Thread & Timer

Rémi Forax

Problème de la gestion des threads

- Les toolkits graphique interdissent d'avoir **plusieurs** threads qui modifient l'interface graphique en même temps

- Une interface graphique est **mono-threadée**

EventDispatchThread

- En Swing, la condition est plus drastique :
  - une seule thread (EventDispatchThread ou **EDT**) **à le droit** d'effectuer des modifications.
  - Cette thread **n'est pas** créé par l'utilisateur
  - Cette thread s'occupe à la fois du traitement des évènement **et** du raffraichissement de l'interface

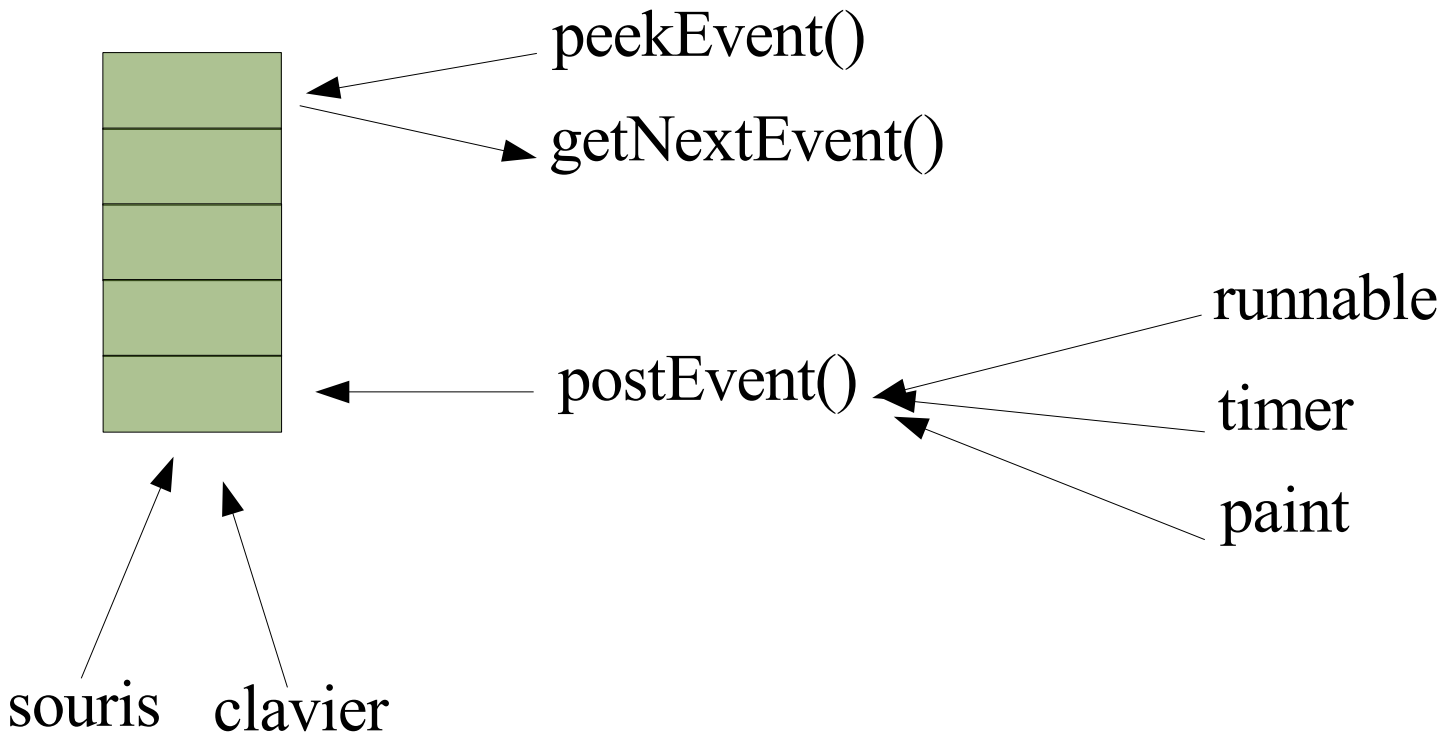
- Si une autre thread fait des modifications, cela peut arriver alors que l'EDT effectue un raffraichissement de l'écran => oups

Boulot de l'EDT

- Swing permet à une seul Thread, l'EventDispatchThread de modifier l'interface graphique
- L'EventDispatchThread à pour tâche :
  - De pomper les évènement de la queue d'évènement
    - EventQueue
  - D'appeler les listeners correspondant à l'évènement
    - EventQueue.dispatchEvent()
  - D'afficher les composants (PaintEvent),
  - De gérer les timers (TimerEvent)
  - De permettre d'exécuter du code dans l'EDT :
    - EventQueue.invokeAndWait/invokeLater

EventQueue

- Queue d'évènement de l'AWT
- Gérer par la classe **java.awt.EventQueue**



EventQueue

- Obtenir la queue système :
  Toolkit.**getSystemEventQueue**()
- Retirer un évènement de la queue (bloquant)
  eventQueue.**getNextEvent**()
- Connaître le prochain évènement
  eventQueue.**peekEvent**()
- Prochain évènement d'un certain type (**eventId**)
  eventQueue.**peekEvent**(eventId)
- La thread courante est-elle l'EDT ?
  EventQueue.**isDispatchThread**()

EventQueue

- Envoyer un évènement
  eventQueue.**postEvent**(AWTEvent)
- Dispatcher l'évènement sur le composant
  protected eventQueue.**dispatchEvent**(AWTEvent)

- Il est possible de créer sa propre queue d'évènement et de transférer le contrôle des événements à celle-ci.
  - Transférer à une nouvelle queue d'évènement
    eventQueue.**push**(EventQueue)
  - Supprimer une queue d'évènement
    (rétablir la queue originale) eventQueue.**pop**()

