SAPIENZA Università di Roma
Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica ed Automatica
Corso di Laurea in Ingegneria dei Sistemi Informatici

Esercitazioni di
PROGETTAZIONE DEL SOFTWARE
A.A. 2010-2011

Threads e Concorrenza in Java Swing

Alessandro Russo

Introduzione

- Per la realizzazione di interfacce grafiche complesse in `Java` tramite `Swing` è fondamentale considerare le problematiche derivanti dalla **concorrenza** e dal **multithreading**

- Un'applicazione che utilizza i componenti della libreria `Swing` deve essere realizzata in modo da sfruttare la concorrenza per creare una interfaccia utente che sia sempre **reattiva** ed in grado di gestire l'interazione con l'utente (*responsiveness*)

- Ma che relazione c'è tra il framework `Swing` e i thread? Come sfruttare al meglio la concorrenza e il multithreading?

Swing e Thread

La realizzazione di interfacce utente tramite `Swing` coinvolge tre tipologie di thread:

1. i **thread iniziali** (*initial threads*): i thread che eseguono il codice iniziale dell'applicazione (es. il `main`) e/o i thread creati dal programmatore

2. l'**event dispatch thread (EDT)**: il thread che esegue il codice di gestione degli eventi e gestisce l'interazione con l'utente (vedi dopo)

3. i **worker thread** (o **background thread**): thread dedicati che eseguono operazioni "lunghe" (*time-consuming*) in background, in modo da non compromettere la reattività del thread che gestisce l'interazione con l'utente (cioè l'EDT)

In generale il programmatore non deve creare esplicitamente questi thread:

- sono forniti dall'ambiente *runtime* di `Java` (il thread iniziale) o dalla libreria `Swing` (l'event dispatch thread e i worker thread)

- ovviamente il programmatore può creare ed utilizzare thread aggiuntivi (cfr. `Thread` e `Runnable`)

I Thread Iniziali

- In ogni programma `Java` la logica applicativa viene gestita da un insieme di thread

- Tipicamente la logica applicativa ha inizio in un solo thread: il thread che invoca ed esegue il metodo `main`

- Spesso in un'applicazione che utilizza i componenti `Swing` il thread iniziale ha il compito di creare ed inizializzare l'interfaccia utente (la GUI, Graphical User Interface); l'evoluzione del programma dipende poi dagli *eventi* generati dall'utente che interagisce con l'applicazione

Esempio 1 - Il Thread Iniziale

La classe MyFrame

```
package esempio1;

import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Esempio 1 - Thread Iniziale");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JButton bottone = new JButton("Premi!");
    // ...
    // crea e inizializza componenti, impostazione layout, impostazione
    // listener...
    // ...
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(bottone);
    System.out.println(Thread.currentThread().getName()); // Stampa "main"
    pack();
    setVisible(true);
  }

  public static void main(String[] args) {
    System.out.println(Thread.currentThread().getName()); // Stampa "main"
    new MyFrame();
  }
}
```

L'Event Dispatch Thread (1/2)

- L'**Event Dispatch Thread (EDT)** è il thread che:
  - esegue il *drawing* dei componenti grafici (cioè "disegna" e mostra l'interfaccia utente)
  - esegue il codice per la **gestione degli eventi** generati dall'interazione tra i componenti grafici e l'utente (pressione di bottoni, ridimensionamento di finestre, click del mouse, etc.)
- L'EDT (anche noto come **AWT-thread**) è un thread di sistema che:
  - viene avviato automaticamente dalla Java VM alla prima chiamata di un metodo `paint()` o `repaint()`
  - gestisce una *coda* di eventi e resta in attesa che nuovi eventi da gestire siano inseriti nella coda
  - preleva gli eventi dalla coda, li notifica ai `Listener` corrispondenti ed esegue il codice di gestione degli eventi

L'Event Dispatch Thread (2/2)

In un'applicazione con interfaccia `Swing` la "vita" dell'EDT contribuisce a determinare la persistenza del programma:

- L'event dispatch thread è un thread *non demone*

  → persiste finché c'è almeno un evento da gestire nella coda o finché almeno una finestra si trova nello stato *visualizzabile* (vedi più avanti)

- La Java VM resta attiva finché è attivo almeno un thread non demone

  → a meno di altri thread, la Java VM (e quindi l'applicazione) resta attiva finché resta attivo l'EDT

Una volta che l'interfaccia grafica creata tramite *Swing* diventa visibile, la maggior parte delle operazioni viene gestita ed eseguita dall'event dispatch thread

- il codice definito in metodi come `actionPerformed`, `mouseClicked`, `windowIconified` e in generale in tutti i metodi di gestione di eventi per i quali è stato definito un ascoltatore (`ActionListener`, `MouseListener`, etc.) viene eseguito dall'EDT

  → per verificare se una porzione di codice è eseguita dall'event dispatch thread è possibile invocare il metodo statico booleano

  ```
  SwingUtilities.isEventDispatchThread()
  ```

Esempio 2 - EDT

La classe MyFrame

```
package esempio2;

import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Esempio 2 - Thread Iniziale");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JButton bottone = new JButton("Premi!");
    bottone.addActionListener(new MyListener());
    // ...
    // crea e inizializza componenti, impostazione layout, impostazione listener
    // ...
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(bottone);
    System.out.println(Thread.currentThread().getName()); // Stampa "main"
    pack();
    setVisible(true);
  }

  public static void main(String[] args) {
    System.out.println(Thread.currentThread().getName()); // Stampa "main"
    new MyFrame();
  }
```

```
}
```

La classe MyListener

```
package esempio2;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyListener implements ActionListener {

  public void actionPerformed(ActionEvent ev) {
    System.out.println(SwingUtilities.isEventDispatchThread()); // true
    System.out.println(Thread.currentThread().getName()); // non e' main!
    // E' AWT-EventQueue-0
    JOptionPane.showMessageDialog(null, "Hai premuto il bottone!");
  }
}
```

The Single-Thread Rule

REGOLA: *Ogni operazione che consiste nella visualizzazione*[*]*, modifica o aggiornamento di un componente* `Swing`*, o che accede allo stato del componente stesso, deve essere eseguita nell'event dispatch thread.*

→ gran parte dei metodi che operano su oggetti della libreria `Swing` non sono *thread-safe*: se invocati da più thread concorrenti posso portare a *deadlock* o ad errori di consistenza della memoria

→ i componenti `Swing` possono essere acceduti da un solo thread alla volta, in particolare dall'event dispatch thread

- Il fatto che il codice di gestione degli eventi generati dall'interazione con l'utente sia eseguito dall'EDT garantisce che la regola venga rispettata

- E se il mio programma non rispetta la regola? Nella maggior parte dei casi funzionerà correttamente, ma può portare ad errori imprevedibili e non sempre riproducibili

  → meglio rispettare la regola :-)

[*]un componente è considerato *visualizzato* quando il corrispondente metodo `paint` è stato invocato; un componente che rappresenta una finestra è *visualizzato* quando su di esso viene invocato il metodo `setVisible(true)`, `show()` oppure `pack()`, e si considerano *visualizzati* tutti i componenti contenuti nella finestra

The Single-Thread Rule: Alcune Eccezioni

In alcuni casi specifici è possibile interagire con componenti `Swing` utilizzando thread diversi dall'event dispatch thread.

