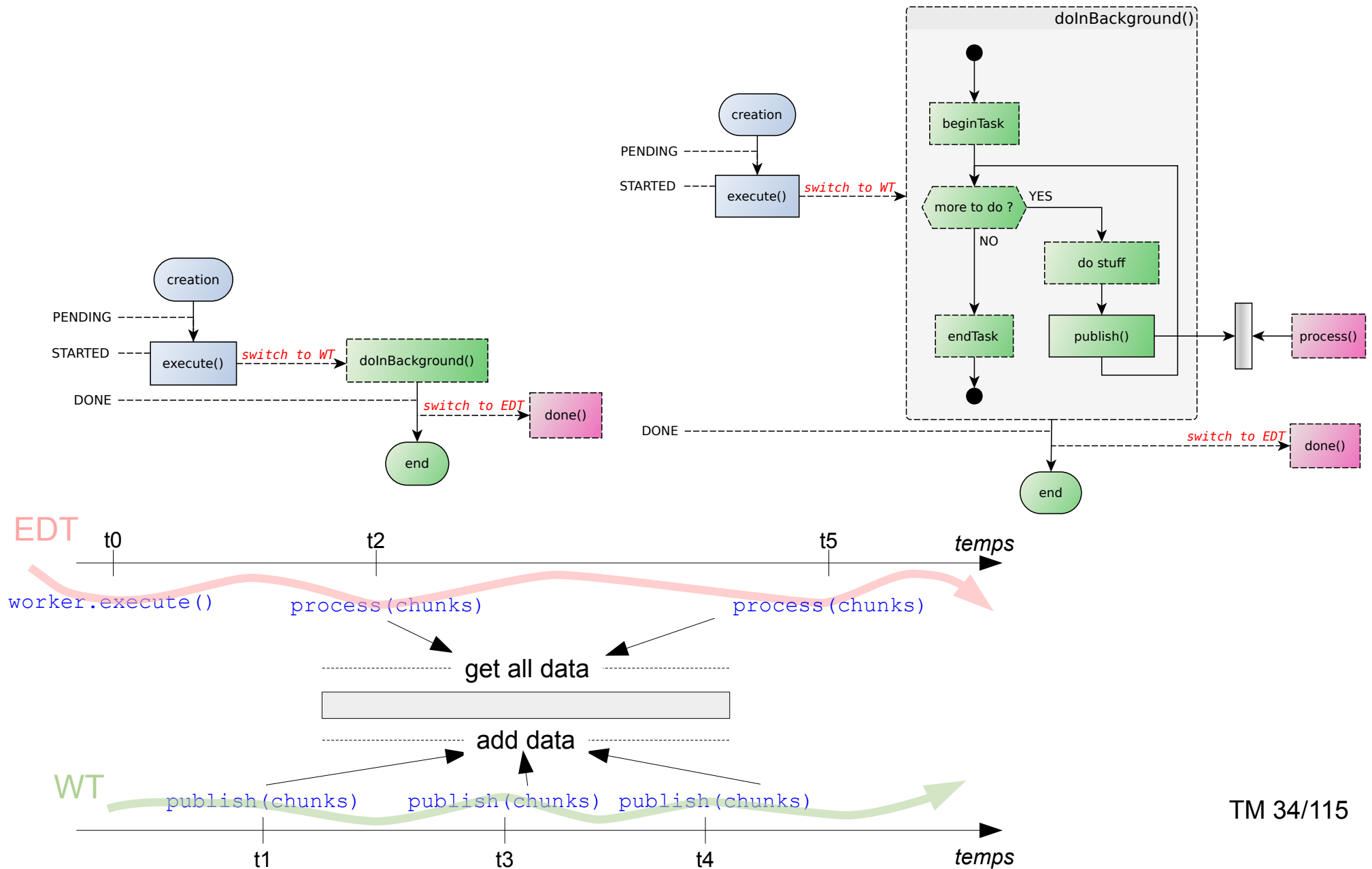
Utilisation d'un `SwingWorker`

 ne fonctionne **que** dans une application graphique

- Principe

- `void execute()` : démarrage de la tâche
- `T get()` : récupération du résultat (bloquante jusqu'en fin de tâche)
- `boolean cancel(boolean)` : annulation de la tâche
- propriétés (R) liées prédéfinies
 - `state (StateValue)`
 - `PENDING` : worker créé mais tâche non démarrée
 - `STARTED` : tâche démarrée mais non terminée
 - `DONE` : tâche terminée
 - `progress (int, entre 0 et 100)`
 - notification des PCL sur EDT