Création de l'EDT

- Swing ne permet pas de crontrôler l'EDT, celle-ci est créée lorsque la fenêtre de l'application devient visible

```
public static void main(String[] args) {
  JFrame frame=new JFrame("DispatchThreadExample");
  JButton button=new JButton("Test");
  EventQueue.isDispatchThread(); // false

  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      EventQueue.isDispatchThread(); // true
    }
  });

  frame.setContentPane(button);
  frame.pack();
  frame.setVisible(true); // l'EDT est créée

  EventQueue.isDispatchThread(); // false
}
```

Gérer sa propre queue d'évènement

- Il es possible de prendre la main sur la queue d'évènement

```
public class MyQueue extends EventQueue {
  protected void dispatchEvent(AWTEvent event) {
    super.dispatchEvent(event);
  }
  public static void main(String[] args) throws InterruptedException {
    Toolkit toolkit=Toolkit.getDefaultToolkit();
    EventQueue queue=toolkit.getSystemEventQueue();

    MyQueue newQueue=new MyQueue();
    queue.push(newQueue);            // transfert

    JFrame frame=new JFrame("hello");
    frame.setSize(400,300);
    frame.setVisible(true);

    for(;;) {
      AWTEvent event=newQueue.getNextEvent(); // bloquant
      System.out.println(event);
      newQueue.dispatchEvent(event);
    }
  }
}
```

Traitement long

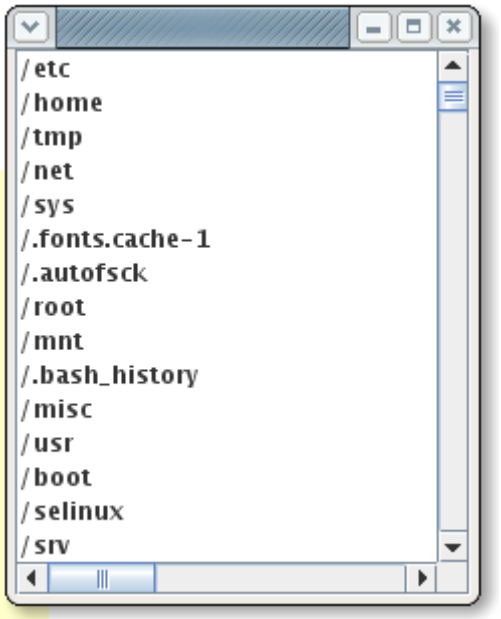
- Lorsque l'on veux effectuer des traitements en réponse à l'utilisateur, on effectue ces traitements dans un **listener** donc ceux-ci sont executés par **EventDispatchThread**

- Comme **EDT** s'occupe aussi du rafraîchissement de l'interface graphique => si le listener met 3min à finir, problème !

- Solution Intermédiaire : Faire les traitements long dans une **Thread**
- Problème : et si le traitement long modifie l'interface graphique, pas possible !!!

Exemple

- On souhaite afficher récursivement la liste de tous les fichiers à partir d'un répertoire en tâche de fond



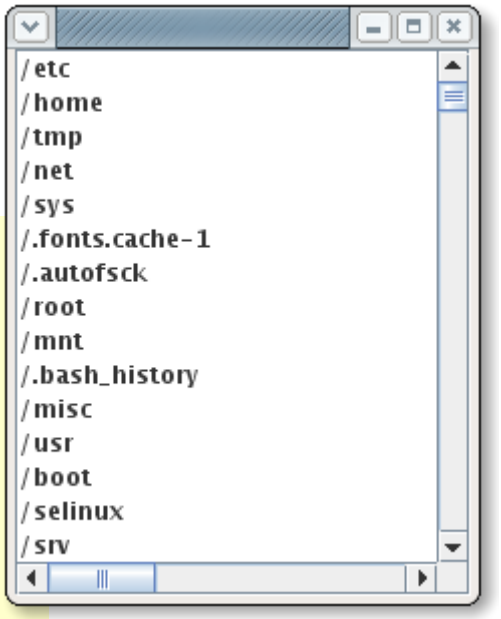

```
static void traverse(FileListModel model,File file) {
  final File[] files=file.listFiles();
  if (files==null) // si pas les droits
    return;
  for(File f:files) {
    if (f.isDirectory())
      traverse(model,f);

    // CE CODE EST FAUX !!!
    model.add(f);
  }
}

public static void main(String[] args) {
  final FileListModel model=new FileListModel();
  new Thread() {
    @Override public void run() {
      traverse(model,new File("/"));
    }
  }.start();
```

Exemple

- Le modèle de liste de fichiers permet l'insertion d'un fichier et informe la vue de l'ajout avec fireIntervalAdded.




```
public class FileListModel extends AbstractListModel {
  public int getSize() {
   return files.size();
  }

  public File getElementAt(int index) {
    return files.get(index);
  }

  public void addAll(File file) {
    int row=files.size();
    files.add(file);
    fireIntervalAdded(this,row,row);
  }

  private final ArrayList<File> files=new ArrayList<File>();
}
```

Traitement long (suite)

