Programmazione Avanzata e Paradigmi
Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module lab 3.2]
GUI FRAMEWORKS & CONCURRENCY
GUI FRAMEWORKS & CONCURRENCY

• Once upon a time GUI applications were single-threaded...
  – GUI events processed by a “main event loop”
• ..modern GUI Frameworks are not so different
  – creating a dedicated event dispatch thread (EDT) for handling GUI events
  – the thread fetches events off a queue and dispatches them to application-defined event handlers
• Most of the modern GUI Frameworks are single-threaded
  – Java, QT, NextStep, Mac OS Cocoa, XWindow...
• Multithreaded GUI as a “failed dream” [*]
  – many attempts, failed due to the generation of problems with race conditions and deadlock

[*] referred in this way by Sun VP Graham Hamilton in his blog
http://weblogs.java.net/blog/kgh/archive/2004/10
MULTITHREADED GUI FRAMEWORKS: THE PROBLEM

- Clash between input event processing and OO modeling of GUI components
  - that can easily lead to *deadlocks* and *race conditions*
- Inconsistent lock ordering
  - in managing user-initiated actions and application-initiated actions
    - user actions “bubble up” from OS to the application
      - OS mouse click -> mouse click event of the GUI toolkit -> high-level event of the application listener...
    - application-actions “bubble down” from application to action
      - changing background color of a component at the application level -> dispatched to component class -> dispatched to OS for rendering,,,
- most of the actions need to lock objects
  - more generally related to MVC implementations
SINGLE-THREADED GUI

- Achieving thread-safety via **thread-confinement**
  - *all GUI objects are accessed exclusively by the event thread*
  - including visual components and data models
  - the application developer must make sure that these objects are properly confined

- Sequential event processing
  - events like kind of task to be processed sequentially by the event thread

- Problems and challenges
  - if one task takes long time to execute, other task must wait
    - blocking the overall GUI

> ...so tasks that execute in the event-thread must return quickly
  - to initiate a long-term task a separate thread must be used
    - es: spell-checking a document, searching the file system

> ...but typically a long-term task must provide a visual feedback for indicating progress or when it completes
  - and this code need to be executed by the event thread...
THREAD CONFINEMENT IN SWING

• All Swing components (such as JButton and JTable) and data models (e.g. Table Model and Tree Model) are confined to the event thread
  – any code that access these objects must run in the event thread
• Some exceptions
  – Swing methods that can be safely called from any thread
    • clearly identified in the Javadoc as thread-safe
THREAD-SAFE SWING METHODS

• Thread-Safe methods
  – SwingUtilities.isEventDispatchThread
    • to check if the current thread is the event thread
  – SwingUtilities.invokeLater
    • to schedule a Runnable for execution on the event thread
  – SwingUtilities.invokeLater
    • to schedule a Runnable task for execution on the event thread, blocking the current thread until it completes
    • cannot be called by the event thread
  – methods to enqueue and repaint or revalidation request on the event queue
  – methods for adding or removing listeners
    • can be called from any thread, but listeners will always be invoked in the event thread
EXECUTOR

• The swing event thread can be thought as a single-threaded Executor that processes tasks from the event queue
  – invokeLater and invokeAndWait used to submit new tasks to execute
A SIMPLE GUI EXECUTOR

- Executor delegating tasks to `SwingUtilities` for executions

```java
public class GuiExecutor extends AbstractExecutorService {
    // Singletons have a private constructor and a public factory
    private static final GuiExecutor instance = new GuiExecutor();

    private GuiExecutor() { }

    public static GuiExecutor instance() { return instance; }

    public void execute(Runnable r) {
        if (SwingUtilities.isEventDispatchThread())
            r.run();
        else
            SwingUtilities.invokeLater(r);
    }

    // Plus trivial implementations of lifecycle methods
}
```
SHORT-RUNNING GUI TASKS

• Can be executed directly by the event thread
• Simple example

```java
final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});
```

• The control never leaves the event thread
  - the event originates in the GUI toolkit, is delivered to the application, the application modifies the GUI in response to user’s action
LONG-RUNNING TASKS

