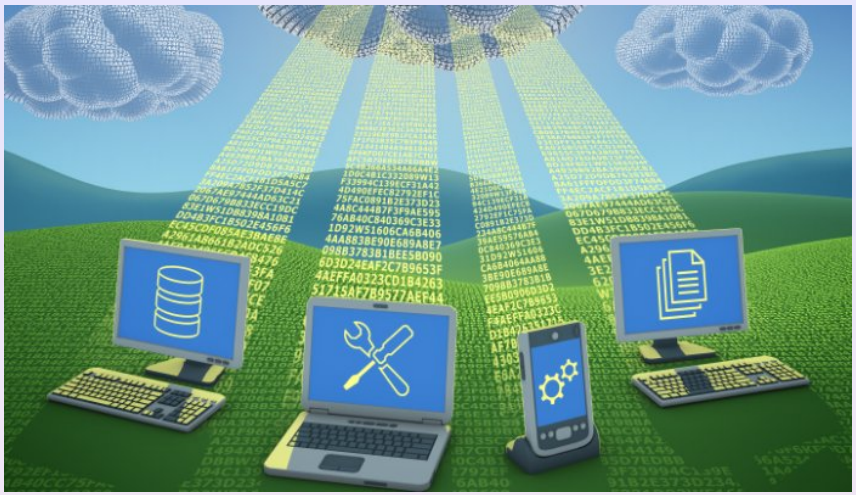


Java and Android Concurrency

# Concurrency and GUIs


fausto.spoto@univr.it


git@bitbucket.org:spoto/java-and-android-concurrency.git

git@bitbucket.org:spoto/java-and-android-concurrency-examples.git

# Graphical Applications

Graphical User Interfaces (GUI) interact with the user through widgets

- windows
- buttons
- labels
- text fields
- sliders
- menus

The Swing and Android libraries implement such components through classical design patterns: strategy, composite, decorator ...

For now, let us consider the simpler case of Swing programming

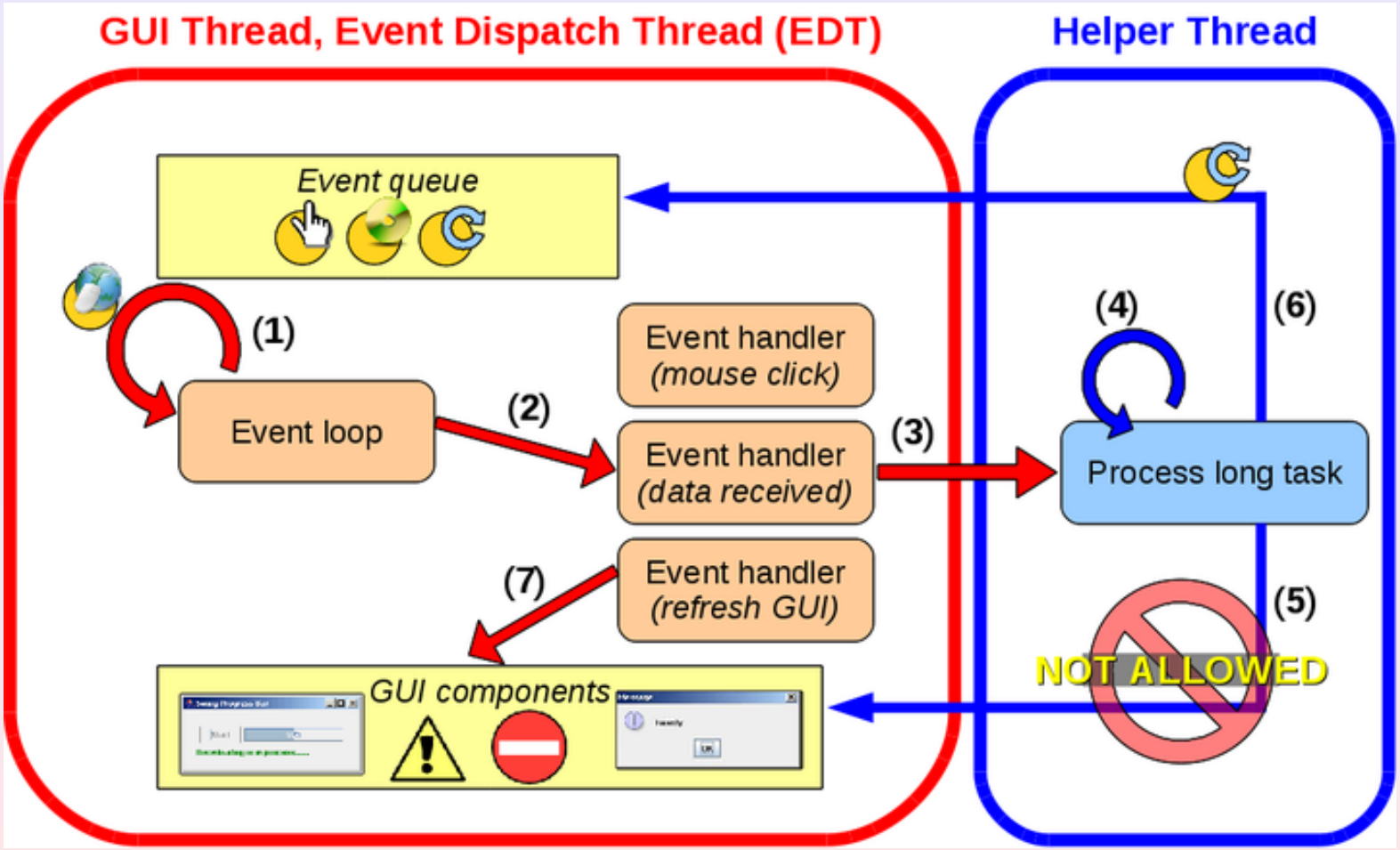
# Why Concurrency is Related to GUI Programming