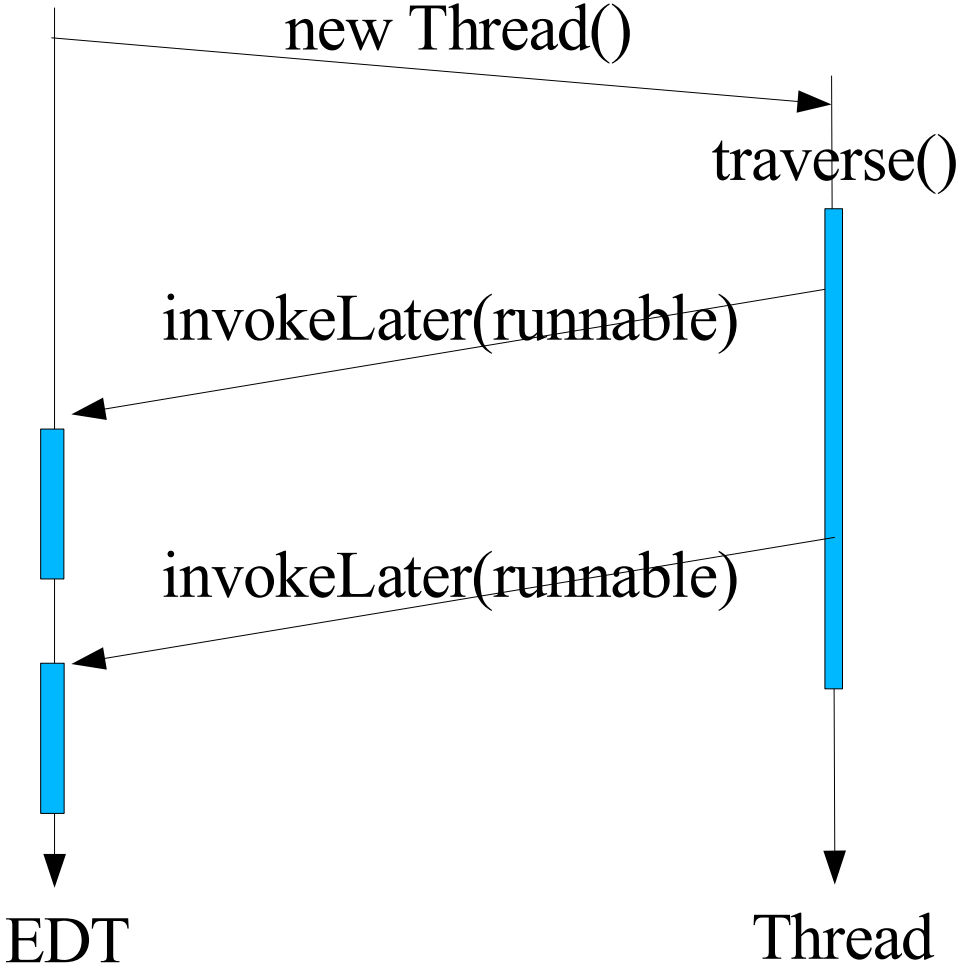
- Problème si le traitement long, qui s'exécute dans une **Thread,** veux changer l'interface graphique
  - Toute la partie SWING n'est pas **synchronized**
  - La partie AWT est synchonizé mais risque de **deadlock**

- Solution :
  utiliser les méthodes EventQueue.**invokeAndWait**() et EventQueue.**invokeLater**() pour demander d'exécuter un **Runnable** par l'EDT

Faire exécuter du code par l'EDT

- La thread qui effectue le traitement long va demander à l'EDT d'exécuter du code qui effectuera le raffraichissement graphique

- On utilise un objet Runnable pour spécifier le code à exécuter



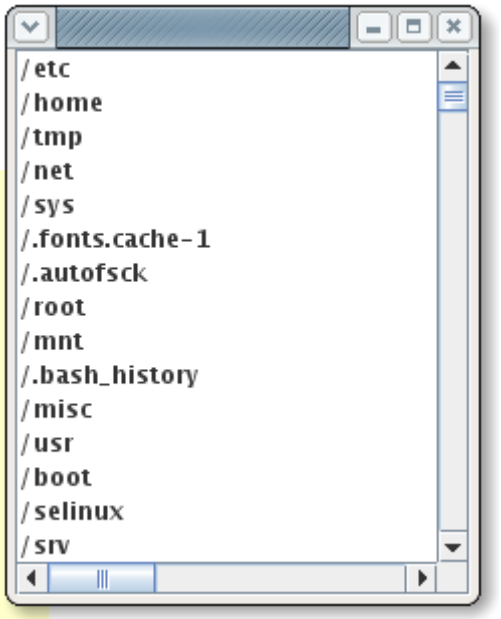
Difference entre les deux invoke*

- EventQueue.invokeLater and invokeAndWait permettent tous les deux à une thread externe de poster un événement qui contient du code à exécuter par l'EDT dans la queue
- **invokeLater**() poste le code et rend la main
  (envoie un message asynchrone)
- **invokeAndWait**() attend la fin de l'exécution du code avant de rendre la main
  (envoie un message synchrone)

- On utilise invokeAndWait() dans le cas où l'on veut être sûr que l'EDT à bien exécuter le code avant d'exécuter la suite

Exemple

- On souhaite afficher récursivement la liste de tous les fichiers à partir d'un répertoire en tâche de fond

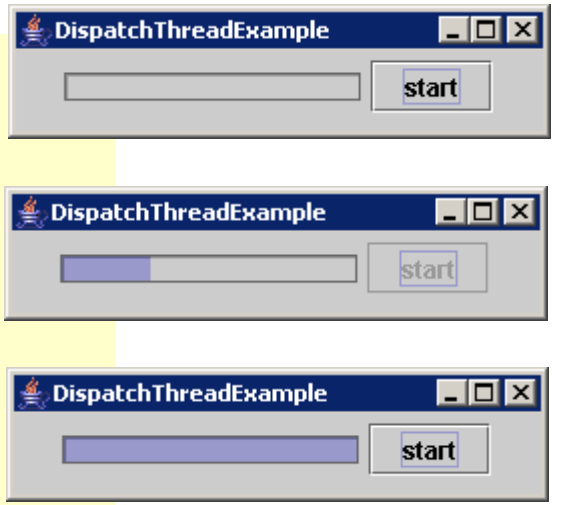



```
static void traverse(final FileListModel model,File file) {
  final File[] files=file.listFiles();
  if (files==null)
    return;
  for(File f:files)
    if (f.isDirectory())
      traverse(model,f);
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      model.addAll(files);
    }
  });
}
public static void main(String[] args) {
  final FileListModel model=new FileListModel();
  new Thread() {
    @Override public void run() {
      traverse(model,new File("/"));
    }
  }.start();
```

Autre exemple

- Afficher la progression d'un traitement long
- Comment fire pour réactiver le bouton start à la fin ?

