POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

Lecture 13

Implementation of Graphical User Interfaces (2)

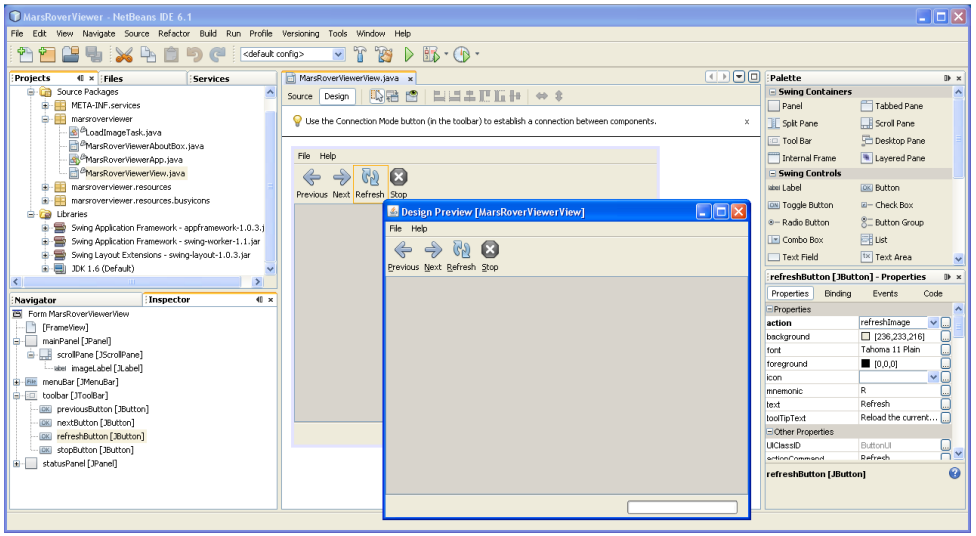
http://www.mtrzaska.com/

Outline

- An introduction
- Creating a simple GUI using the NetBeans visual editor.
- Long-running actions and GUI
  - Problems,
  - A solution using threads.
- Creating a complex application.
- Using a GUI editor for developing MDI apps.

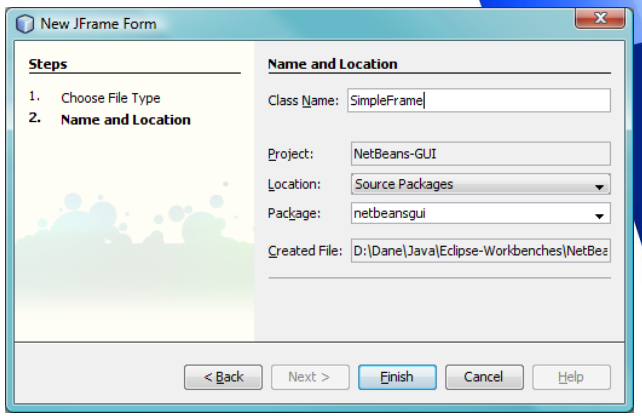
The NetBeans Editor

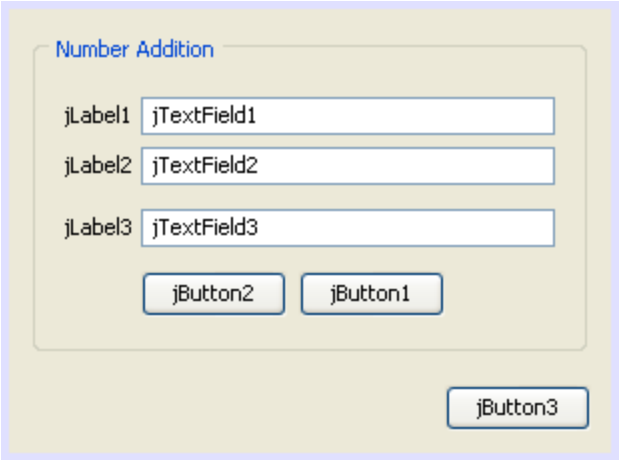
- Full visual design of the GUI,
- Very good system supporting aligning controls (*layout manager*),
- Many different widgets,
- A support for the *Beans Binding technology (JSR 295)*
- A support for the *Swing Application Framework (JSR 296) – put on hold.*