- Alcuni metodi associati a componenti `Swing` sono *thread-safe* e sono esplicitamente segnalati nella documentazione (API) dalla dicitura *"This method is thread safe, although most Swing methods are not."*

  → ad esempio, il metodo `append(String str)` di `JTextArea`

- L'interfaccia utente `Swing` può essere creata e visualizzata nel thread principale (`main`)

  → come negli esempi presentati finora nel corso

- I seguenti metodi di `JComponent` possono essere invocati da qualsiasi thread: `repaint()`, `revalidate()` e `invalidate()`

  → `repaint()` e `revalidate()` chiedono all'EDT di invocare `paint()` e `validate()`; il metodo `invalidate()` segnala che un componente e i componenti che lo contengono richiedono di essere validati

- Le liste di `Listener` possono essere modificate da qualsiasi thread tramite i metodi `add`*ListenerType*`Listener` e `remove`*ListenerType*`Listener` methods

The Single-Thread Rule: Motivazioni

Perché i componenti `Swing` non sono *thread-safe*? Perché i progettisti della libreria `Swing` hanno effettuato questa scelta?

1. **Prestazioni:** realizzare componenti *thread-safe* avrebbe introdotto costi aggiuntivi per la sincronizzazione e la gestione dei lock, riducendo la velocità e la reattività della libreria

2. **Complessità:** programmare con librerie *thread-safe* è complesso e si incorre facilmente in deadlock

3. **Testabilità e determinismo:** applicazioni con componenti che supportano accessi da thread multipli sono difficili da testare (o addirittura impossibili in alcuni casi...) e il multithreading avrebbe generato un'interleaving impredicibile nella gestione di eventi

4. **Estendibilità:** estendere componenti *thread-safe* avrebbe richiesto ai programmatori di creare nuovi componenti che fossero a loro volta *thread-safe*, e ciò richiede padronanza assoluta della programmazione multithread

$\rightarrow$ limitando l'accesso ad un solo thread (l'EDT), la libreria `Swing` offre prestazioni elevate, non richiede che il programmatore abbia conoscenze avanzate di multithreading, garantisce una gestione sequenziale degli eventi (tramite la coda gestita dall'EDT) ed è facilmente estendibile

Interagire con l'Event Dispatch Thread (1/2)

Una volta che l'interfaccia utente `Swing` viene visualizzata, tutte le interazioni che derivano da *eventi* (mouse click, pressione bottoni, etc.) sono gestite dall'event dispatch thread

→ Ma se si ha la necessità di interagire con i componenti grafici `Swing` in base ad una logica applicativa che non dipende da eventi associati ai componenti stessi? Come rispettare la *Single-Thread Rule*?

Si considerino i seguenti casi:

1. Applicazioni che, prima di poter essere utilizzate dall'utente, eseguono lunghe operazioni di inizializzazione

   → tipicamente viene costruita e mostrata una GUI (es. splashscreen) mentre avviene l'inizializzazione; al termine dell'inizializzazione la GUI viene poi modificata o aggiornata

2. Applicazioni la cui interfaccia deve essere aggiornata o modificata al verificarsi di eventi non generati da componenti `Swing`

   → ad esempio, un'applicazione server in cui la GUI viene aggiornata alla ricezione di una richiesta da parte di un'applicazione client

La modifica o l'aggiornamento della GUI (che richiedono di modificare componenti `Swing`) NON devono essere eseguiti da parte di thread diversi dall'EDT!

Interagire con l'Event Dispatch Thread (2/2)

La classe `SwingUtilities` (del package `javax.swing`) offre due metodi statici che consentono a qualsiasi thread di interagire con l'EDT in modo che esegua del codice definito dal programmatore

→ il codice da eseguire (che tipicamente crea, modifica, o aggiorna componenti `Swing`) deve essere definito nel metodo `run()` di un oggetto `Runnable`

1. `invokeLater(Runnable target)`
   - consente di richiedere che il codice definito nel `run()` del `Runnable` passato come argomento venga eseguito dall'event dispatch thread
   - il metodo ritorna immediatamente, senza attendere che l'EDT esegua il codice richiesto (invocazione ed esecuzione *asincrona*)
     → il thread che invoca il metodo può continuare immediatamente la sua esecuzione

2. `invokeAndWait(Runnable target)`
   - consente di richiedere che il codice definito nel `run()` del `Runnable` passato come argomento venga eseguito dall'event dispatch thread
   - il metodo attende che il codice venga eseguito dall'EDT e poi ritorna (invocazione ed esecuzione *sincrona*)
     → il thread che invoca il metodo resta in attesa che l'EDT esegua il codice richiesto

Dettagli su `invokeLater()` e `invokeAndWait()`

Quando un thread invoca i metodi `invokeLater(Runnable target)` e `invokeAndWait(Runnable target)`

- il riferimento all'istanza di `Runnable` passato come argomento viene inserito nella *coda* gestita dall'event dispatch thread
- l'EDT continua a processare gli eventi in attesa (*pending*) nella coda (se presenti)
- quando incontra un `Runnable` nella coda, l'EDT ne esegue il metodo `run()`

NOTE

- non viene creato un nuovo `Thread` per ogni oggetto `Runnable` inserito nella coda dalle invocazioni di `invokeLater(Runnable target)` e `invokeAndWait(Runnable target)`

  → è l'event dispatch thread stesso che esegue il codice definito dal programmatore nel `run()` nel `Runnable`

- evitare di invocare `invokeLater()` e soprattutto `invokeAndWait()` dall'event dispatch thread

  → l'EDT ovviamente non ha bisogno di invocare questi metodi per agire su componenti `Swing`

  → l'invocazione di `invokeAndWait()` dall'EDT genera
  `java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread`
  l'EDT non può attendere se stesso :-)

Esempio 3a - Inizializzazione

Semplice applicazione che:

- crea e visualizza una GUI con una label e tre bottoni

  → inizialmente i bottoni sono disabilitati e la label mostra un messaggio che informa l'utente che è in corso l'inizializzazione dell'applicazione

- esegue le operazioni di inizializzazione

- al termine dell'inizializzazione, aggiorna la GUI

  → i bottoni vengono abilitati e la label mostra un messaggio che informa l'utente che l'inizializzazione è stata completata

Esempio 3a

La classe MyFrame

```
package esempio3a;

import java.awt.*;
import javax.swing.*;
import java.lang.reflect.InvocationTargetException;

public class MyFrame extends JFrame {

  private JLabel statusLabel;
  private JButton button1, button2, button3;

  public MyFrame() {
    super("Esempio 3");
    System.out.println("Sono il thread " + Thread.currentThread().getName()
        + " - Creo la GUI");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    statusLabel = new JLabel("Sto inizializzando l'applicazione, attendere...");
    JPanel centerPanel = new JPanel();
    button1 = new JButton("Bottone 1");
    button1.setEnabled(false);
    button2 = new JButton("Bottone 2");
    button2.setEnabled(false);
    button3 = new JButton("Bottone 3");
    button3.setEnabled(false);
    centerPanel.add(button1);
    centerPanel.add(button2);
```

```
  centerPanel.add(button3);
  getContentPane().add(statusLabel, BorderLayout.PAGE_START);
  getContentPane().add(centerPanel, BorderLayout.CENTER);
  pack();
  setLocationRelativeTo(null);
  setVisible(true);
  // da questo momento in poi solo l'EDT puo' aggiornare la GUI!!!
  inizializzaApplicazione();
}

private void inizializzaApplicazione() {
  try {
    // simulo attivita' di inizializzazione
    System.out.println("Sono il thread " + Thread.currentThread().getName()
        + " - Inizializzo l'applicazione");
    Thread.sleep(10000);
  } catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
  // inizializzazione completata...dovrei aggiornare la GUI, cambiando il
  // testo della label e abilitando i bottoni...ma sono nel thread main, non
  // posso farlo! Posso farlo solo tramite l'EDT!
  Runnable target = new Runnable() {

    @Override
    public void run() {
      statusLabel.setText("Inizializzazione completata!!!");
      button1.setEnabled(true);
      button2.setEnabled(true);
      button3.setEnabled(true);
```

```
      }
    };
    System.out.println("Sono il thread " + Thread.currentThread().getName()
        + " - Chiamo invokeAndWait()");
    try {
      SwingUtilities.invokeAndWait(target);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    } catch (InvocationTargetException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    System.out.println("Sono il thread " + Thread.currentThread().getName()
        + " - L'EDT ha aggiornato la GUI :-)");
  }

  public static void main(String[] args) {
    new MyFrame();
  }

}
```

