1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 28: Concurrency with Swing

Concurrency in Swing

- A well-written Swing program uses concurrency to create a user interface that never “freezes“, i.e., the program is always responsive to user interaction
  - Note that most of the methods in Swing classes are not “thread safe”: you need to ensure that all calls to them are handled in the same thread to avoid memory consistency errors
- A Swing programmer deals with three different kinds of threads:
  - ***Initial threads***, where the initial application code is executed
  - ***Event dispatch threads***, where the code for handling events is executed
  - ***Worker threads***, where time-consuming background tasks are executed

Initial threads

- In Swing programs, the initial threads typically just create an object implementing **`Runnable`** interface that initializes the GUI and schedule that object for execution on the event dispatch thread
  - GUI creation task is scheduled by invoking either **`invokeLater()`** or **`invokeAndWait()`** method of SwingUtilities package
  - Both methods take a single argument, i.e., a **`Runnable`** object defining the new task.
  - The methods differ in that `invokeLater()` simply schedules the task and returns, whereas `invokeAndWait()` waits for the scheduled task to finish before returning.

InvokeLater example (1)

```
1  import java.awt.event.*;
2  import  javax.swing.*;
3  public class InvokeLaterExample {
4    private static void createAndShowGUI() {
5      System.out.println("Creating GUI...");
6      try {
7        Thread.sleep(1000);
8      } catch(Exception e) {
9        e.printStackTrace();
10     }
11     System.out.println("GUI created!");
12   }
13   public static void main(String[] args) {
14     javax.swing.SwingUtilities.invokeLater(new Runnable() {
15       public void run() {
16         createAndShowGUI();
17       }
18     });
19     System.out.println("invokeLater() completed!");
20   }
21 }
```

Let us play that it takes one second to create GUI

`InvokeLater()` returns immediately after scheduling the new `Runnable` to execute

```
$ java InvokeLaterExample
invokeLater() completed!
Creating GUI...
```

InvokeLater example (2)

```
1  import java.awt.event.*;
2  import  javax.swing.*;
3  public class InvokeLaterExample {
4    private static void createAndShowGUI() {
5      System.out.println("Creating GUI...");
6      try {
7        Thread.sleep(1000);
8      } catch(Exception e) {
9        e.printStackTrace();
10     }
11     System.out.println("GUI created!");
12   }
13   public static void main(String[] args) {
14     javax.swing.SwingUtilities.invokeLater(new Runnable() {
15       public void run() {
16         createAndShowGUI();
17       }
18     });
19     System.out.println("invokeLater() completed!");
20   }
21 }
```

`createAndShowGUI()` completes execution one second later

```
$ java InvokeLaterExample
invokeLater() completed!
Creating GUI...
GUI created!
$
```

InvokeAndWait example

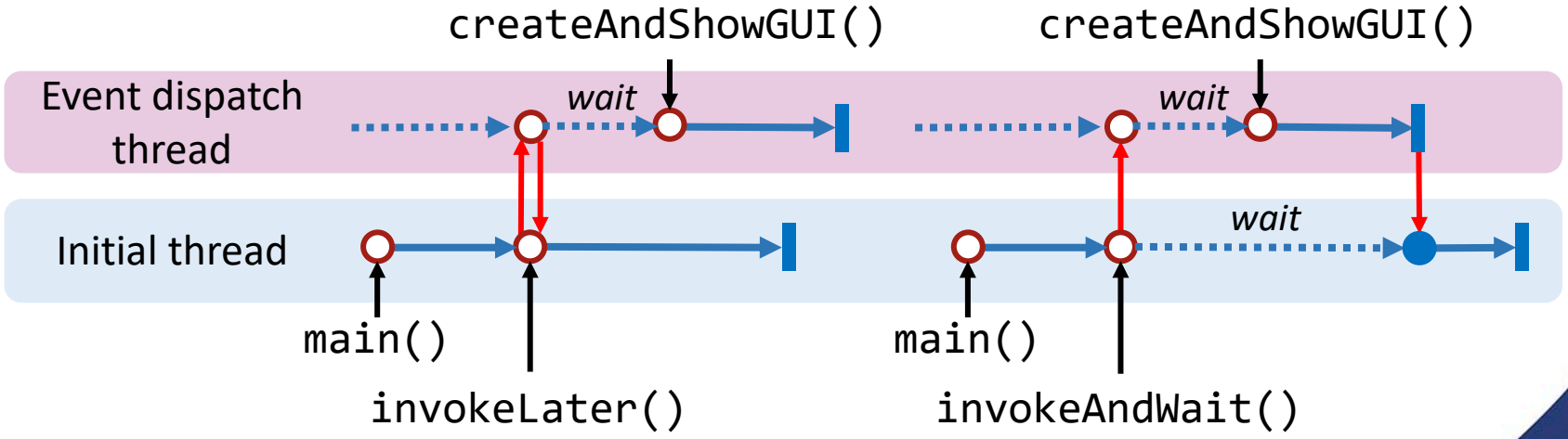
```
1  import java.awt.event.*;
2  import javax.swing.*;
3  public class InvokeAndWaitExample {
4    private static void createAndShowGUI() {
5      System.out.println("Creating GUI...");
6      try {
7        Thread.sleep(1000);
8      } catch(Exception e) {
9        e.printStackTrace();
10     }
11     System.out.println("GUI created!");
12   }
13   public static void main(String[] args) {
14     try {
15       javax.swing.SwingUtilities.invokeAndWait(new Runnable() {
16         public void run() {
17           createAndShowGUI();
18         }
19       });
20     } catch(Exception e) {
21       e.printStackTrace();
22     }
23     System.out.println("invokeAndWait() completed!");
24   }
25 }
```