## GUI data structures are typically non-thread-safe

- they cannot be used by more threads concurrently
- a special thread takes care of their management (main thread, user interface thread, event dispatch thread (EDT), graphical thread. . . )
- long tasks cannot be executed in the user interface thread, but must be offlined to worker threads
- when worker threads complete, they usually have to schedule events for the EDT

# GUI Libraries Use Thread Confinement

GUI data structures are confined to the EDT:




Picture taken from `http://parallel.auckland.ac.nz/ParallelIT/PT_QuickStart.html`

# Communication between EDT and Worker Threads

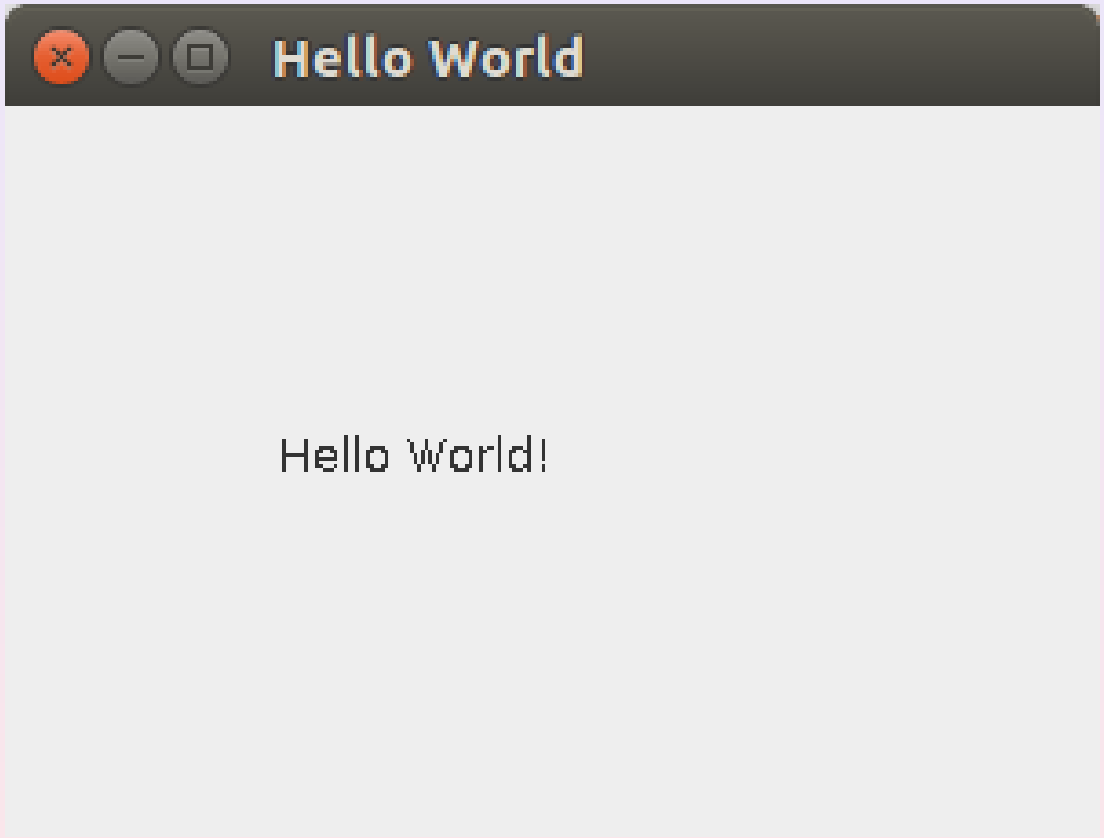
The EDT can schedule tasks on a worker thread by using any technique for spawning concurrent executions, such as:

- create and start a `Thread`
- submit the task to an executor

The worker thread asks the EDT to run a task by calling the method

```
void java.awt.EventQueue.invokeLater(Runnable task)
```

# A First Example of the Use of Swing



# