Multi-threading in Swing

See also:
- Swing tutorial
- Flipper example

22 Mar 2010
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Outline for today

- SwingWorker class for threads in Swing
- Sending and receiving results:
  - doInBackground() and done()
- Publishing progress updates / interim results:
  - publish() and process()
- Cancelling a background task
  - cancel() and isCancelled()

Threads in Swing

- Swing programs have multiple threads:
  - Init thread (main() setup before GUI)
  - Event dispatch thread (interacts w/GUI)
  - Any worker threads you create
- Only the event dispatch thread should access the GUI (change widget text, etc.)
  - Worker threads have to ask the event dispatch thread to update the GUI
- How do worker threads communicate to the event dispatch thread?

SwingWorker abstract class

- Subclass of Thread that allows you to:
- Define the task to be done in background
- Run code on the event dispatch thread when the worker thread is done
- Return an object from the worker thread to the event dispatch thread
- Send progress updates from the worker thread to the event dispatch thread
- Define bound properties: when the worker thread changes them, events get sent to the event dispatch thread

Using SwingWorker

- SwingWorker is abstract: so subclass it
  - **class Fetcher extends SwingWorker {**
- SwingWorker is templated: specify the type/class of object returned by the bg task:
  - **class Fetcher extends SwingWorker<Image, Void>**
- Override doInBackground() to define the task:
  - **public Image doInBackground() { … }**
    - Return type is same as in template
    - Should only modify local variables
    - Return result of the long-running task

Getting the result: done()

- Override the done() method to define how the event dispatch thread gets the results:
  - **public void done() {**
    try {
    myButton.setIcon(get());
    } except (InterruptedException e) {
    } except (ExecutionException e) {
    }
- This method is run on the event dispatch thread
- Not called until the worker thread has finished
  - get() blocks until worker is finished
- Copies from return value of doInBackground()

Starting the worker thread

- To get the worker thread running:
  Create an instance of your subclass of SwingWorker and call its .execute() method
  - **Fetcher fetcher = new Fetcher();**
  - **fetcher.execute();**
    - Different from usual Thread.start()
- This could be done in the action listener for a button, for instance

Example with SwingWorker

```
public void actionPerformed(ActionEvent evt) {
    (new SwingWorker<ImageIcon, Void>() {
        public ImageIcon doInBackground() {
            ImageIcon img =
                (ImageIcon) serverIn.getObject();
            return img;
        }
        public void done() {
            try {
                myButton.setIcon( get() );
            } except (InterruptedException e) {
            } except (ExecutionException e) {
            }
        }
    }).execute();
}
```

event listener for button

anonymous class

slow task

get obj returned by doInBackground()

run by event disp. thread

start the thread

Publishing progress updates

- The worker thread may send objects to the event dispatch thread as interim results:
- Declare type of interim result in template:
  - **… extends SwingWorker<Image, Float> {**
- From doInBackground(), call publish():
  - **publish(bytesFetched / totBytes);**
- Override process() to specify how event dispatch thread handles an update:
  - **public void process(List<Float> updates) {**
    - Given a List of accumulated updates
- publish() may be called very very frequently!

Summary of SwingWorker

```
(new SwingWorker<ImageIcon, Float>() {
    public ImageIcon doInBackground() {
        // long task
        // periodically call publish() with an update
        // return an ImageIcon
    }
    public void process( List<Float> updates ) {
        // update progress bar UI, etc.
    }
    public void done() {
        try {
            // get() ImageIcon result, then setIcon(), etc.
        } except (InterruptException e) { ... }
    }
}).execute();
```

Cancelling a background task

- Call the .cancel() method of the worker thread
  - Means thread can't be an anon. object
- In the worker thread (doInBackground()), check if we've been cancelled: if (isCancelled())
- Or cancel using interrupts:
  - Call cancel(true) instead of just cancel()
  - Worker thread receives InterruptException
  - Only if worker thread is doing something that can raise InterruptException: Thread.sleep(), network send/receive, ...