Utilization of the Editor – a Simple Sample

- A new project (The Adder): File > New Project: Java/Java Application
- Create a new window (`JFrame`): project node and menu New/JFrame Form
- Add `JPanel` (together with a right *Border*)
- Add:
  - Labels
  - Text Fields
  - Buttons.





http://www.netbeans.org/kb/60/java/gui-functionality.html

Utilization of the Editor – a Simple Sample (2)

- Change text fields' names and buttons:
  - Select a control,
  - Form the context menu choose *Change Variable Name* and provide the right name (e.g. `txtFirstNumber` or `btnAdd`).
- Change labels and buttons texts
  - Double click on the label and enter the new text,
  - Single click on the button and change the *Text* property in the *Properties* window.

http://www.netbeans.org/kb/60/java/gui-functionality.html

Utilization of the Editor – a Simple Sample (3)

- Add a functionality
  - We need to know when the button will be pressed. Hence we will use a *listener* listening for a particular *event* for the control, e.g. a button.
  - Select a control (e.g. the button) and from the context menu choose: *Events/Action/actionPerformed*. The IDE will generate an empty event procedure.
  - In the procedure there will be a business method call which will execute some functionality. It is important to call a business method rather then placing there the entire business code.

.../gui-functionality.html

Utilization of the Editor – a Simple Sample (4)

- Add a functionality - *cont.*
  - For the *Exit* button and a name of the method finished the application, e.g. `exit()`. Because such a method does not exist, the IDE will show an error message and ask if the method should be created.

```
@SuppressWarnings("unchecked")
cannot find symbol
symbol : method exit()
location: class netbeansgui.NumberAddition
erformed(java.awt.event.ActionEvent evt) {
    exit();
Create Method exit() in netbeansgui.NumberAddition

    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
```

  - Add a JavaDoc comment by typing **/**** and pressing the *Enter* button.

```
    /**
     * Exits the application.
     */
    private void exit() {
        System.exit(0);
    }
```

  - The above procedure should be repeated for all buttons.

http://www.netbeans.org/kb/60/java/gui-functionality.html

Utilization of the Editor – a Simple Sample (5)

- Add a functionality - *cont.*

```
/**
 * Clears the input and output fields.
 */
private void clearInput() {
   txtFirstNumber.setText("");
   txtSecondNumber.setText("");
   txtResult.setText("");
}

/**
 * Counts the result.
 */
private void count() {
   float num1, num2, result;

   // Convert the text into number
   var txtNumber1 = txtFirstNumber.getText();
   num1 = (txtNumber1 == null || txtNumber1.isEmpty()) ? 0 : Float.parseFloat(txtNumber1);
   var txtNumber2 = txtSecondNumber.getText();
   num2 = (txtNumber2 == null || txtNumber2.isEmpty()) ? 0 : Float.parseFloat(txtNumber2);

   // Count the result
   result = num1 + num2;

   // Put the result into the field
   txtResult.setText(String.valueOf(result));
}
```

http://www.netbeans.org/kb/60/java/gui-functionality.html

Utilization of the Editor – a Simple Sample (6)

- Start the program by selecting the *Run file* from the project's context menu.
- In case of lack of the `main` method add the following code:

```
public static void main(String args[]) {
   java.awt.EventQueue.invokeLater(new Runnable() {

      public void run() {
         new NumberAddition().setVisible(true);
      }
   });
}
```

http://www.netbeans.org/kb/60/java/gui-functionality.html

Utilization of the Editor – a Simple Sample (7)

Demonstration of

the NetNeans GUI editor

Long-running actions and GUI

- Users do not like when a computer does not respond to their commands.
- It could happen when the application executes some long-lasting actions, e.g.
  - Searching for data,
  - Calculations,
  - Data processing,
  - Downloading/uploading resources,
  - Waiting for services.

Long-running actions and GUI (2)

- In such a case we need to:
  - Tell a user about the long-lasting actions,
  - Update on-the-fly progress information,
  - Allow cancelling them.
- Studies show that even long-lasting actions seems to be faster when the progress information is present.
- If we are able to foresee that the action will take long time that warn the user.

Long-running actions and GUI (3)

- Why the following code will not work according to our expectations?

```
Public void longWorkingMethod() {

    For(...) {
        // Executing a long lasting method,
        // e.g. looking for an information

        // Updating GUI
        progressBar.setProgress(...);
    }
}
```

?!

- What seems to be a problem?
- How to solve it?