Esempio 3b - Server

Semplice applicazione che rappresenta un Server che riceve messaggi da un Client

- il Server ha una GUI con una label che mostra il numero di messaggi ricevuti, una barra di avanzamento il cui stato dipende dal numero di messaggi ricevuti, ed un'area di testo in cui vengono visualizzati i messaggi ricevuti

- ogni volta che il Server riceve un messaggio:

  $\rightarrow$ il messaggio viene visualizzato nell'area di testo

  $\rightarrow$ la label con il numero di messaggi ricevuti viene aggiornata

  $\rightarrow$ la barra di avanzamento viene aggiornata

Esempio 3b

La classe Server

```
package esempio3b;

import java.awt.*;
import javax.swing.*;

public class Server extends JFrame {

  private JProgressBar barraMessaggi;
  private JLabel numeroMessaggi;
  private JTextArea areaMessaggi;
  private int messaggiRicevuti = 0;

  public Server() {
    super("Server");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JPanel pannelloSuperiore = new JPanel();
    JLabel label = new JLabel("Messaggi ricevuti: ");
    numeroMessaggi = new JLabel(messaggiRicevuti + "");
    barraMessaggi = new JProgressBar(0, 30);
    pannelloSuperiore.add(label);
    pannelloSuperiore.add(numeroMessaggi);
    pannelloSuperiore.add(barraMessaggi);
    areaMessaggi = new JTextArea(10, 50);
    JScrollPane scrollPane = new JScrollPane(areaMessaggi);
    areaMessaggi.setEditable(false);
    getContentPane().add(pannelloSuperiore, BorderLayout.PAGE_START);
```

```
    getContentPane().add(scrollPane, BorderLayout.CENTER);
    pack();
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public void nuovoMessaggio(String messaggio) {
    messaggiRicevuti++;
    // aggiungo il nuovo msg alla textArea...posso farlo direttamente perche
    // append e' thread-safe!
    areaMessaggi.append("Messaggio " + messaggiRicevuti + ": " + messaggio
        + "\n");
    final int msgNum = messaggiRicevuti;
    // aggiorno il numero di messaggi e la barra: devo farlo tramite l'EDT!!!
    Runnable target = new Runnable() {
      @Override
      public void run() {
        areaMessaggi.setCaretPosition(areaMessaggi.getText().length());
        numeroMessaggi.setText(msgNum + "");
        barraMessaggi.setValue(msgNum);
      }
    };
    SwingUtilities.invokeLater(target);
  }
}
```

La classe Client

```
package esempio3b;
```

```
public class Client {

  private Server server;

  public Client(Server server) {
    this.server = server;
  }

  public void avvia() {
    int n;
    for (int i = 0; i < 30; i++) {
      n = i + 1;
      server.nuovoMessaggio("Questo e' il messaggio numero " + n);
      try {
        // solo per far vedere i messaggi che arrivano
        Thread.sleep((int) (10.0 * Math.random() + 1) * 50);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
  }

  public static void main(String[] args) {
    Server server = new Server();
    try {
      Thread.sleep(5000); // piccola pausa
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
```

```
    Client client = new Client(server);
    client.avvia();
  }
}
```

Creare la GUI nell'Event Dispatch Thread

Tra le eccezioni alla *Single-Thread Rule* c'è la possibilità di creare i componenti `Swing` della GUI nel thread principale (o in un altro thread)

→ tuttavia, sempre più spesso si preferisce adottare un'approccio in cui la creazione, inizializzazione e visualizzazione della GUI avviene nell'event dispatch thread

In accordo con la *Single-Thread Rule* si ha che il thread principale

- crea un oggetto `Runnable` nel cui metodo `run()` avviene la creazione e visualizzazione della GUI

- interagisce con l'EDT tramite `invokeLater()` o `invokeAndWait()` affinché la creazione e visualizzazione della GUI avvenga nell'EDT

Esempio 3c

```
package esempio3c;

import javax.swing.*;

public class MyFrame extends JFrame {

  public MyFrame() {
    super("Esempio");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // ...
    // creo e aggiungo altri componenti
    // ...
    pack();
    setVisible(true);
  }

  public static void main(String[] args) {
    Runnable init = new Runnable() {
      public void run() {
        new MyFrame();
      }
    };
    SwingUtilities.invokeLater(init);
  }
}
```

Event Dispatch Thread: Osservazioni

Riassumendo quanto visto finora:

- i componenti forniti dalla libreria `Swing`, in generale, non sono *thread-safe*
- secondo la *Single-Thread Rule* ogni aggiornamento, modifica ed accesso a componenti `Swing` deve essere eseguito dall'EDT
- il codice di gestione degli eventi AWT viene sempre eseguito dall'EDT
- per eseguire un aggiornamento, modifica o accesso a componenti `Swing` da un thread qualsiasi è necessario utilizzare i metodi `invokeLater()` e `invokeAndWait()` affinché le operazioni siano eseguite dall'EDT

L'event dispatch thread processa sequenzialmente gli eventi e le richieste di altri thread (corrispondenti alle invocazioni di `invokeLater()` e `invokeAndWait()`) contenute nella *coda* che esso gestisce

→ le operazioni eseguite dall'EDT devono essere brevi e non-bloccanti!

→ operazioni molto lunghe (es. computazioni complesse, attività di I/O, etc.) non devono mai essere eseguite dall'EDT!

EDT e Long-Running o Blocking Tasks

Se l'EDT è impegnato nell'esecuzione di attività lunghe e complesse o nell'esecuzione di codice con istruzioni bloccanti (es. `wait()` o metodi con istruzioni di sincronizzazione)

→ l'applicazione appare "congelata" (*freezed*) e sembra non rispondere ai comandi dell'utente

Perchè?

→ poiché l'EDT è impegnato o bloccato, gli eventi si accumulano nella *coda* e non vengono processati

PROBLEMA: se al verificarsi di un evento originato dall'interazione con un componente `Swing` (es. pressione di un bottone) è necessario eseguire operazioni lunghe e complesse (*long-running task*), qual è la metodologia da adottare?

- il codice di gestione dell'evento viene eseguito dall'EDT

- le operazioni lunghe e complesse non possono essere eseguite dall'EDT per non rischiare il *freezing* dell'applicazione

Esempio 4a - Blocco della GUI

La classe MyFrame

```
package esempio4a;

import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

public class MyFrame extends JFrame {

  JTextArea areaTesto;
  JTextField campoTesto;

  public MyFrame() {
    super("Esempio 4 - Freezing");
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    MyListener listener = new MyListener(this);

    JLabel label = new JLabel("Inserisci testo: ");
    campoTesto = new JTextField(20);
    JButton enterButton = new JButton("Enter");
    enterButton.addActionListener(listener);
    enterButton.setActionCommand("enter");
    JPanel pannelloSuperiore = new JPanel();
    pannelloSuperiore.add(label);
    pannelloSuperiore.add(campoTesto);
    pannelloSuperiore.add(enterButton);
```

```
areaTesto = new JTextArea(20, 30);
JScrollPane scrollCentrale = new JScrollPane(areaTesto);

JTree tree = new JTree(buildTree());
tree.getSelectionModel().setSelectionMode(
    TreeSelectionModel.SINGLE_TREE_SELECTION);
JScrollPane treeView = new JScrollPane(tree);

JButton azione1 = new JButton("Azione 1");
azione1.addActionListener(listener);
azione1.setActionCommand("azione1");
JButton azione2 = new JButton("Azione 2");
azione2.addActionListener(listener);
azione2.setActionCommand("azione2");
JButton freeze = new JButton("Premi!");
freeze.addActionListener(listener);
freeze.setActionCommand("freeze");
JPanel pannelloInferiore = new JPanel();
pannelloInferiore.add(azione1);
pannelloInferiore.add(azione2);
pannelloInferiore.add(freeze);

JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
    treeView, scrollCentrale);

Container container = getContentPane();
container.add(pannelloSuperiore, BorderLayout.PAGE_START);
container.add(splitPane, BorderLayout.CENTER);
container.add(pannelloInferiore, BorderLayout.PAGE_END);
```

```
  pack();
  setLocationRelativeTo(null);
  setVisible(true);
}

private DefaultMutableTreeNode buildTree() {
  DefaultMutableTreeNode top = new DefaultMutableTreeNode("Corso Java");
  DefaultMutableTreeNode nodoJava = new DefaultMutableTreeNode("Lezioni");
  DefaultMutableTreeNode nodoCodice = new DefaultMutableTreeNode("Codice");
  nodoJava.add(new DefaultMutableTreeNode("Lezione 1.pdf"));
  nodoJava.add(new DefaultMutableTreeNode("Lezione 2.pdf"));
  nodoJava.add(new DefaultMutableTreeNode("Lezione 3.pdf"));
  nodoJava.add(new DefaultMutableTreeNode("Lezione 4.pdf"));
  nodoJava.add(new DefaultMutableTreeNode("Lezione 5.pdf"));
  nodoJava.add(new DefaultMutableTreeNode("Lezione 6.pdf"));
  nodoCodice.add(new DefaultMutableTreeNode("Applicazione.java"));
  nodoCodice.add(new DefaultMutableTreeNode("Test.java"));
  nodoCodice.add(new DefaultMutableTreeNode("GUI.java"));
  nodoCodice.add(new DefaultMutableTreeNode("MyFrame.java"));
  top.add(nodoJava);
  top.add(nodoCodice);
  return top;
}

public static void main(String[] args) {
  Runnable init = new Runnable() {
    public void run() {
      new MyFrame();
    }
  };
  // creo la GUI nell'EDT
```

```
    SwingUtilities.invokeLater(init);
  }
}
```

La classe MyListener

```
package esempio4a;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/*
 * NOTA: nell'implementazione che segue viene mostrato un esempio di gestione NON
 * corretta di eventi che determinano l'esecuzione di attivita' di lunga durata
 */
public class MyListener implements ActionListener {

  private MyFrame frame;

  public MyListener(MyFrame frame) {
    this.frame = frame;
  }

  @Override
  public void actionPerformed(ActionEvent e) { // eseguito dall'EDT
    String command = e.getActionCommand();
    if (command == "enter") {
      if (!frame.campoTesto.getText().equals("")) {
        frame.areaTesto.append(frame.campoTesto.getText() + "\n");
      }
```

```
      frame.campoTesto.setText("");
    } else if (command == "azione1") {
      frame.areaTesto.append("Hai premuto il bottone Azione 1!\n");
    } else if (command == "azione2") {
      frame.areaTesto.append("Hai premuto il bottone Azione 2!\n");
    } else if (command == "freeze") {
      frame.areaTesto
          .append("Hai premuto il bottone per l'esecuzione di un'attivita' lunga...\n");
      // simulo esecuzione attivita' lunga...
      try {
        // NOTA: questo codice viene eseguito dall'EDT...l'applicazione si
        // blocca!
        // questo e' un ERRORE!
        Thread.sleep(20000);
        frame.areaTesto.append("Attivita' completata!\n");
      } catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
      }
    }
  }
}
```

Esempio 4b - Blocco della GUI

Semplice applicazione che:

- consente di inserire un numero di conto ed il relativo PIN
- mostra un bottone di ricerca e un bottone per cancellare la ricerca (inizialmente disabilitato)

Quando viene premuto il bottone di ricerca, l'applicazione:

- legge il numero di conto ed il PIN inseriti
- si connette al server della banca per ottenere il credito presente sul conto (operazione che può richiedere diversi secondi...)

Durante la ricerca il bottone di ricerca viene disabilitato e il bottone di cancellazione viene abilitato; al termine della ricerca:

- il credito viene mostrato in un campo di testo
- il bottone di ricerca viene nuovamente abilitato e quello di cancellazione viene disabilitato

Che succede se l'operazione di ricerca viene eseguita dall'event dispatch thread?

Esempio 4b

La classe SearchFrame

```
package esempio4b;

import java.awt.*;
import javax.swing.*;

public class SearchFrame extends JFrame {

  JTextField numeroConto;
  JPasswordField numeroPin;
  JTextField valoreCredito;
  JButton cerca;
  JButton cancella;

  public SearchFrame() {
    super("Conto Corrente");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLabel conto = new JLabel("Numero conto: ");
    numeroConto = new JTextField(10);
    JLabel pin = new JLabel("PIN: ");
    numeroPin = new JPasswordField(5);
    JLabel credito = new JLabel("Credito: ");
    valoreCredito = new JTextField(7);
    valoreCredito.setEditable(false);
    JPanel pannelloSuperiore = new JPanel();
    pannelloSuperiore.add(conto);
    pannelloSuperiore.add(numeroConto);
```

```
  pannelloSuperiore.add(pin);
  pannelloSuperiore.add(numeroPin);
  pannelloSuperiore.add(credito);
  pannelloSuperiore.add(valoreCredito);

  cerca = new JButton("Cerca");
  cancella = new JButton("Cancella");
  cancella.setEnabled(false);
  MyListener listener = new MyListener(this);
  cerca.setActionCommand("cerca");
  cerca.addActionListener(listener);
  cancella.setActionCommand("cancella");
  cancella.addActionListener(listener);
  JPanel pannelloInferiore = new JPanel();
  pannelloInferiore.add(cerca);
  pannelloInferiore.add(cancella);

  getContentPane().add(pannelloSuperiore, BorderLayout.PAGE_START);
  getContentPane().add(pannelloInferiore, BorderLayout.PAGE_END);

  pack();
  setLocationRelativeTo(null);
  setVisible(true);
}

public static void main(String[] args) {
  Runnable init = new Runnable() {
    public void run() {
      new SearchFrame();
    }
  };
```

```
    // creo la GUI nell'EDT
    SwingUtilities.invokeLater(init);
  }

}
```