javax.swing.SwingWorker



Exemple trivial de `SwingWorker`

- Aucune interaction avec l'environnement graphique

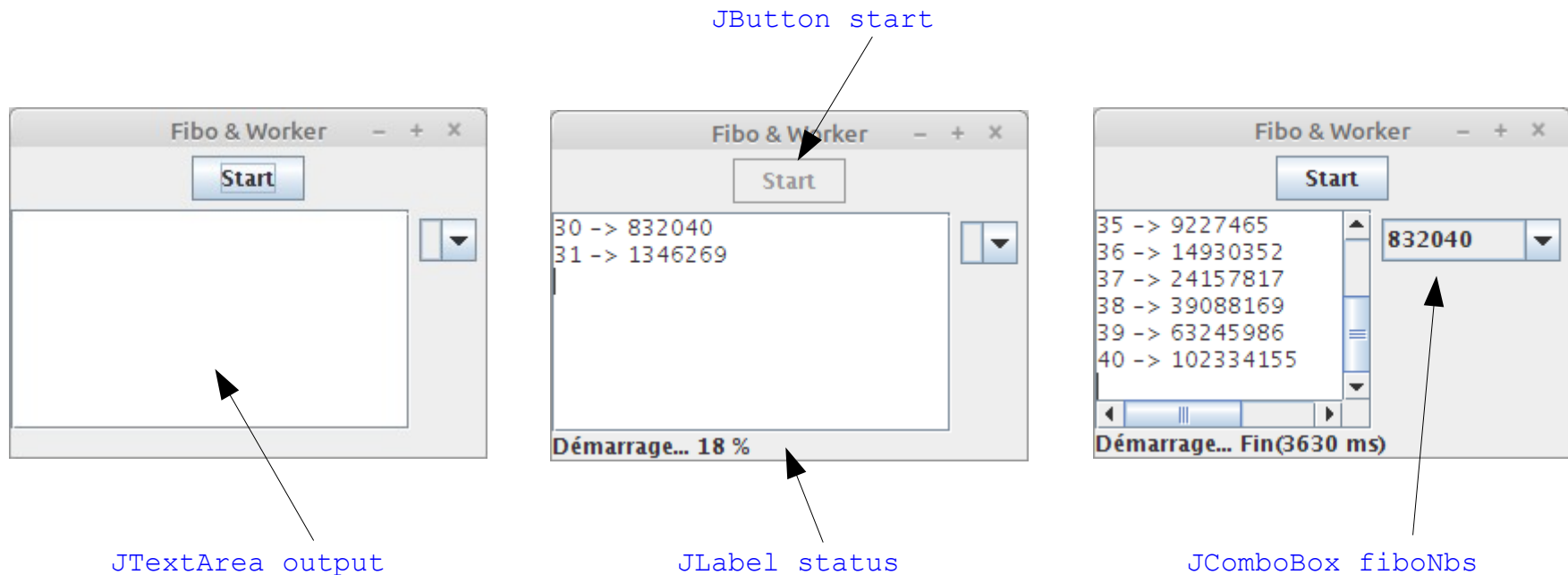
```
new SwingWorker<Void, Void>() {  
    @Override  
    protected Void doInBackground() throws Exception {  
        System.out.print("démarrage... ");  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            // rien, on s'arrête  
        }  
        System.out.println("arrêt");  
        return null;  
    }  
}.execute();
```

T ::= Void
V ::= Void

Exemple d'utilisation complet

SwingWorker

- Affichage des nombres de Fibonacci

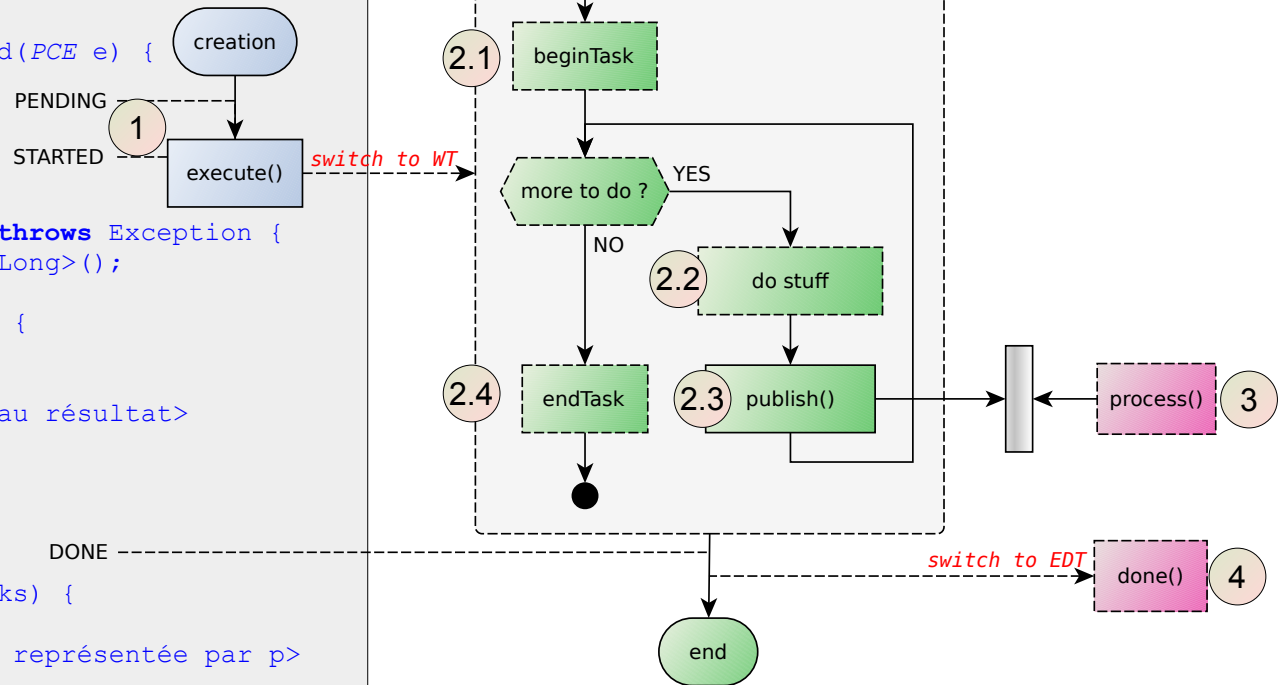


```

public class AppliFiboWorker {
    ...
    private void connectControllers() {
        ...
        start.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                ...
                worker = new FiboWorker(30, 40);
                worker.execute();
            }
        });
    }
    private class FiboWorker extends SwingWorker<List<Long>, Pair> {
        FiboWorker(int b, int e) {
            ...
            addPropertyChangeListener(
                new PCL() {
                    public void propertyChanged(PCE e) {
                        <modifier status>
                    }
                }
            );
        }
        @Override
        protected List<Long> doInBackground() throws Exception {
            2.1 List<Long> result = new ArrayList<Long>();
            <démarrer chrono>
            2.2 for (int i = begin; i <= end; i++) {
                fibo.computeValue(i);
                <publier la nouvelle valeur
            2.3         & ajouter la nouvelle valeur au résultat>
            }
            2.4 <arrêter chrono>
            return result;
        }
        @Override
        3 protected void process(List<Pair> chunks) {
            for (Pair p : chunks) {
                <afficher sur output la valeur représentée par p>
                <modifier status>
            }
        }
        @Override
        4 protected void done() {
            <modifier status, start et le modèle de fiboNbs>
        }
    }
}

```

utilisation NON MVC !



Codage d'un `SwingWorker<T, V>`

- **`protected T doInBackground()`**
 - **`throws Exception`**
 - méthode abstraite conteneur de la tâche à exécuter
 - s'exécute sur un thread privé (WT)
 - `T` : type de la valeur retournée par la tâche
 - peut lever des exception
 - peut retourner une valeur (récupérable avec `get`)

```
@Override
protected List<Long> doInBackground() throws Exception {
    List<Long> result = new ArrayList<Long>();
    <démarrer chrono>
    for (int i = begin; i <= end; i++) {
        fibo.computeValue(i);
        <publier la nouvelle valeur
        & ajouter la nouvelle valeur au résultat>
    }
    <arrêter chrono>
    return result;
}
```


Codage d'un `SwingWorker<T, V>`

- **protected void done()**
 - méthode vide, conteneur de l'action à exécuter immédiatement en fin de tâche
 - s'exécute sur EDT
 - exécutée aussi en cas d'annulation
- **public T get()**
 - méthode bloquante jusqu'à la fin de la tâche
 - `InterruptedException` si thread courant interrompu
 - `ExecutionException` si erreur au cours de la tâche
 - `CancellationException` si tâche annulée

```
@Override
protected void done() {
    ...
    List<Long> result = null;
    try {
        result = get();
    } catch (Exception e) {
        // rien : result == null
    }
    if (result == null) {
        fiboNbs.setModel(new DefaultComboModel());
    } else {
        Object[] data = result.toArray();
        fiboNbs.setModel(new DefaultComboModel(data));
    }
}
```

- **protected void** process(List<V>)
 - méthode vide, conteneur du traitement des valeurs transmises depuis `doInBackground()`
 - s'exécute sur EDT
 - **V** : type des valeurs échangées entre WT et EDT
 - la transmission depuis `doInBackground` se fait par appel à **void** `publish(V...)`

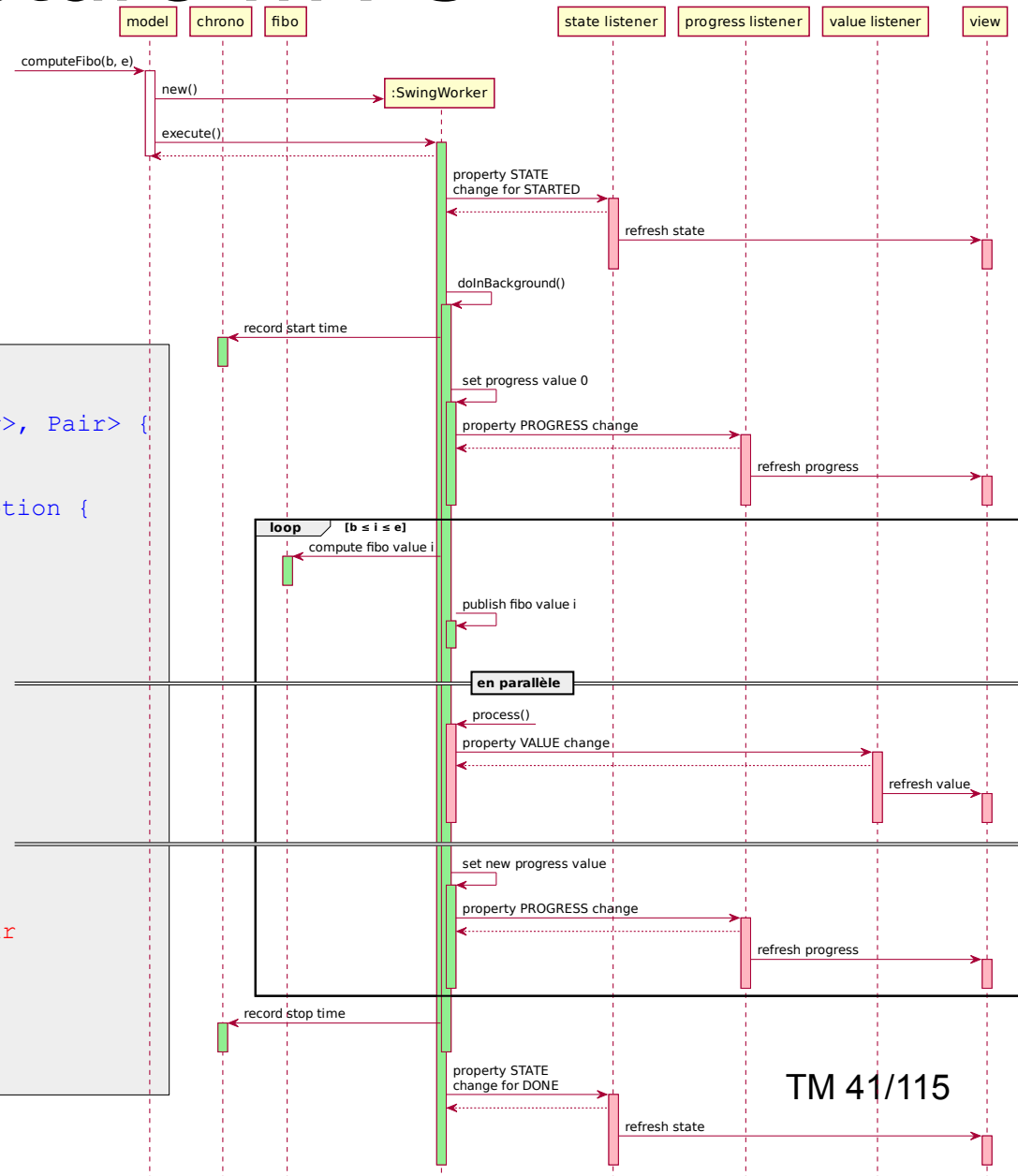
```
@Override
protected List<Long> doInBackground() throws Exception {
    ...
    for (int i = begin; i <= end; i++) {
        ...
        publish(new Pair(fibo.getRank(), fibo.getValue()));
        ...
    }
    ...
}

@Override
protected void process(List<Pair> chunks) {
    for (Pair p : chunks) {
        String line = p.rank() + " -> " + p.value() + "\n";
        output.append(line);
        ...
    }
}
```

- **protected final void** publish(V...)
 - s'exécute sur WT mais appelle `process` sur EDT
 - ne doit être utilisée que dans `doInBackground`
 - compactage éventuel des appels à `process` :
 - `publish(a)` puis `publish(b)` peut donner lieu à `process(a, b)`

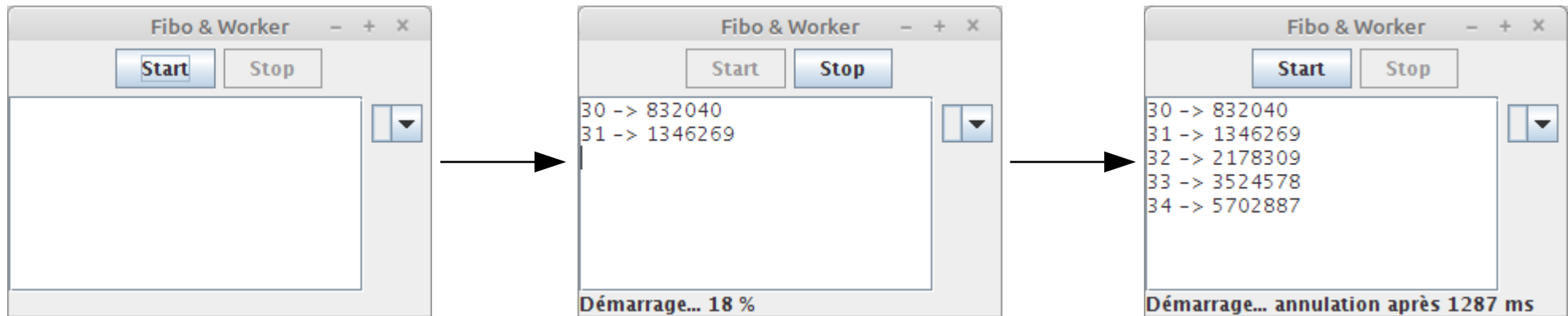
Un `SwingWorker` inscrit dans une architecture MVC

```
class FiboModel {
    ...
    private class FiboWorker extends SwingWorker<List<Long>, Pair> {
        ...
        @Override
        protected List<Long> doInBackground() throws Exception {
            List<Long> result = new ArrayList<Long>();
            <démarrer chrono & initialiser progress>
            for (int i = begin; i <= end; i++) {
                fibo.computeValue(i);
                <publier la nouvelle valeur
                & ajouter la nouvelle valeur au résultat
                & modifier progress>
            }
            <arrêter chrono>
            return result;
        }
        @Override
        protected void process(List<Pair> chunks) {
            for (Pair p : chunks) {
                <notifier les PCL d'un changement de valeur
                pour la pte value>
            }
        }
    }
}
```



Exemple d'annulation sur un SwingWorker

- Annulation par clic sur le bouton « Stop »

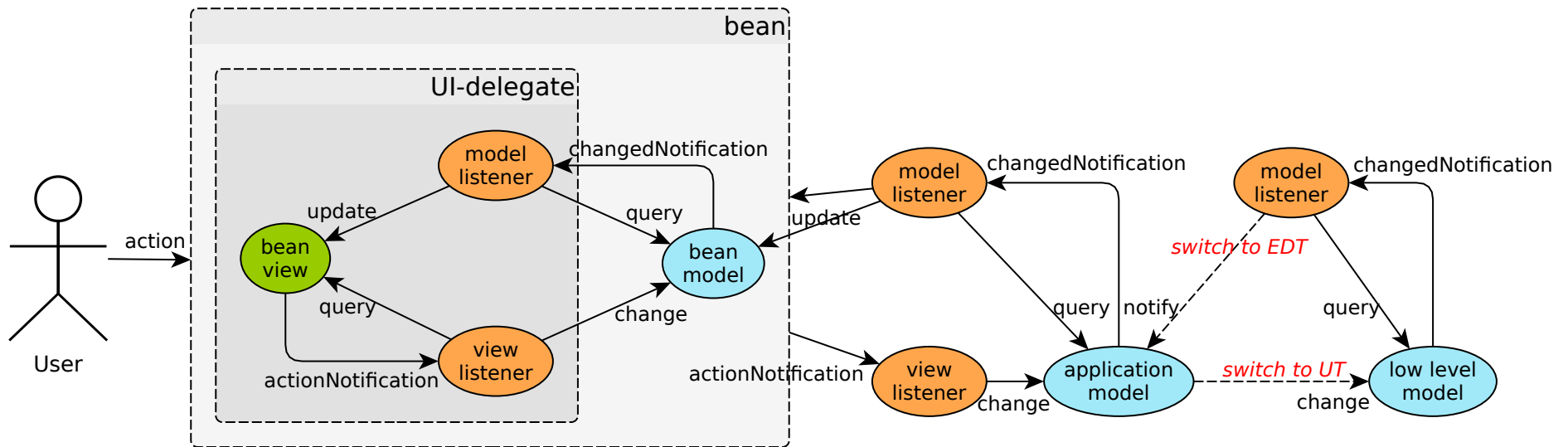


```
class FiboModel {
    public void cancel() { worker.cancel(true); }
    private class FiboWorker extends SwingWorker<List<Long>, Pair> {
        @Override
        protected List<Long> doInBackground() throws Exception {
            List<Long> result = new ArrayList<Long>();
            <démarrer chrono & initialiser progress>
            for (int i = begin; !isCancelled() && i <= end; i++) {
                try {
                    fibo.computeValue(i);
                } catch (InterruptedException e1) {
                    // rien
                }
                if (isCancelled()) {
                    result = null;
                } else {
                    <publier la nouvelle valeur
                    & ajouter la nouvelle valeur au résultat
                    & modifier progress>
                }
            }
            return result;
        }
        @Override
        protected void done() { <arrêter chrono> }
    }
}
```

```
public class AppliFibo {
    ...
    private void connectControllers() {
        ...
        cancelButton.addActionListener(
            new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    model.cancel();
                }
            });
    }
}
```

- **public final boolean** `cancel(boolean)`
 - retourne **true** ssi la tâche a pu être annulée
 - interrompt un appel bloquant dans WT
ssi `arg == true`
 - immédiatement (même si `doInB` n'est pas finie !)
 - passe dans l'état `DONE`
 - exécute `done()`
 - un appel à `get()` lèvera une `CancellationExpn`
 - `doInBackground` doit gérer l'annulation
 - `isCancelled()` indique s'il y a eu annulation

Application graphique concurrente schéma de synthèse

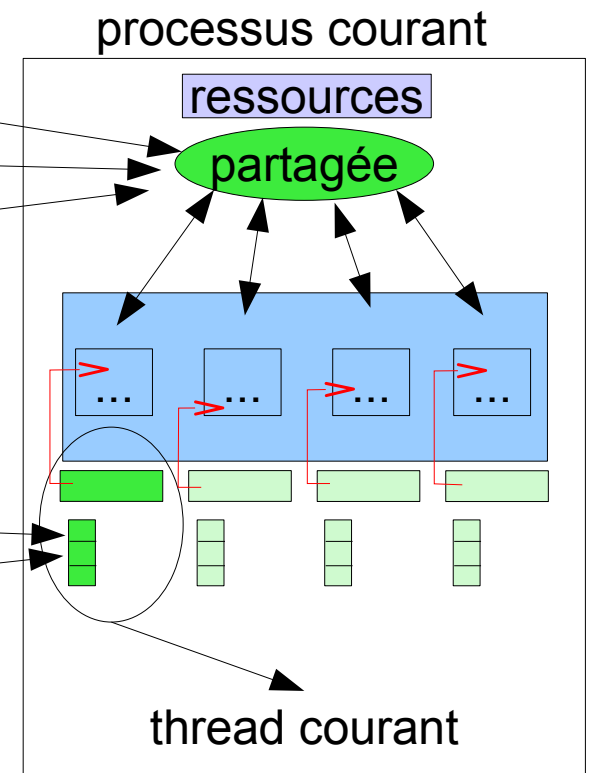


Vocabulaire de la concurrence

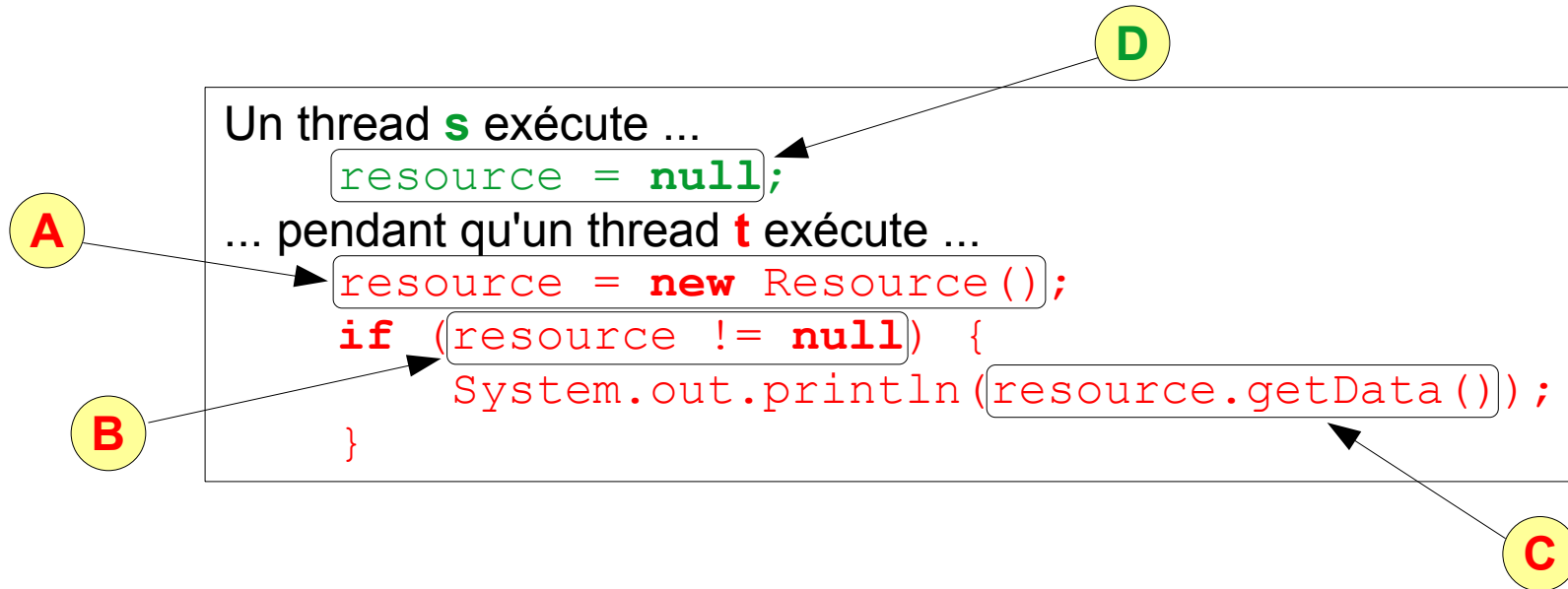
- **Variable partagée** = plusieurs threads peuvent accéder à cette variable au cours de l'exécution du programme
- **Accès concurrent à une variable** = les threads peuvent accéder « en même temps » à cette variable (partagée)
- **Accès exclusif à une variable** = tant qu'un thread accède à cette variable (partagée), les autres threads ne peuvent pas y accéder

Peut-on partager toute variable ?

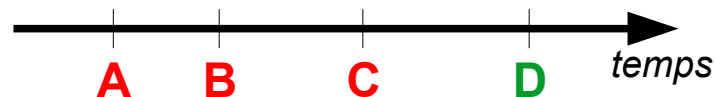
- Peuvent être partagées :
 - variables d'instance
 - variables de classe
 - éléments des tableaux
- Ne peuvent pas être partagées :
 - variables locales
 - paramètres formels



Situation de compétition (*race condition*)



une possibilité d'exécution :



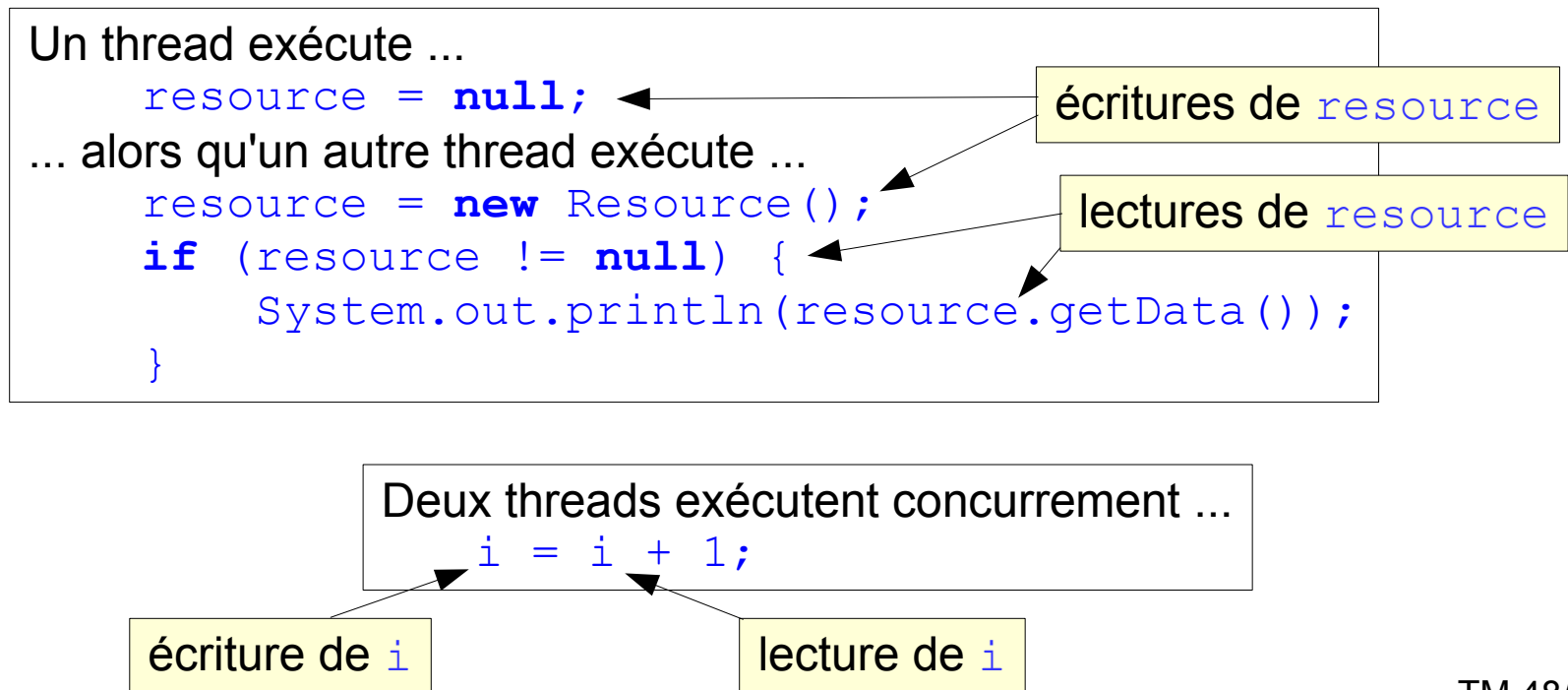
comportement correct

une autre possibilité d'exécution :



`NullPointerException`

- **Situation de compétition** (*race condition*) : lorsque le comportement d'une portion de code dépend de l'ordre d'activation des threads
- Exemples :



```
class Accu extends java.lang.Object{
Accu();
    Code:
        0:    aload_0
        1:    invokespecial    #10; //Method java/lang/Object."<init>":()V
        4:    return
```

```
public void inc();
```

```
    Code:
        0:    aload_0
        1:    dup
        2:    getfield            #17;
        5:    lconst_1
        6:    ladd
        7:    putfield            #17;
       10:    return
```

```
class Accu {
    private long i;
    public void inc() {
        i = i + 1;
    }
    public long value() {
        return i;
    }
}
```

```
public long value();
```

```
    Code:
        0:    aload_0
        1:    getfield            #17;
        4:    lreturn
```

```
}
```

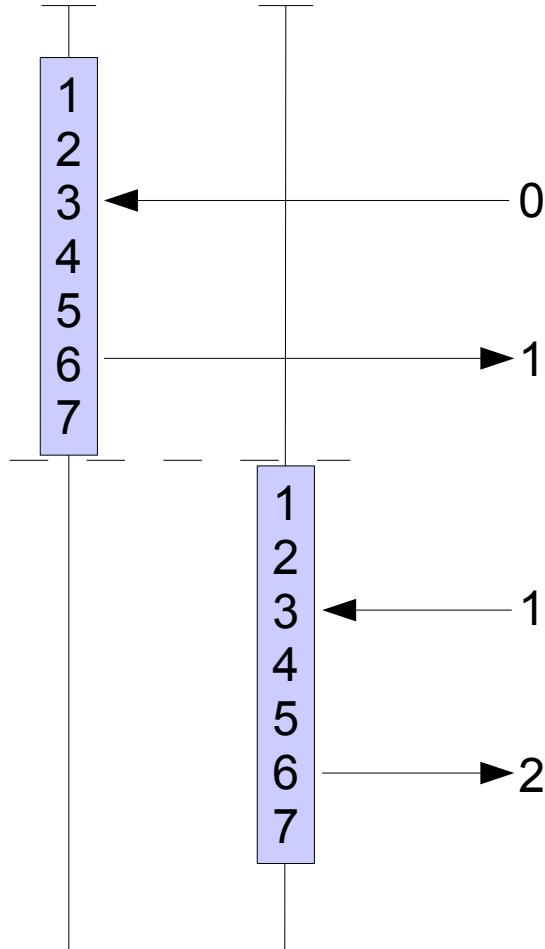
```
void inc()
```

```

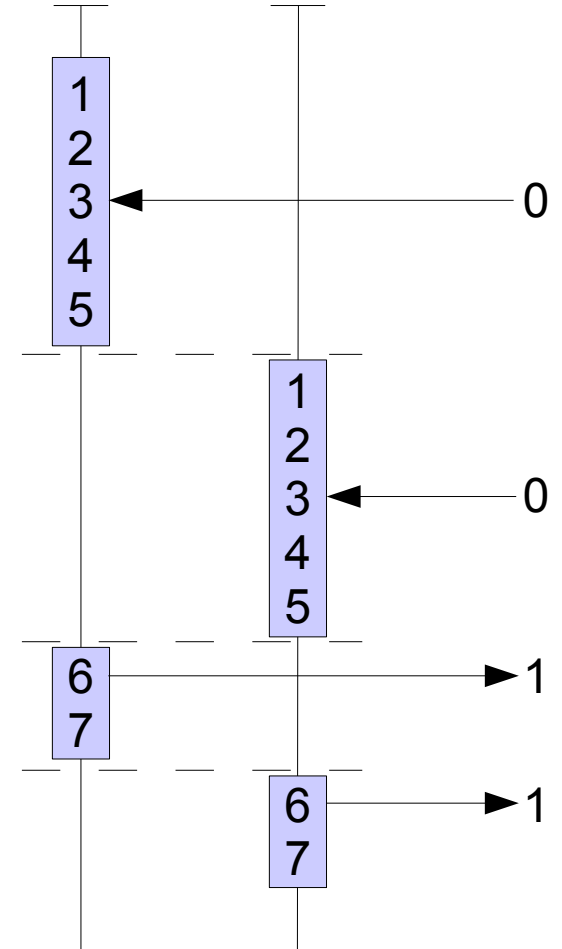
1  aload_0
2  dup
3  getfield
4  lconst_1
5  ladd
6  putfield
7  return

```

Thread t Thread s variable i



Thread t Thread s variable i



Synchronisation de section critique

t demande le verrou de (l'objet référencé par) lock

```
class Accu {  
    private final Object lock = new Object();  
    private long i;  
    public void inc() {  
        synchronized(lock) {  
            i = i + 1;  
        }  
    }  
    public long value() {  
        synchronized(lock) {  
            return i;  
        }  
    }  
}
```

t prend le verrou de lock

Pendant que t exécute ce bloc
tout thread demandant le
verrou de lock sera bloqué

section critique :
portion de code qui doit être exécutée
de manière atomique et par un seul thread à la fois
pour préserver la sémantique du programme

t libère le verrou de lock

Tous les threads bloqués sur
lock ont une chance d'obtenir
le verrou de lock

si plusieurs threads exécutent `inc` et `value` alors

- `i` est une *variable partagée*
- l'accès à `i` est *exclusif*
- si les instructions n'étaient pas synchronisées
alors on serait dans une *situation de compétition*

Méthodes synchronisées

```
class Accu {  
    private long i;  
    public synchronized void inc() {  
        i = i + 1;  
    }  
    ...  
}
```

```
class Accu {  
    private long i;  
    public void inc() {  
        synchronized (this) {  
            i = i + 1;  
        }  
    }  
    ...  
}
```

résultats équivalents :
`i = i + 1;`
exécutée dans un bloc
synchronisé sur **this**

Méthodes statiques synchronisées

```
class UneClasse {  
    ...  
    public static synchronized void m() {  
        ...  
    }  
}
```

```
class UneClasse {  
    ...  
    public static void m() {  
        synchronized (UneClasse.class) {  
            ...  
        }  
    }  
}
```

résultats équivalents :
corps de `m()`
exécuté dans un bloc synchronisé
sur `UneClasse.class`

Propriétés des verrous

- Tout objet possède un verrou qui est réentrant :
 - ré-obtention automatique d'un verrou déjà acquis
 - la libération d'un verrou doit se faire autant de fois que son acquisition

```
class X {  
    public synchronized void m() {  
        ...  
    }  
    public synchronized void p() {  
        this.m();  
        ...  
    }  
}
```

Durant cet appel,
le verrou de **this**
aura été pris 2 fois

- Conservent leurs verrous :
 - thread évincé
 - thread ayant rendu la main (**yield**)
 - thread endormi (**sleep**)