InvokeAndWait() does not return before createAndShowGUI() has completed

```
$ java InvokeLaterExample
Creating GUI...
GUI created!
invokeAndWait() completed!
$
```

Summary: invokeLater and invokeAndWait

- Method `invokeLater()` is *asynchronous* (i.e., *non-blocking*), whereas `invokeAndWait()` is *synchronous* (i.e., *blocking*)



Event dispatch threads

- Since most Swing object methods are not “thread safe”, invoking them from multiple threads causes a risk of thread interference
  - Some Swing component methods are labelled “thread safe” in the API specification, and they can be safely invoked from any thread
  - All the other Swing component methods must be invoked from the *event dispatch thread*
    - Programs ignoring this rule may function correctly most of the time, but are prone to unpredictable errors that are difficult to track and reproduce
    - Tasks on the event dispatch thread must finish quickly; if they do not, unhandled events back up and the user interface becomes unresponsive

Event dispatch thread example

```
1   import java.awt.event.*;
2   import javax.swing.*;
3   public class EventDispatcherExample {
4     private static void createAndShowGUI() {
5       System.out.print("Creating GUI in " + Thread.currentThread());
6       System.out.println("Is event dispatch thread: " +
7         SwingUtilities.isEventDispatchThread());
8       JFrame frame = new JFrame("Event Dispatch Demo");
9       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10      JButton button = new JButton("Press Me!");
11      button.addActionListener(new ActionListener() {
12        public void actionPerformed(ActionEvent e) {
13          System.out.println("Button event in " + Thread.currentThread());
14          System.out.println("Is event dispatch thread: " +
15            SwingUtilities.isEventDispatchThread());
16        }
17      });
18      frame.add(button);
19      frame.pack();
20      frame.setVisible(true);
21    }
...
...
```

You can use static method `currentThread()` to get information about the current thread

You can use static method `isEventDispatchThread()` to determine whether you are currently in the event dispatch thread


```
$ java EventDispatcherExample
Creating GUI in Thread[AWT-EventQueue-0,6,main]
Is event dispatch thread: true
Button event in Thread[AWT-EventQueue-0,6,main]
Is event dispatch thread: true
$
```

Worker threads and SwingWorker

- When a Swing program needs to execute a long-running task, it usually uses a *worker thread*, also known as *background thread*
  - Each task running on a worker thread is represented by an instance of a subclass of abstract class **`SwingWorker`**
- Three threads are involved in the life cycle of a `SwingWorker`:
  - Current thread (often event dispatch thread): calls the **`execute()`** method to schedule execution of `SwingWorker`
  - Worker thread: calls **`doInBackground()`** method, where all the background activity should happen
  - Event dispatch thread: `SwingWorker` invokes **`process()`** and **`done()`** methods on this thread

SwingWorker example (1)

```
1  import javax.swing.*;
2  import javax.swing.SwingUtilities.*;
3  import javax.swing.SwingWorker.*;
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.beans.*;
7  public class SwingWorkerExample {
8    private static SwingWorker createWorker() {
9      return new SwingWorker() {
10       @Override protected Boolean doInBackground() throws Exception {
11         setProgress(0);
12         for(int i=0; i<=100; i++) {
13           Thread.sleep(500);
14           setProgress(i);
15         }
16         return false;
17       }
18     };
19   }
```

For your `SwingWorker` object, you need to override `doInBackground()` method to implement the background task to be executed

In this example, the only task is to increase a counter and update progress two times per second

SwingWorker example (2)

```
20    private static void createAndShowGUI() {
21      JFrame frame = new JFrame();
22      JPanel panel = new JPanel();
23      JButton button = new JButton("Start");
24      JProgressBar progBar = new JProgressBar(0,100);
25      progBar.setValue(0);
26      progBar.setStringPainted(true);
27      panel.add(button);
28      panel.add(progBar);
29      frame.add(panel);
30      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31      frame.setLocationRelativeTo(null);
32      frame.setSize(250, 100);
33      frame.setVisible(true);
34      SwingWorker worker;
```

Create UI with a button and progress bar

Worker object variable for later use

SwingWorker example (3)

```
35      button.addActionListener(new ActionListener() {
36        @Override public void actionPerformed(ActionEvent e) {
37          button.setEnabled(false);
38          progBar.setValue(0);
39          SwingWorker worker = createWorker();
40          worker.addPropertyChangeListener(
41            new PropertyChangeListener() {
42              public void propertyChange(PropertyChangeEvent e) {
43                if ("progress".equals(e.getPropertyName())) {
44                  progBar.setValue((Integer)e.getNewValue());
45                }
46                else if ("state".equals(e.getPropertyName())) {
47                  if (e.getNewValue() == StateValue.DONE) {
48                    button.setText("Restart");
49                    button.setEnabled(true);
50                  }
51                }
52              }
53          });
54          worker.execute();
```

Add `ActionListener` to the button to create and execute `SwingWorker` object when the button is pressed

SwingWorker example (4)

```
35      button.addActionListener(new ActionListener() {
36        @Override public void actionPerformed(ActionEvent e) {
37          button.setEnabled(false);
38          progBar.setValue(0);
39          SwingWorker worker = createWorker();
40          worker.addPropertyChangeListener(
41            new PropertyChangeListener() {
42              public void propertyChange(PropertyChangeEvent e) {
43                if ("progress".equals(e.getPropertyName())) {
44                  progBar.setValue((Integer)e.getNewValue());
45                }
46                else if ("state".equals(e.getPropertyName())) {
47                  if (e.getNewValue() == StateValue.DONE) {
48                    button.setText("Restart");
49                    button.setEnabled(true);
50                  }
51                }
52              }
53          });
54          worker.execute();
```

Add `PropertyChangeListener` to the worker to handle progress and status updates from the worker thread

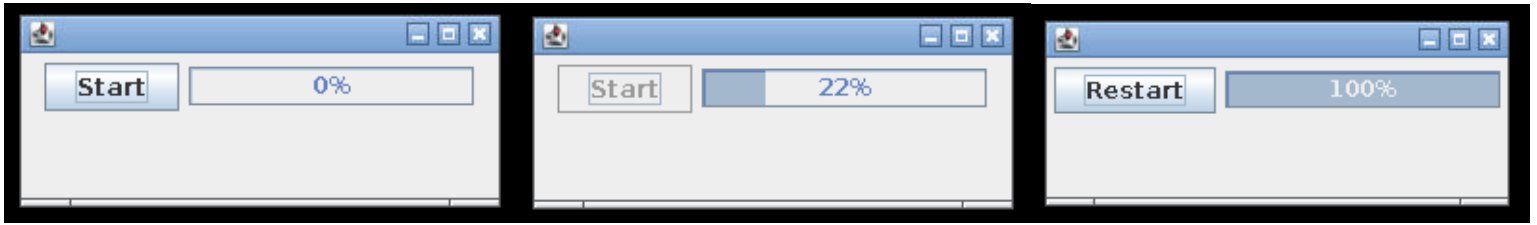
Update progress bar when the worker thread calls `setProgress()` method

Reinitialise button when the worker thread is done

SwingWorker example (5)

```
55          }
56        });
57    }
58    public static void main(String[] args) {
59      SwingUtilities.invokeLater(new Runnable() {
60        public void run() {
61          createAndShowGUI();
62        }
63      });
64    }
65  }
```

The `main()` method calls `invokeLater()` to create GUI and start the application



Questions, comments?