La classe MyListener

```
package esempio4b;

import java.awt.event.*;

public class MyListener implements ActionListener {

  private SearchFrame frame;

  public MyListener(SearchFrame frame) {
    this.frame = frame;
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command == "cerca") {
      cerca();
    } else { // command e' "cancella"
      cancella();
    }
  }
```

```
  private void cerca() {
    String numConto = frame.numeroConto.getText();
    String pin = new String(frame.numeroPin.getPassword());
    frame.cerca.setEnabled(false);
    frame.cancella.setEnabled(true);
    String creditoResiduo = ricercaContoInBanca(numConto, pin);
    frame.valoreCredito.setText(creditoResiduo);
    frame.cerca.setEnabled(true);
    frame.cancella.setEnabled(false);
  }

  private String ricercaContoInBanca(String numConto, String pin) {
    if (numConto.equals("") || pin.equals(""))
      return "";
    try {
      // Simulo connessione a server della banca...
      // Questo codice viene eseguito dall'EDT: l'applicazione si blocca!
      // E' un errore!
      Thread.sleep(10000);
      // credito ricevuto dalla banca
      return "1,234.56";
    } catch (InterruptedException ex) {
      return "Ricerca annullata";
    }
  }

  private void cancella() {
    System.out.println("Cancella ricerca...");
    // qui c'e' il codice per cancellare la richiesta in corso...
    // ma il bottone per cancellare non puo' essere mai premuto :-(
  }
}
```

Long-Running o Blocking Tasks: Soluzione

PROBLEMA: se al verificarsi di un evento originato dall'interazione con un componente `Swing` (es. pressione di un bottone) è necessario eseguire operazioni lunghe e complesse o bloccanti (*long-running* o *blocking task*), qual è la metodologia da adottare?

→ se tutto viene eseguito dall'EDT si rischia il blocco (*freezing*) della GUI e l'applicazione non è reattiva (e l'utente non è per nulla contento...)

SOLUZIONE: le operazioni lunghe e complesse o bloccanti devono essere eseguite in altri thread (**worker threads**)!

→ l'EDT lancia il nuovo thread e può così continuare a processare e gestire gli eventi, mantenendo la reattività della GUI

ATTENZIONE: se il thread avviato dall'EDT ha bisogno di aggiornare componenti grafici durante la sua esecuzione e/o al termine dell'operazione lunga o bloccante, non può farlo direttamente

→ la *Single-Thread Rule* deve sempre essere rispettata e quindi eventuali aggiornamenti e modifiche di componenti grafici devono essere eseguiti dall'EDT

Worker Thread e Attività in Background

In generale, le attività da eseguire al verificarsi di un evento (es. pressione di un bottone) richiedono di:

1. accedere a componenti `Swing` per leggere input inseriti dall'utente (es. il testo di un `JTextField`, le `JCheckBox` selezionate) → da eseguire nell'EDT

2. modificare o aggiornare la GUI (es. abilitare/disabilitare bottoni, creare barra di avanzamento) → da eseguire nell'EDT

3. avviare ed eseguire la computazione (es. connessione ad un server, ricerca in base di dati, download di un file) → da eseguire in *background* in un thread separato (worker thread)

4. aggiornare la GUI per mostrare l'avanzamento della computazione (es. aggiornare barra di avanzamento, mostrare anteprima di file scaricati) → da eseguire nell'EDT

5. aggiornare o modificare la GUI al termine della computazione per mostrare i risultati (es. risultati della ricerca, foto scaricate) → da eseguire nell'EDT

Creare manualmente un thread che esegue la computazione e interagisce con l'event dispatch thread per gli aggiornamenti e modifiche della GUI può essere complesso

→ a partire dal JDK 1.6 Java mette a disposizione del programmatore la classe `SwingWorker` del package `javax.swing`

Attività in Background: la Classe `SwingWorker`

- La classe `SwingWorker` consente di gestire ed eseguire in *background* attività lunghe e complesse o bloccanti che necessitano di interagire con la GUI `Swing` al termine della computazione e/o durante il processamento

- È una classe *astratta* che implementa l'interfaccia `RunnableFuture` (wrapper per le interfacce `Runnable` e `Future`) e permette di:
  - definire le attività e la computazione da eseguire in background
  - produrre aggiornamenti sull'avanzamento della computazione e comunicare risultati intermedi all'event dispatch thread per aggiornare la GUI
  - ritornare all'EDT un oggetto che rappresenta il risultato della computazione
  - eseguire nell'EDT il codice di aggiornamento della GUI al termine della computazione
  - definire *bound properties*: proprietà (variabili di istanza) che quando modificate dal worker thread causano l'invio di un evento all'EDT

- È inoltre possibile interagire con uno `SwingWorker` per interrompere e cancellare l'esecuzione

Usare la Classe SwingWorker

La classe SwingWorker è definita come

```
public abstract class SwingWorker<T,V> extends Object implements RunnableFuture
```

- T è il tipo del risultato prodotto dall'esecuzione del worker thread
- V è il tipo dei risultati intermedi prodotti durante l'esecuzione

Utilizzare la classe SwingWorker: elementi di base

- estendere la classe SwingWorker definendo il tipo del risultato finale e dei risultati intermedi che verranno prodotti

```
public class EsecutoreAttivita extends SwingWorker<ImageIcon, Integer> {...}
```

- eseguire l'overriding del metodo protected abstract T doInBackground() che deve contenere il codice per l'esecuzione dell'attività lunga o bloccante e ritornare un oggetto dello stesso tipo dichiarato nella definizione della classe

```
protected ImageIcon doInBackground() {
  // esecuzione attivita' lunga o bloccante
  // ritorna ImageIcon
}
```

- per avviare uno SwingWorker invocare su di esso il metodo execute()

```
public void actionPerformed(ActionEvent evt) {
  EsecutoreAttivita worker = new EsecutoreAttivita();
  worker.execute();
}
```

Risultato Finale e Risultati Intermedi (1/2)

Al termine dell'esecuzione del metodo `doInBackground()`:

- il worker thread che ha eseguito il `doInBackground()` invoca il metodo `done()`
- il risultato della computazione può essere ottenuto invocando sull'oggetto `SwingWorker` il metodo `get()`

  → NOTA: se invocato prima del completamento della computazione, il metodo `get()` è bloccante)

Per eseguire attività specifiche (tipicamente, aggiornare la GUI con il risultato della computazione) al termine dell'esecuzione dell'attività lunga o bloccante:

→ eseguire l'overriding del metodo `done()`, il quale

- viene invocato automaticamente al completamento di `doInBackground()`
- viene eseguito nell'**event dispatch thread** ⇒ è possibile aggiornare la GUI!
- accede al risultato della computazione invocando il metodo `get()`

```
protected void done() { // eseguito nell'EDT
  try {
    ImageIcon result = get(); // il risultato di doInBackground
    // aggiorna la GUI
  } catch(Exception e) {...}
}
```

Risultato Finale e Risultati Intermedi (2/2)

Durante l'esecuzione del metodo `doInBackground()` uno `SwingWorker` può produrre risultati intermedi della computazione che devono essere usati per aggiornare la GUI (ad esempio, se si sta scaricando un insieme di immagini, si vuole che ogni immagine venga visualizzata appena viene scaricata)

→ all'interno del metodo `doInBackground()` invocare il metodo `publish()` specificando uno o più risultati intermedi della computazione (del tipo specificato nella definizione della classe)

→ eseguire l'overriding del metodo `process(List<V> chunks)`, il quale

- riceve una lista di risultati intermedi prodotti e notificati da un'invocazione di `publish()`
- viene eseguito nell'**event dispatch thread** ⇒ è possibile aggiornare la GUI!

```
protected ImageIcon doInBackground() {
  while(condizione) {
    // produci risultato intermedio di tipo Integer
    publish(resIntermedio);
  }
  // ritorna risultato finale ImageIcon
}

protected void process(List<Integer> chunks) { // eseguito nell'EDT
  //aggiorna GUI con risultati intermedi
}
```

Interrompere l'Esecuzione

Per segnalare ad uno `SwingWorker` che l'attività in esecuzione in background deve essere interrotta

→ invocare sull'oggetto `SwingWorker` il metodo `cancel(boolean mayInterruptIfRunning)`

- il parametro booleano consente di specificare se il worker thread deve essere interrotto (con `InterruptException`) nel caso in cui sia in attesa su una istruzione bloccante (es. `wait()` o I/O di rete)

→ nel metodo `doInBackground()` è necessario verificare periodicamente tramite il metodo `isCancelled()` se sia stata ricevuta una richiesta di interrompere l'esecuzione (tramite `cancel()`)

```
protected ImageIcon doInBackground() {
  while(!isCancelled()) { // controllo periodicamente se bisogna interrompere l'esecuzione
    // produci risultato intermedio di tipo Integer
    publish(resIntermedio);
  }
  // ritorna risultato finale ImageIcon
}
```

→ è opportuno verificare se sia stata ricevuta una richiesta di interrompere l'esecuzione anche nei metodi `done()` e `process()`

Gestire lo Stato di uno `SwingWorker` (1/2)

Ogni oggetto della classe `SwingWorker` è caratterizzato da due variabili di stato:

1. la variabile `progress`

   → valore intero compreso tra 0 e 100 che rappresenta lo stato di avanzamento dell'attività eseguita dal worker thread

   → il valore può essere letto o scritto tramite appositi metodi (`getProgress()` e `setProgress(in progress)`)

   → tipicamente, il valore viene aggiornato durante l'esecuzione di `doInBackground()` e viene letto per aggiornare eventuali componenti grafici che dipendono dallo stato di avanzamento (es. barra di progresso)

2. la variabile `state`

   → indica lo stato in cui si trova il worker thread durante il suo ciclo di vita

   - `PENDING`: stato iniziale dello `SwingWorker` prima dell'invocazione di `doInBackground()`
   - `STARTED`: stato dello `SwingWorker` durante l'esecuzione di `doInBackground()`
   - `DONE`: stato dello `SwingWorker` dopo l'esecuzione di `doInBackground()`

   → può essere letta tramite il metodo `getState()`

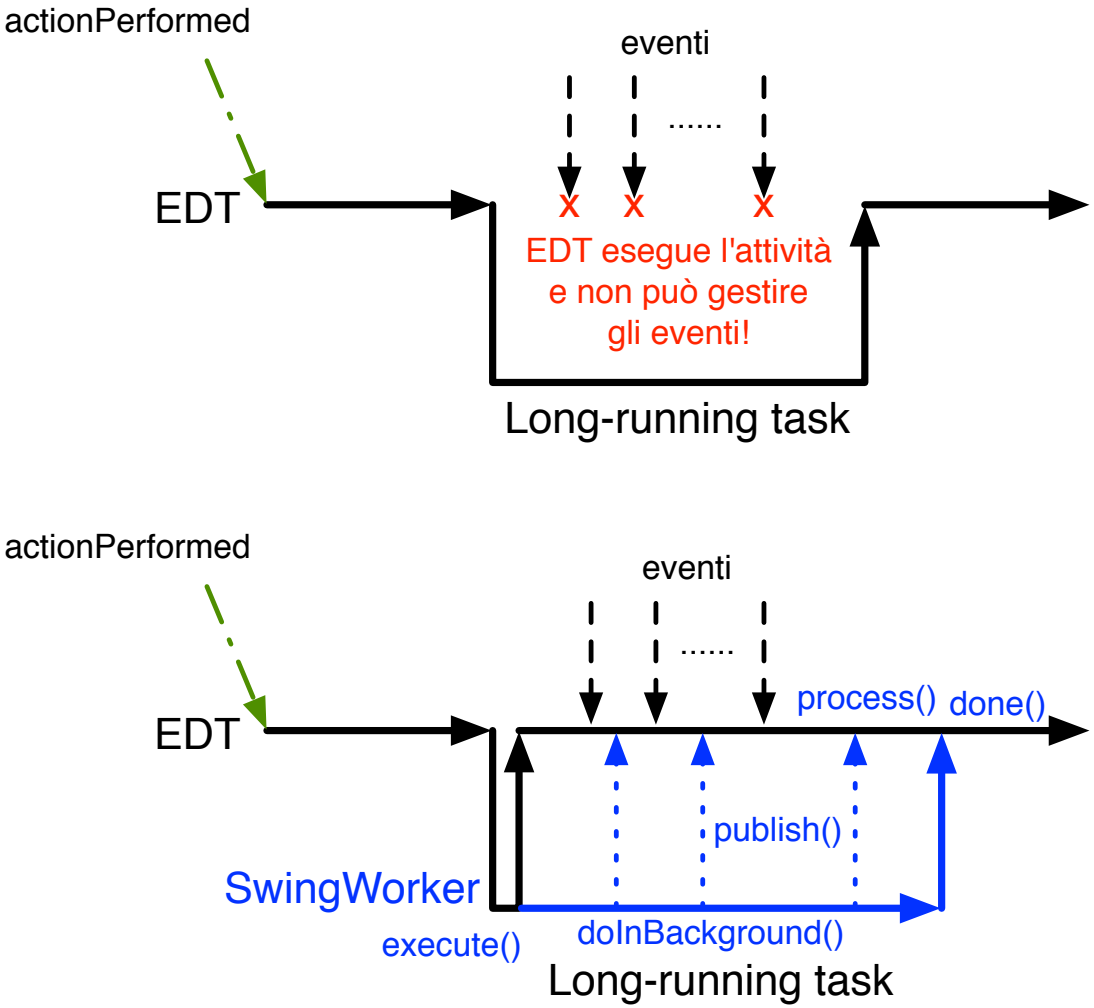
Gestire lo Stato di uno `SwingWorker` (2/2)

NOTA: ogni volta che il valore delle variabili `progress` e `state` cambia, viene generato un evento di notifica per gli eventuali listeners registrati

→ per monitorare lo stato di uno `SwingWorker` ed eseguire azioni specifiche (es. aggiornare una barra di avanzamento) in seguito ad un cambio di stato è necessario implementare e registrare un `PropertyChangeListener`

→ si veda l'esempio 5b per i dettagli

EDT e Worker Threads



Esempio 5a - SwingWorker

Realizziamo l'Esempio 4a utilizzando la classe `SwingWorker`.

Il thread che effettuerà la ricerca in background:

- produce come risultato finale una stringa che rappresenta il credito presente sul conto
- non produce risultati intermedi da processare
- deve gestire opportunamente le richieste di cancellazione

Esempio 5a

La classe SearchFrame

```
package esempio5a;

import java.awt.*;
import javax.swing.*;

public class SearchFrame extends JFrame {

  JTextField numeroConto;
  JPasswordField numeroPin;
  JTextField valoreCredito;
  JButton cerca;
  JButton cancella;
  JLabel statoRicerca;

  public SearchFrame() {
    super("Conto Corrente");
    System.out.println("Ciao, sono " + Thread.currentThread().getName()
        + " e creo la GUI");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLabel conto = new JLabel("Numero conto: ");
    numeroConto = new JTextField(10);
    JLabel pin = new JLabel("PIN: ");
    numeroPin = new JPasswordField(5);
    JLabel credito = new JLabel("Credito: ");
    valoreCredito = new JTextField(7);
    valoreCredito.setEditable(false);
```

```
JPanel pannelloSuperiore = new JPanel();
pannelloSuperiore.add(conto);
pannelloSuperiore.add(numeroConto);
pannelloSuperiore.add(pin);
pannelloSuperiore.add(numeroPin);
pannelloSuperiore.add(credito);
pannelloSuperiore.add(valoreCredito);

cerca = new JButton("Cerca");
cancella = new JButton("Cancella");
cancella.setEnabled(false);
MyListener listener = new MyListener(this);
cerca.setActionCommand("cerca");
cerca.addActionListener(listener);
cancella.setActionCommand("cancella");
cancella.addActionListener(listener);
JPanel pannelloBottoni = new JPanel();
pannelloBottoni.add(cerca);
pannelloBottoni.add(cancella);

JLabel labelRicerca = new JLabel("Stato: ");
JPanel pannelloStato = new JPanel();
statoRicerca = new JLabel("Inserire i dati");
pannelloStato.add(labelRicerca);
pannelloStato.add(statoRicerca);

JPanel pannelloInferiore = new JPanel(new BorderLayout());
pannelloInferiore.add(pannelloBottoni, BorderLayout.PAGE_START);
pannelloInferiore.add(pannelloStato, BorderLayout.PAGE_END);

getContentPane().add(pannelloSuperiore, BorderLayout.PAGE_START);
```

```
    getContentPane().add(pannelloInferiore, BorderLayout.PAGE_END);

    pack();
    setLocationRelativeTo(null);
    setVisible(true);
  }

  public static void main(String[] args) {
    System.out.println("Ciao, sono " + Thread.currentThread().getName()
        + " e chiedo all'EDT di creare la GUI");
    Runnable init = new Runnable() {
      public void run() {
        new SearchFrame();
      }
    };
    // creo la GUI nell'EDT
    SwingUtilities.invokeLater(init);
  }

}
```

La classe MyListener

```
package esempio5a;

import java.awt.event.*;

public class MyListener implements ActionListener {

  private SearchFrame frame;
```

```
private SearchWorker worker;

public MyListener(SearchFrame frame) {
  this.frame = frame;
}

@Override
public void actionPerformed(ActionEvent e) { // eseguito da EDT
  String command = e.getActionCommand();
  if (command == "cerca") {
    cerca();
  } else { // command e' "cancella"
    cancella();
  }
}

private void cerca() { // eseguito da EDT
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + ": aggiorno la GUI e attivo lo SwingWorker");
  String numConto = frame.numeroConto.getText();
  String pin = new String(frame.numeroPin.getPassword());
  if (numConto.equals("") || pin.equals("")) {
    return;
  }
  frame.cerca.setEnabled(false);
  frame.cancella.setEnabled(true);
  frame.statoRicerca.setText("Ricerca in corso...");
  // creo ed avvio SwingWorker che eseguira' la ricerca
  worker = new SearchWorker(numConto, pin, frame);
  worker.execute();
}
```

```
  private void cancella() {
    worker.cancel(true);
  }
}
```

La classe SearchWorker

```
package esempio5a;

import java.util.concurrent.ExecutionException;
import javax.swing.*;

/*
 * SwingWorker che ritorna una stringa come risultato finale
 * e non produce risultati intermedi (Void)
 */
public class SearchWorker extends SwingWorker<String, Void> {

  private String numeroConto;
  private String pin;
  private SearchFrame frame;

  public SearchWorker(String conto, String pin, SearchFrame frame) {
    numeroConto = conto;
    this.pin = pin;
    this.frame = frame;
  }

  @Override
```

```
protected String doInBackground() throws Exception {
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + " ed eseguo doInBackground()");
  if (!isCancelled()) {
    try {
      // Simulo connessione a server della banca...
      // Questo codice viene eseguito da un worker thread in background
      Thread.sleep(10000);
      // credito ricevuto dalla banca
      return "" + (int) (10000.0 * Math.random());
    } catch (InterruptedException ex) {
      System.out.println("Ciao, sono " + Thread.currentThread().getName()
          + " e sono stato interrotto mentre eseguivo doInBackground()");
      return "";
    }
  } else {
    return "";
  }
}

@Override
protected void done() { // chiamato dopo doInBackground ed eseguito dall'EDT
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + " ed eseguo done()");
  frame.cerca.setEnabled(true);
  frame.cancella.setEnabled(false);
  if (isCancelled()) {
    frame.statoRicerca.setText("Ricerca annullata");
  } else {
    frame.statoRicerca.setText("Ricerca completata!");
    String result;
```

```
      try {
        result = get();
        frame.valoreCredito.setText(result);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      } catch (ExecutionException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
  }

}
```

Esempio 5b - SwingWorker

Semplice applicazione che alla pressione di un tasto simula il download di un insieme di immagini

- ogni volta che un'immagine viene scaricata, essa viene immediatamente visualizzata
- al termine del download di tutte le immagini viene visualizzato in un apposito campo il numero totale di KB scaricati
- l'avanzamento viene mostrato tramite una barra di progresso
- in ogni momento è possibile interrompere il download

Il thread `SwingWorker` che effettuerà il download in background:

- produce come risultato finale un intero che rappresenta i KB totali scaricati
- produce risultati intermedi che rappresentano le immagini scaricate
- aggiorna lo stato di avanzamento (`progress`) ogni volta che una nuova immagine viene scaricata
- deve gestire opportunamente le richieste di cancellazione

Esempio 5b

La classe FrameImmagini

```
package esempio5b;

import java.awt.*;
import javax.swing.*;

public class FrameImmagini extends JFrame {

  JTextField campoTesto;
  JLabel download;
  JButton scarica, cancel;
  JTextArea areaTesto;
  JPanel pannelloCentrale;
  JProgressBar barra;

  public FrameImmagini() {
    super("Immagini");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    JLabel label = new JLabel("Inserisci testo: ");
    campoTesto = new JTextField(20);
    JButton testoButton = new JButton("Enter");
    testoButton.setActionCommand("enter");
    MyListener listener = new MyListener(this);
    testoButton.addActionListener(listener);
    JPanel pannelloSuperiore = new JPanel();
    pannelloSuperiore.add(label);
    pannelloSuperiore.add(campoTesto);
```

```
pannelloSuperiore.add(testoButton);

JLabel img1 = new JLabel("1", JLabel.CENTER);
JLabel img2 = new JLabel("2", JLabel.CENTER);
JLabel img3 = new JLabel("3", JLabel.CENTER);
JLabel img4 = new JLabel("4", JLabel.CENTER);
JLabel img5 = new JLabel("5", JLabel.CENTER);
JLabel img6 = new JLabel("6", JLabel.CENTER);
JLabel img7 = new JLabel("7", JLabel.CENTER);
JLabel img8 = new JLabel("8", JLabel.CENTER);
JLabel img9 = new JLabel("9", JLabel.CENTER);
pannelloCentrale = new JPanel(new GridLayout(3, 3));
pannelloCentrale.add(img1);
pannelloCentrale.add(img2);
pannelloCentrale.add(img3);
pannelloCentrale.add(img4);
pannelloCentrale.add(img5);
pannelloCentrale.add(img6);
pannelloCentrale.add(img7);
pannelloCentrale.add(img8);
pannelloCentrale.add(img9);

areaTesto = new JTextArea(5, 25);
areaTesto.setEditable(false);
JScrollPane scroll = new JScrollPane(areaTesto);
scarica = new JButton("Scarica!");
scarica.setActionCommand("scarica");
cancel = new JButton("Annulla");
cancel.setEnabled(false);
cancel.setActionCommand("cancel");
scarica.addActionListener(listener);
```

```
  cancel.addActionListener(listener);
  download = new JLabel("--");
  JLabel labelScaricati = new JLabel("KB scaricati: ");
  barra = new JProgressBar();
  JPanel pannelloBottoni = new JPanel();
  pannelloBottoni.add(scarica);
  pannelloBottoni.add(cancel);
  pannelloBottoni.add(labelScaricati);
  pannelloBottoni.add(download);
  pannelloBottoni.add(barra);
  JPanel pannelloInferiore = new JPanel(new BorderLayout());
  pannelloInferiore.add(scroll, BorderLayout.CENTER);
  pannelloInferiore.add(pannelloBottoni, BorderLayout.PAGE_END);

  getContentPane().add(pannelloSuperiore, BorderLayout.PAGE_START);
  getContentPane().add(pannelloCentrale, BorderLayout.CENTER);
  getContentPane().add(pannelloInferiore, BorderLayout.PAGE_END);

  setSize(700, 550);
  setLocationRelativeTo(null);
  setVisible(true);
}

public static void main(String[] args) {
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + " e chiedo all'EDT di creare la GUI");
  Runnable init = new Runnable() {
    public void run() {
      new FrameImmagini();
    }
  };
```

```
    // creo la GUI nell'EDT
    SwingUtilities.invokeLater(init);
  }
}
```

La classe MyListener

```
package esempio5b;

import java.awt.event.*;

public class MyListener implements ActionListener {

  private FrameImmagini frame;
  private ImmaginiWorker worker;

  public MyListener(FrameImmagini frame) {
    this.frame = frame;
  }

  @Override
  public void actionPerformed(ActionEvent e) { // eseguito da EDT
    String command = e.getActionCommand();
    if (command == "scarica") {
      scarica();
    } else if (command == "cancel") {
      cancella();
    } else {
      frame.areaTesto.append("Testo: " + frame.campoTesto.getText() + "\n");
    }
```

```
  }

  private void scarica() { // eseguito da EDT
    System.out.println("Ciao, sono " + Thread.currentThread().getName()
        + ": aggiorno la GUI e attivo lo SwingWorker");
    frame.scarica.setEnabled(false);
    frame.cancel.setEnabled(true);
    frame.areaTesto.append("Inizio download immagini....\n");
    frame.download.setText("--");
    frame.barra.setIndeterminate(true);
    frame.pannelloCentrale.removeAll();
    frame.pannelloCentrale.validate();
    frame.pannelloCentrale.repaint();
    // creo ed avvio SwingWorker che eseguira' il download
    MonitorProgresso monitorProgresso = new MonitorProgresso(frame.barra);
    worker = new ImmaginiWorker(frame);
    worker.addPropertyChangeListener(monitorProgresso);
    worker.execute();
  }

  private void cancella() {
    frame.areaTesto.append("Richiesta di cancellazione...\n");
    frame.barra.setIndeterminate(false);
    frame.barra.setValue(0);
    worker.cancel(true);
  }
}
```

La classe ImmaginiWorker

```
package esempio5b;

import java.util.List;
import java.util.concurrent.ExecutionException;
import javax.swing.*;

public class ImmaginiWorker extends SwingWorker<Integer, ImageIcon> {

  private FrameImmagini frame;
  private int downloaded = 0;

  public ImmaginiWorker(FrameImmagini frame) {
    this.frame = frame;
  }

  @Override
  protected Integer doInBackground() throws Exception {
    System.out.println("Ciao, sono " + Thread.currentThread().getName()
        + " ed eseguo doInBackground()");
    int i = 1;
    while (!isCancelled() && i < 10) {
      Thread.sleep(1000);
      String fileName = "immagini/img" + i + ".png";
      ImageIcon img = new ImageIcon(fileName);
      downloaded += (int) (1000.0 * Math.random());
      i++;
      setProgress(i * 10);
      publish(img);
    }
    return downloaded;
  }
```

```
@Override
protected void process(List<ImageIcon> infoList) {
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + " ed eseguo process()");
  for (ImageIcon img : infoList) {
    if (isCancelled()) {
      break;
    }
    frame.pannelloCentrale.add(new JLabel(img));
    frame.pannelloCentrale.revalidate();
    frame.pannelloCentrale.repaint();
    frame.areaTesto.append("Nuova immagine scaricata!\n");
  }
}

@Override
protected void done() { // chiamato dopo doInBackground ed eseguito dall'EDT
  System.out.println("Ciao, sono " + Thread.currentThread().getName()
      + " ed eseguo done()");
  frame.scarica.setEnabled(true);
  frame.cancel.setEnabled(false);

  if (isCancelled()) {
    frame.areaTesto.append("Download interrotto!!!\n");
  } else {
    try {
      frame.areaTesto.append("Ricerca completata!!!\n");
      frame.download.setText(get() + "");
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
```

```
        e.printStackTrace();
      } catch (ExecutionException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
  }
}
```

La classe MonitorProgresso

```
package esempio5b;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import javax.swing.JProgressBar;

public class MonitorProgresso implements PropertyChangeListener {

  private JProgressBar barraAvanzamento;

  public MonitorProgresso(JProgressBar barra) {
    barraAvanzamento = barra;
  }

  @Override
  public void propertyChange(PropertyChangeEvent evt) { // eseguito da EDT
    System.out.println("Ciao, sono " + Thread.currentThread().getName()
        + " ed aggiorno la barra");
    String strPropertyName = evt.getPropertyName();
```

```
    if ("progress".equals(strPropertyName)) {
      barraAvanzamento.setIndeterminate(false);
      int progress = (Integer) evt.getNewValue();
      barraAvanzamento.setValue(progress);
    }
  }
}
```

Riferimenti Utili

- Oracle - Concurrency in Swing

  `http://download.oracle.com/javase/tutorial/uiswing/concurrency/`

- Improve Application Performance With SwingWorker in Java SE 6

  `http://java.sun.com/developer/technicalArticles/javase/swingworker/`

- Java SE 6 API - `SwingUtilities` e `SwingWorker`

  `http://download.oracle.com/javase/6/docs/api/javax/swing/SwingUtilities.html`

  `http://download.oracle.com/javase/6/docs/api/javax/swing/SwingWorker.html`