• Some of the processing must be *offloaded* to another thread
  – exploiting executors

• Two main cases
  – long-term task without visual feedbacks
    • simple case, quite unfrequent
  – long-term task with visual feedbacks
    • complex case, most frequent
BINDING A TASK WITHOUT VISUAL FEEDBACKS

- Exploiting a simple separated executor (or thread)
- Example:

```java
ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        backgroundExec.execute(new Runnable() {
            public void run() { doBigComputation(); }
        });
    }
});
```
LONG-RUNNING TASK WITH USER FEEDBACKS

- The long-running task must submit another task to run in the event thread whenever the user interface must be updated
- Example:

```java
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation();
                } finally {
                    GuiExecutor.instance().execute(new Runnable() {
                        public void run() {
                            button.setEnabled(true);
                            label.setText("idle");
                        }
                    });
                }
            }
        });
    }
});
```
SwingWorker

- Java 6.0 provides auxiliary classes for making it easier to program complex long-term tasks that can interact with the GUI
  - a
- SwingWorker class
  - provide a direct support for task cancellation, completion notification and progress indication

```java
class SwingWorker<T,V> implements RunnableFuture<T> {
    ...
    // to be overridden
    protected abstract  T doInBackground();
    protected  void done()
    ...
    protected final void publish(V... chunks)
    protected void process(List<V> chunks);
    ...
    // to be directly used
    boolean cancel(boolean mayInterruptIfRunning);
    protected  void setProgress(int progress);
    ...
}
```
TASK EXECUTION AND INTERFACE UPDATE (1/2)

- Support for asynchronous task execution & consequent interface update
  - `doInBackground`
    - encapsulate the computational body of the task to be executed asynchronously w.r.t. GUI activity, computing a result or throwing an exception if unable to do so
    - executed by some thread, not by the Swing EDT
  - `done`
    - encapsulate the action to do on the GUI when the task completed
    - executed by the Swing EDT
TASK EXECUTION AND INTERFACE UPDATE (2/2)

• Support for asynchronous update of interfaces
  – `publish(V... chunks)`
    • used from inside `doInBackground` to deliver intermediate results for processing on the Event Dispatch Thread inside the process method
  – `process(List<V> chunks);`
    • receives data chunks from the `publish` method asynchronously on the EDT
class CounterTask extends SwingWorker<Integer, Integer> {

    protected Integer doInBackground() throws Exception {
        int i = 0;
        int sum = 0;
        int maxCount = 10;
        while (!isCancelled() && i < maxCount) {
            sum+=i;
            i++;
            publish(new Integer[] { i });
            setProgress(100 * i / maxCount);
            Thread.sleep(1000);
        }
        return sum;
    }

    protected void process(List<Integer> chunks) {
        for (int i : chunks)
            System.out.println("Step "+i);
    }

    protected void done() {
        if (isCancelled()){
            System.out.println("Task cancelled.");
        } else {
            System.out.println("Task completed.");
        }
    }
}

```java
public class SwingWorkerTest {
    public static void main(String[] args) {
        JTextArea textArea = new JTextArea(10, 20);
        JProgressBar progressBar = new JProgressBar(0, 100);
        CounterTask task = new CounterTask();

        JButton startButton = new JButton("Start");
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { task.execute(); }});

        JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { task.cancel(true); } });

        task.addPropertyChangeListener(new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                if ("progress".equals(evt.getPropertyName())) {
                    progressBar.setValue((Integer) evt.getNewValue());
                }
            }
        });

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(startButton);
        buttonPanel.add(cancelButton);
        JPanel cp = new JPanel();
        LayoutManager layout = new BoxLayout(cp, BoxLayout.Y_AXIS);
        cp.setLayout(layout);
        cp.add(buttonPanel);
        cp.add(new JScrollPane(textArea));
        cp.add(progressBar);
        JFrame frame = new JFrame("SwingWorker Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(cp);
        frame.pack();
        frame.setVisible(true);
    }
}
```