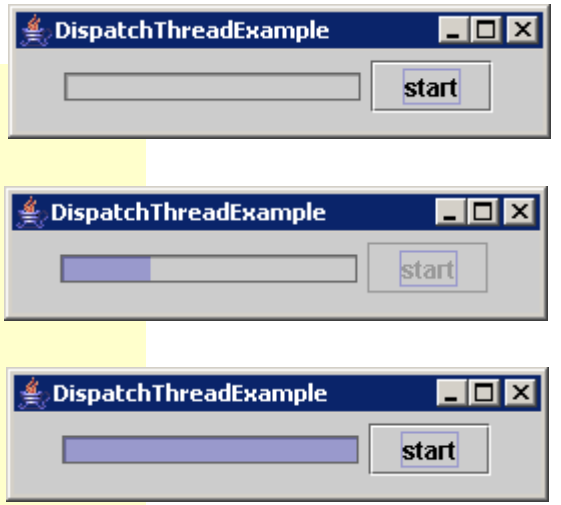
```
final JProgressBar progressBar=new JProgressBar(0,100);
final JButton button=new JButton("start");
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    button.setEnabled(false);
    new Thread() {
      public void run() {
        for(int i=1;i<=100;i++) {
          // truc long
          final int percent=i;
          EventQueue.invokeLater(new Runnable() {
            public void run() {
              progressBar.setValue(percent);
            }
          });
        }
        ... // to be continued
      }
    }.start(); } });
```



Réactivation du bouton start

- On utilise invokeAndWait() qui permet d'attendre que le code de l'interface graphique soit bien exécuté

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    button.setEnabled(false);
    new Thread() {
      public void run() {
        for(int i=1;i<=100;i++) {
          // truc long
          ...
        }
        try {
          EventQueue.invokeAndWait(new Runnable() {
            public void run() {
              button.setEnabled(true);
            }
          });
        } catch (   // plein d'exceptions
        ) {}
      }
    }.start(); } });
```



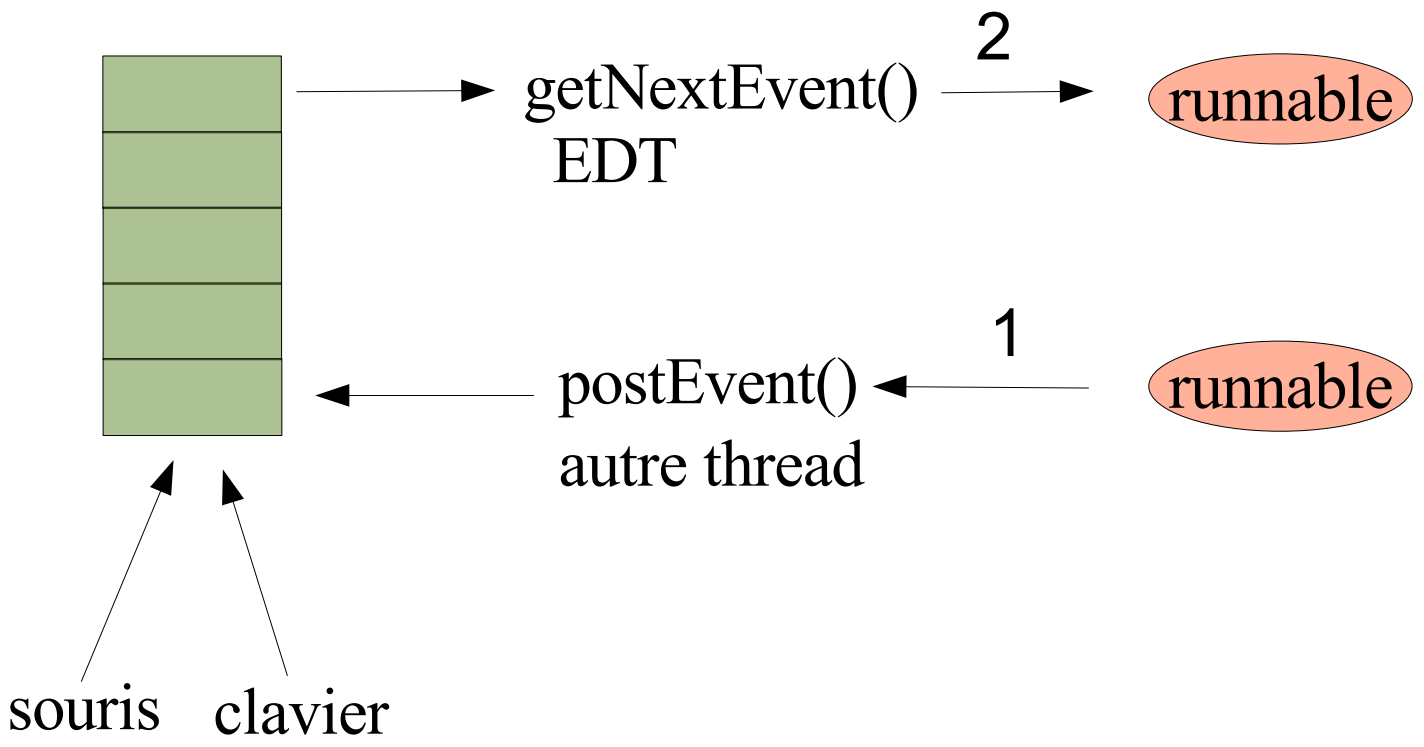
InvokeAndWait et Exception

- InterruptedException correspond au fait qu'un code est appelé interrupt() sur la tad courante lors de l'attente
- InvocationTargetException, une exception s'est produite dans le Runnable

```
try {
  EventQueue.invokeAndWait(new Runnable() {
    public void run() {
      button.setEnabled(true);
    }
  });
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
} catch (InvocationTargetException e) {
  Throwable cause= e.getCause();
  if (cause instanceof RuntimeException)
    throw (RuntimeException)cause;
  else
    if (cause instanceof Error)
      throw (Error)cause;
    else
      throw new AssertionError(cause);
}
```

EventQueue.invoke*

1. Le **Runnable** est encapsulé dans un événement et posté dans l'**EventQueue** (eventQueue.**postEvent**(event))
2. Lorsque l'on effectue un eventQueue.**dispatchEvent**() sur cet évènement, la méthode **run**() du **Runnable** est appelée

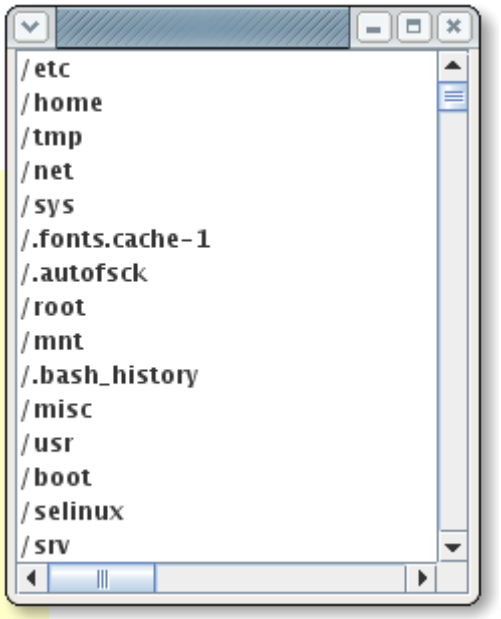


EventQueue.invoke*

- Problèmes des méthodes **EventQueue.invoke*** :
  - Comme on poste un événement il faut faire attention à ne pas trop en posté pour ne pas noyer les ‘vrais’ évènements (souris, clavier, rafraîchissements)
  - Lors du traitement de l’évènement par l’EDT, les données peuvent déjà être **caduques**
  - Le code est **asymétrique** (cf UI avancée)
  - Il faut gérer plein d’exceptions (surtout avec EventQueue.**invokeAndWait**())

Bulk operations

- Pour limiter les échanges et donc la synchronisation, on fait des affichages moins fréquent, 1 par répertoire et non 1 par fichier




```
static void traverse(final FileListModel model,File file) {
  final File[] files=file.listFiles();
  if (files==null)
    return;
  for(File f:files)
    if (f.isDirectory())
      traverse(model,f);
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      model.addAll(files);
    }
  });
}
public static void main(String[] args) {
  final FileListModel model=new FileListModel();
  new Thread() {
    @Override public void run() {
      traverse(model,new File("/"));
    }
  }.start();
```

Bulk operations

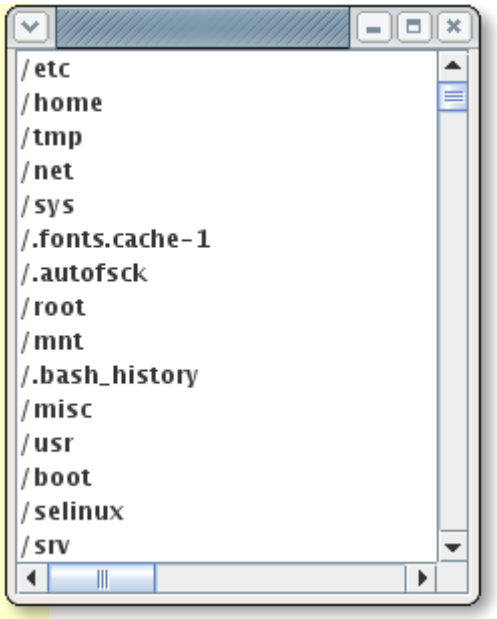
- Le modèle permet pour cela l'ajout de plusieurs fichiers

```
public class FileListModel extends AbstractListModel {
  public int getSize() {
   return files.size();
  }

  public File getElementAt(int index) {
    return files.get(index);
  }

  public void addAll(File... fs) {
    int row=files.size();
    Collections.addAll(files,fs);
    fireIntervalAdded(this,row,files.size()-1);
  }

  private final ArrayList<File> files=new ArrayList<File>();
}
```



Utilisation inhabituelle de invokeLater

- Si on change les listeners qui viennent de nous appeler

```
button.addActionListener(new ActionListener() {
  public void performed(ActionEvent event) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.gtk.GTKLookAndFeel");
        SwingUtilities.updateComponentTreeUI(frame);
      }
    });
  }
});
```

- Si l'on est plus royaliste que le roi

```
public static void main(String[] args) {
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      JFrame frame=new JFrame();
      frame.setContentPane(new JButton("Ok"));
      frame.setSize(400,300);
      frame.setVisible(true);
    }
  });
}
```

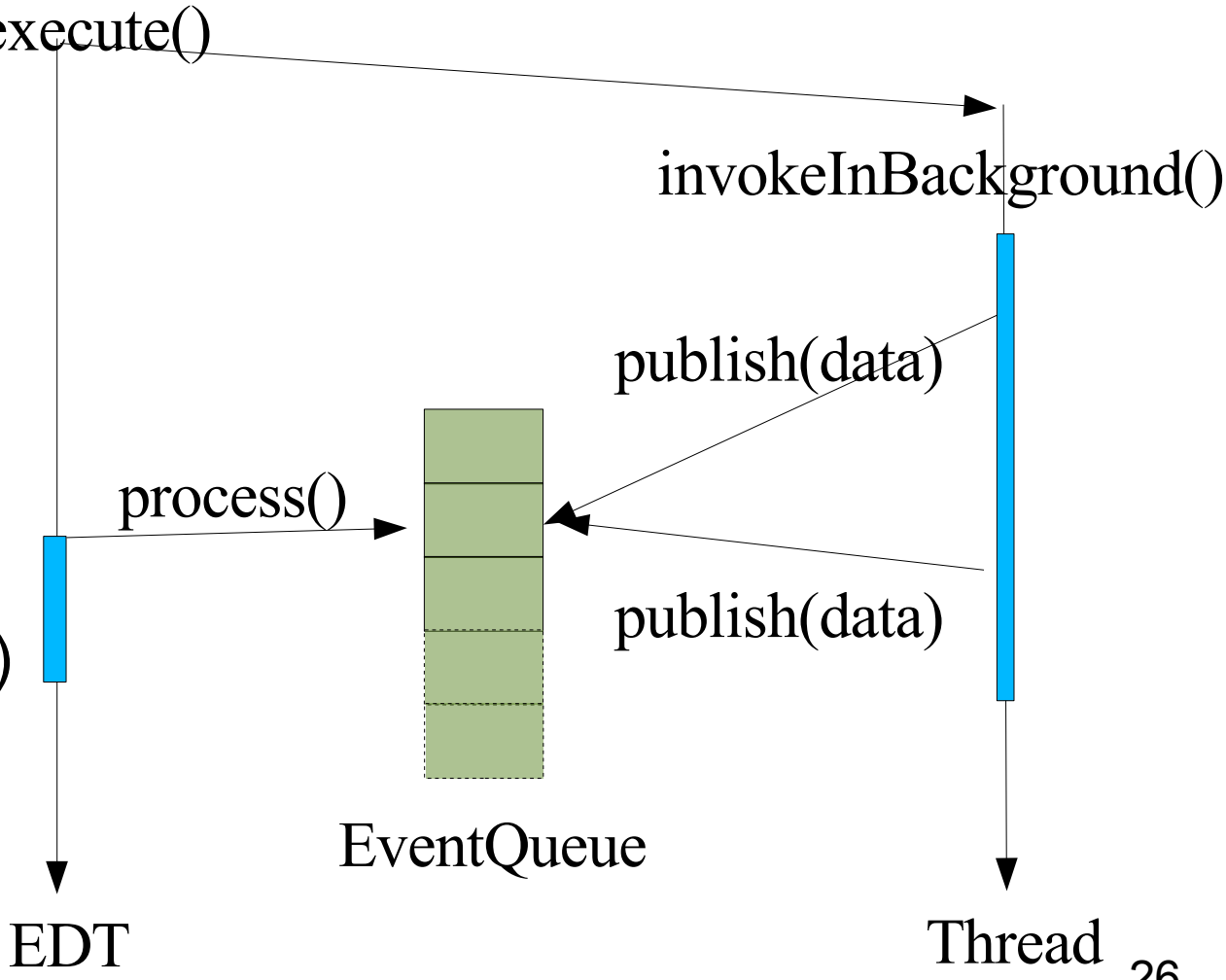
SwingWorker

- Lorsque le traitement long fait pas de rafraichissement, il est préférable d'utiliser javax.swing.**SwingWorker** (1.6)
- Comment ça marche
  - La thread main appel **execute**() pour lancé le worker thread et **get**() pour être blockée sur la réponse (comme les Future)
  - La worker thread execute la méthode **doInBackground**() par un executor (**ExecutorService**). Cette méthode peut appelée les méthodes **publish**() et/ou **setProgress**()
  - L'EDT appel la méthode **process**() suite à la méthode **publish**() pour effectuée les changement graphique puis la méthode **done**() à la fin.

SwingWorker

- Dans invokeInBackground() exécuté par une thread, on fait des appels à publish, si un evènement n'existe pas, on poste un nouveau, si l'évènement existe, on ajoute les données à l'évènement existant

- Lorsque l'EDT traite l'évènement, celle-ci appelle la méthode process() avec toute les données de l'évènement



SwingWorker

- SwingWorker<T,V> est paramétrée par T le type de retour de **doInbackground**() et **get**() et V le type des valeurs appelées entre **publish**() et **process**()
- Dans **doInBackground**()
  - Pour plusieurs appels à **publish**(), l'EDT sera appelée une fois avec toutes les valeurs des différents **publish**() (**coalescent**)
  - Pour plusieurs appels à **setProgress**(), les listeners enregistrés **addPropertyChangeListener**() ne seront appelée qu'avec la dernière valeur au moment de la lecture de l'evènement par l'EDT. (pas de valeur **caduque**)

Utilisation du SwingWorker

- En utilisant le SwingWorker

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
  final FileListModel model=new FileListModel();
  SwingWorker<Integer,File> worker=new SwingWorker<Integer,File>() {
    @Override protected Integer doInBackground() {
      return traverse(model,new File("c:\\"));
    }
    private int traverse(final FileListModel model,File file) {
      final File[] files=file.listFiles();
      if (files==null)
        return 0;
      int sum=0;
      for(File f:files)
        if (f.isDirectory())
          sum+=traverse(model,f);

      publish(files);
      return sum+files.length;
    }
    @Override protected void process(File... files) {
      model.add(files);
    }
  };
  worker.execute(); // utilise un executor
```

Utilisation du SwingWorker

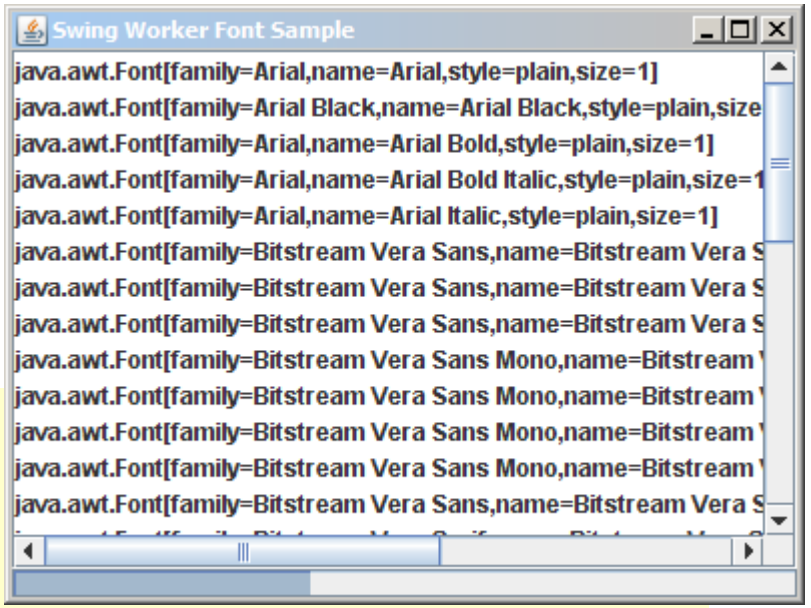
- Permet d'afficher le nombre de répertoires parcourus en demandant la valeur renvoyé par doInBackground()
- L'appel **get**() est bloquant tant que la worker thread n'a pas fini d'exécuter **doInbackground**()

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
  ...
  JList list=new JList(model);
  JFrame frame=new JFrame("Swing Worker Sample");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setContentPane(new JScrollPane(list));
  frame.setSize(400,300);
  frame.setVisible(true);

  System.out.println("files "+worker.get()); // bloquant
}
```

Et avec une progressbar

- On, veut avoir une progress bar indiquant le chargement progressif des fontes




```
class FontModel extends AbstractListModel {
   public int getSize() {
     return fontList.size();
   }
   public Font getElementAt(int index) {
     return fontList.get(index);
   }
   public void addAll(List<? extends Font> fonts) {
     int firstRow=fontList.size();
     fontList.addAll(fonts);
     fireIntervalAdded(this,firstRow,firstRow+fonts.size());
   }
   private final ArrayList<Font> fontList=new ArrayList<Font>();
}
```

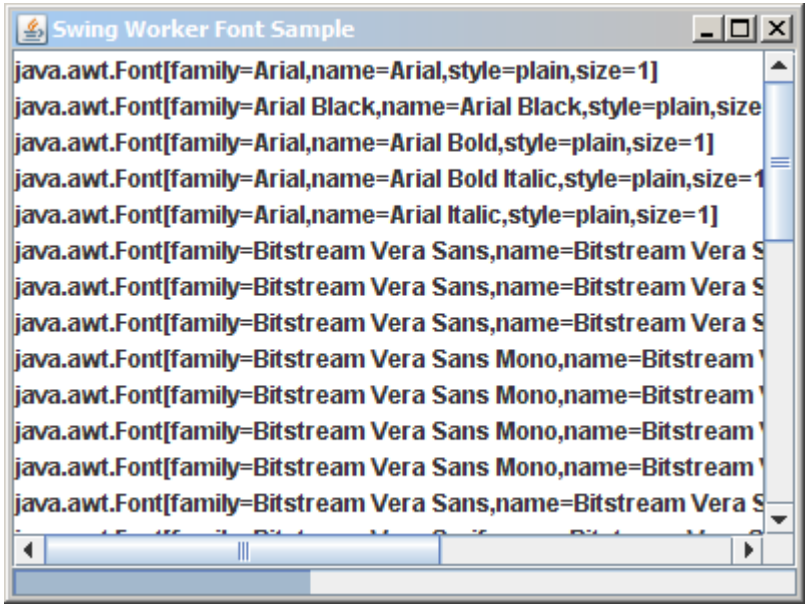
Et avec une progressbar

```
final JProgressBar bar=new JProgressBar(0,100);
SwingWorker<Void,Font> worker=new SwingWorker<Void,Font>() {
  @Override
  protected Void doInBackground() {
    Font[] fonts=env.getAllFonts();
    for(int i=0;i<fonts.length;i++) {
      publish(fonts[i].deriveFont(11));
      setProgress((i+1)*100/fonts.length);  // on indique une progression

      try {
        Thread.sleep(100);  // juste pour ralentir l'exemple
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
    return null;
  }
  @Override
  protected void process(List<Font> fonts) {
    model.addAll(fonts);
  }
};
...
worker.execute();
```

Et avec une progressbar

- Progress est une propriété lié (java bean) il est donc possible de s'enregistrer en tant que PropertyChangeListener sur celle-ci.




```
final JProgressBar bar=new JProgressBar(0,100);
SwingWorker<Void,Font> worker=new SwingWorker<Void,Font>() {
   ...
};
worker.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent evt) {
    if (!"progress".equals(evt.getPropertyName()))
      return;
    bar.setValue((Integer)evt.getNewValue());
  }
});
worker.execute();
```

Méthodes qui poste des évènements

- InvokeLater et invokeAndWait ne sont pas les seules méthodes à poster des évènements :
- Les demandes :
  - d'affichage
    Component.repaint()
  - de prise en compte de modification de la hiérarchie
    Component.validate(),
    Jcomponent.revalidate()
  - D'exécution périodique (timer)

  postent des évènements dans la queue d'évènement
- Ces méthodes sont donc **threadsafe** et peuvent être appelées sans invokeLater/invokeAndWait par une autre thread que l'EDT.

Demande de raffraichissement

- Il existe deux façon de demander le raffraichissement graphique d'un composant
  - Le traitement direct,
    dans le cas ou l'on est dans l'EDT
    Component.paintImmediately()
  - Le traitement en postant un PaintEvent,
    donc possible avec une autre Thread que l'EDT
    Component.repaint()

- Dans le cas de repaint, les évènements PaintEvent sont coalescents, plusieurs événements peuvent être regroupé en un seul.

Digital Clock avec repaint()

- On raffraichit l'horloge toute les secondes avec repaint()

```
public class DigitalClock extends JComponent {
  @Override protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.clearRect(0,0,getWidth(),getHeight());
    Graphics2D g2d=(Graphics2D)g;
    g2d.setFont(font);
    FontRenderContext context=g2d.getFontRenderContext();
    GlyphVector gv=font.createGlyphVector(context,new Date().toString());
    float y=(float)((getHeight()-gv.getVisualBounds().getHeight())/2.0);
    g2d.drawGlyphVector(gv,0f,y);
  }
```

```
final DigitalClock clock=new DigitalClock();
new Thread(new Runnable() {
  public void run() {
    Thread currentThread=Thread.currentThread();
    for(;!currentThread.isInterrupted();) {
      clock.repaint();

      try {
        Thread.sleep(999);
      } catch (InterruptedException e) {
        currentThread.interrupt();
      }
  } }
}).start();
```

Avec paintImmediately

- On raffraichit l'horloge toute les secondes avec paintImmediately

```
final DigitalClock clock=new DigitalClock();
new Thread(new Runnable() {
  public void run() {
    Thread currentThread=Thread.currentThread();
    for(;!currentThread.isInterrupted();) {
      EventQueue.invokeLater(new Runnable() {
        public void run() {
          clock.paintImmediately(0,0,clock.getWidth(),clock.getHeight());
        }
      });

      try {
        Thread.sleep(999);
      } catch (InterruptedException e) {
        currentThread.interrupt();
      }
    }
  }
}).start();
```

- Dans ce cas les événements ne sont pas coalescents, car ils sont dus à invokeLater() !

repaint() vs paintImmediately()

- Si on est hors de l'EDT
  - **repaint**() est la seule option possible
- Si on est dans l'EDT, cela dépend
  - Si on permet à l'utilisateur de faire plusieurs changement sur une classe qui vont avoir des impacts graphiques
    - On utilise **repaint**() pour profiter de la coalescence des évènements
  - Sinon
    - On utilise **paintImmediately**()
- **paintImmediately**() n'est en général pas très utilisée

Le timer swing

- Swing possède un mécanisme de timer (javax.swing.Timer) qui exécute la méthode **actionPerformed** d'un **ActionListener** dans l'EDT
- Le timer poste un événement TimerEvent lorsqu'un timer est déclenché, le temps (*delay*) entre deux déclenchements est paramétrable
- Constructeur:
  Timer(int delay, ActionListener listener)
- Utilise le mécanisme de listener classique
  add/removeActionListener()
- Le timer peut être démarré ou arrêté par n'importe quelle thread
  start()/stop()

Autres méthodes du timer swing

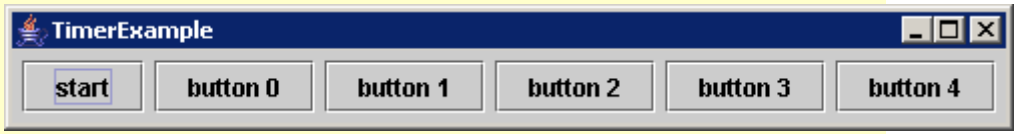
- Indique si le temier est actif
  isRunning()
- Fixe le delai avant un premier appel aux listeners
  get/setInitialDelay(int initialDelay)
- Fixe le delai entre deux appels aux listeners
  get/setDelay(int delay)
- Indique si le timer doit répéter les appels aux listeners
  is/setRepeat(boolean repeat)
- Regroupe les délais dépassés
  is/setCoalescece(boolean coalesce)
- Tous les délais sont fixés en millisecondes
  de temps « réel ».

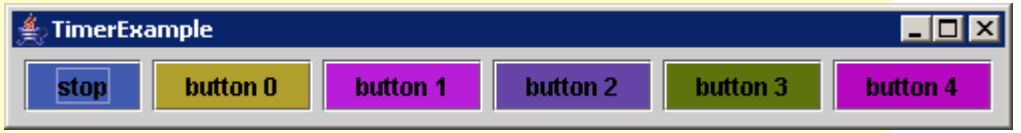
Exemple

- On fait cycler les couleurs des boutons

```
JFrame frame=new JFrame("TimerExample");
final JPanel panel=new JPanel();
final Timer timer=new Timer(300,new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    for(int i=0;i<panel.getComponentCount();i++)
      panel.getComponent(i).setBackground(getRandomColor());
  }
});
final JButton timerButton=new JButton("start");
timerButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if (timer.isRunning())
      timer.stop();
    else
      timer.start();

    timerButton.setText((timer.isRunning())?"stop":"start");
  }
});
panel.add(timerButton);
for(int i=0;i<5;i++)
  panel.add(new JButton("button "+i));
```





Digital Clock avec un timer

- En reprenant l'exemple de l'horloge digital

```
final DigitalClock clock=new DigitalClock();
new Timer(999,new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    clock.paintImmediately(0,0,clock.getWidth(),clock.getHeight());
  }
}).start();

JFrame frame=new JFrame("Clock");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(clock);
frame.setSize(400,300);
frame.setVisible(true);
```