Long-running actions and GUI (4)

- Unfortunately, modern programming languages do not support such a functionality directly.
- The proper implementation is based on **threads**.
- Usually such an implementation requires a utility class shipped with a programming platform:
  - `SwingWorker` for the Java Swing,
  - `BackgroundWorker` for the MS C# (Windows Forms),
  - `javafx.concurrent.Task` dla JavaFX (*JavaDoc*).

Long-running actions and GUI (5)

- Both classes are built using a similar approach. A programmer has to override the method:
  - Executing the long-lasting action: `SwingWorker.doInBackground()`,
  - Executed when the job is done: `SwingWorker.done()`.
- There are also some other useful methods allowing
  - Cancelling the operation,
  - Getting partial results.

The SwingWorker class - Samples

- In the minimal version we need to override the method executing the long-running action.
- The `SwingWorker` class is strongly typed (Java generics) by:
  - the type returned by the long-lasting methods:
    - `doInBackground()`,
    - `get()`. The method waits for the action to finish the action (such waiting freezes the GUI).
  - The type of the partial results (as a list).

The SwingWorker class - Samples (2)

- The sample counts a sum from 1 to the given `maxNumber`.

```
public class Worker extends SwingWorker<Integer, Void> {

    @Override
    /**
     * Count the sum of numbers from 1 to the given maxNumber and return it as
String.
     */
    protected Integer doInBackground() throws Exception {
        // Count the sum of numbers from 1 to the given maxNumber
        final int maxNumber = 500000000;
        int sum = 0;

        for (long i = 1; i < maxNumber; i++) {
            sum += i;
        }

        return sum;
    }
}
```

The SwingWorker class - Samples (3)

- Utilization of the sample.

```
public static void main(String[] args) {
    Worker worker = new Worker();

    System.out.println("Counting...");

    // Start the long-lasting action
    worker.execute();

    // But the control flow returns immediately
    System.out.println("Waiting for the result...");

    // Exceptions required by SwingWorker
    try {
        // Waiting and checking if it is finished
        while(!worker.isDone()) {
            Thread.sleep(1000);
        }

        // We have the result - get it
        var sum = worker.get();

        System.out.println("The result: " + sum);
    } catch (InterruptedException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (ExecutionException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
ex);
    }
}
```

The SwingWorker class - Samples (4)

- Sample counting a sum from 1 to the `maxNumber` (with a console progress info)

```
public class WorkerPlus extends SwingWorker<Integer, Void> {

    /**
     * Count the sum of numbers from 1 to the given maxNumber with updates.
     */
    @Override
    protected Integer doInBackground() throws Exception {
        // Count the sum of numbers from 1 to the given maxNumber
        final int maxNumber = 500000000;
        int sum = 0;

        for (int i = 1; i < maxNumber; i++) {
           sum += i;
           if(i % 100 == 0) {
                // Store the progress info
                double percentage = 100.0 * (float) i / maxNumber;
                setProgress((int) percentage);
            }
        }

        return sum;
    }
}
```

The SwingWorker class - Samples (5)

- Utilization of the sample.

```
public static void main(String[] args) {
    var worker = new WorkerPlus();

    System.out.println("Counting...");

    // Start the long-lasting action
    worker.execute();

    // But the control flow returns immediately
    System.out.println("Waiting for the result...");


    // Exceptions required by SwingWorker
    try {
        // Waiting and checking if it is finished
        while(!worker.isDone()) {
            System.out.println("Progress: " + worker.getProgress());
            Thread.sleep(100);
        }

        // We have the result - get it
        int floatSum = worker.get();

        System.out.println("The result: " + String.valueOf(floatSum));
    } catch (InterruptedException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ExecutionException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

The SwingWorker class – a GUI Sample

- Create a dedicated window (`JDialog`) in the editor
  - `JLabel` showing a message,
  - `JProgressBar` with visual information about a progress,
  - `JButton` allowing cancelling the action,
  - A couple of utility methods.

The SwingWorker class – a GUI Sample (2)

- Create a dedicated window (`JDialog`) in the editor - *cont.*

```java
public class ProgressDialog extends javax.swing.JDialog {
   private SwingWorker worker;

   /**
    * Sets a worker connected to the dialog.
    * @param worker
    */
   public void setWorker(SwingWorker worker) {
      this.worker = worker;
   }

   /**
    * Creates new form ProgressDialog
    */
   public ProgressDialog(java.awt.Frame parent, String message) {
      super(parent, true);
      initComponents();

      lblMessage.setText(message);
   }

   public void setValue(Integer progress) {
      progressBar.setValue(progress);
   }

   private void initComponents() {
       // [...]
   }
}
```

The SwingWorker class – a GUI Sample (3)

- Extend the `SwingWorker` class by adding implementation of the long-lasting action.

```
public class NumberGenerator extends SwingWorker<List<Integer>,
Void> {
    private List<Integer> numbers;
    private int maxCount;
    ProgressDialog progressDialog;

    public NumberGenerator(int maxCount, ProgressDialog
progressDialog) {
        super();

        this.maxCount = maxCount;
        this.progressDialog = progressDialog;
    }

    // [...]

    @Override
    protected void done() {
        super.done();

        // Hide the GUI (dialog window)
        progressDialog.setVisible(false);
    }
}
```

The SwingWorker class – a GUI Sample (4)

- Extend the `SwingWorker` class by adding implementation of the long-lasting action - *cont.*

```
public class NumberGenerator extends SwingWorker<List<Integer>, Void> {
    private List<Integer> numbers;
    private int maxCount;
    ProgressDialog progressDialog;

    // [...]

    /** Generates the numbers and returns them as a list. */
    @Override
    protected List<Integer> doInBackground() throws Exception {
        final int maxNumber = 1000;
        int i = 1;

        numbers = new LinkedList<Integer>();

        // Check if we already generated all required numbers all the operation has been cancelled
        while(i < maxCount && isCancelled() == false) {
            Integer curInt = new Integer((int)(Math.random() * (double) maxNumber));
            numbers.add(curInt);

            if(i % 100 == 0) {
                // Store the update info
                setProgress((int)(100.0 * (float)i / maxCount));
            }

            i++;
        }
        return numbers;
    }}
```

The SwingWorker class – a GUI Sample (5)

- Create a window which will start the generation process.

```
public class MainFrame extends javax.swing.JFrame {

   public MainFrame() {
      initComponents();
   }

   private void generate() {
      final int count = 10000000;

      // Create a GUI with the update info
      final ProgressDialog dialog = new ProgressDialog(this, "Please
wait...");

      // Create the numbers generator
      NumberGenerator generator = new NumberGenerator(count, dialog);

      dialog.setWorker(generator);

      // Add a listener triggering the update info
      generator.addPropertyChangeListener(new PropertyChangeListener() {
         public void propertyChange(PropertyChangeEvent evt) {
            if ("progress".equals(evt.getPropertyName())) {
               dialog.setValue((Integer) evt.getNewValue());
            }
         }
      });

      // Start the generation process
      generator.execute();

      // Show the modal dialog
      dialog.setVisible(true);
   }

   // [...]
```

Extending the SwingWorker Class

- Let's create a class which make it easier working with the `SwingWorker` and GUI
  - Functionality for data processing (`SwingWorker`),
  - Showing progress information using GUI.
- The `ProgressWorker<finalResult, partialResult>` class
  - Extends the `SwingWorker`,
  - Contains modified instance of the `ProgressDialog` class.

Extending the SwingWorker Class (2)

- The constructor

```
public abstract class ProgressWorker<finalResult, partialResult> extends
SwingWorker<finalResult, partialResult> {
    private ProgressDialog progressDialog;

    /**
     * Creates a new instance presenting the user's message.
     * @param progressMessage
     */
    public ProgressWorker(String progressMessage) {
        super();

        progressDialog = new ProgressDialog(null, progressMessage);
        progressDialog.setWorker(this);

        // Add a listener to have updates about the progress
        addPropertyChangeListener(
                new PropertyChangeListener() {
                    public void propertyChange(PropertyChangeEvent evt) {
                        if ("progress".equals(evt.getPropertyName())) {
                            progressDialog.setValue((Integer) evt.getNewValue());
                        }
                    }
                }
        );
    }

    // [...]
```

Extending the SwingWorker Class (3)

- Special methods
  - `Start()`
  - `done()`.

```
public abstract class ProgressWorker<finalResult, partialResult>
                        extends SwingWorker<finalResult, partialResult>
{
    private ProgressDialog progressDialog;

    // [...]

    /**
     * Starts the long-lasting activity and shows GUI.
     * This method should be executed instead of the execute() method.
     */
    public void start() {
        execute();

        progressDialog.setVisible(true);
    }

    /**
     * Executed automatically when all calcutaions are finished or
cancelled.
     * Call the super method and hides the GUI.
     */
    @Override
    protected void done() {
        super.done();

        // Hide GUI
        progressDialog.setVisible(false);
    }
```

Utilization of the `ProgressWorker` Class

```java
public class NumberGenerator extends ProgressWorker<List<Integer>, Void> {
    private List<Integer> numbers;
    private int maxCount;

    public NumberGenerator(int maxCount) {
        // We need to call the super-class constructor with the user's message
        super("Generating numbers (" + maxCount + "))... Please wait...");

        this.maxCount = maxCount;
    }

    @Override
    protected List<Integer> doInBackground() throws Exception {
        final int maxNumber = 1000;
        int i = 1;

        numbers = new LinkedList<Integer>();

        while(i < maxCount && isCancelled() == false) {
            Integer curInt = new Integer((int)(Math.random() * (double) maxNumber));
            numbers.add(curInt);

            if(i % 100 == 0) {
                // Update progress info
                setProgress((int)(100.0 * (float)i / maxCount));
            }

            i++;
        }

        return numbers;
    }
}
```

Utilization of the `ProgressWorker` Class (2)

```java
public class NumberFinder extends ProgressWorker<Integer, Void> {
    private List<Integer> numbers;
    Integer numberToFind;

    public NumberFinder(List<Integer> numbers, Integer numberToFind) {
        super("Counting occurencies of the number: " + numberToFind + "...");

        this.numberToFind = numberToFind;
        this.numbers = numbers;
    }
    @Override
    protected Integer doInBackground() throws Exception {
        int count = 0;
        int i = 0;

        for(Integer number : numbers) {
            if(isCancelled()) {
                return new Integer(count);
            }

            if(number.equals(numberToFind)) {
                count++;
            }

            if(i % 100 == 0) {
                // Update progress info
                setProgress((int)(100.0 * (float)i / numbers.size()));
            }

            i++;
        }

        return new Integer(count);
    }}
```

Utilization of the `ProgressWorker` Class (3)

```
public class GeneratorAndFinder extends javax.swing.JFrame {
    // [...]

    private void GenerateFind() throws NumberFormatException, HeadlessException {

        try {
            // Generate a list with numbers
            NumberGenerator gen = new NumberGenerator((Integer)
spnCount.getValue());
            gen.start();

            // Check if cancelled
            if(gen.isCancelled()) {
                JOptionPane.showMessageDialog(null, "The operation has been
cancelled.");
                return;
            }

            // Get the result list
            List<Integer> numbers = gen.get();

            JOptionPane.showMessageDialog(null, "The generated list constains: "
                + numbers.size() + " numbers.\nPress OK to start counting
occurencies.");

             // [...]
```

Utilization of the `ProgressWorker` Class (4)

```
public class GeneratorAndFinder extends javax.swing.JFrame {
    // [...]

    private void GenerateFind() throws NumberFormatException, HeadlessException {

        try {

            // cont.

            // Count the occurencies of the given number
            NumberFinder finder = new NumberFinder(numbers,

Integer.parseInt(txtNumberToFind.getText()));
            finder.start();

            // Check if cancelled
            if(finder.isCancelled()) {
                JOptionPane.showMessageDialog(null, "The operation has been cancelled.");
                return;
            }
            // Get the result
            int count = finder.get();
            // Show the result
            JOptionPane.showMessageDialog(null, "The number " + txtNumberToFind.getText()
                                                + " occured " + count + " times.");
        } catch [...]
    }
```

The MDI Application

- Divide the source codes into packages, e.g.
  - `mt.mas.sampleapplication`
  - `mt.mas.sampleapplication.data`
  - `mt.mas.sampleapplication.gui`
- Create dedicated windows for particular functionalities (`JInternalFrame`)
- Each business functionality should be placed in a separated method (no business code in event handlers!).
- Use the `ProgressWorker` class to show the progress of the operation.

The MDI Application

Demonstration of using NetBeans

The Summary

- Visual GUI editors really simplifie the development of the GUI.
- To implement long-running operation in a right way we need to employ threads.
- It could be helpful to use the `SwingWorker` class or its modified version the `ProgressWorker` class.
- It is a good idea to create complex applications as the MDI apps.

Source files

Download source files for all MAS lectures

http://www.mtrzaska.com/plik/mas/mas-source-files-lectures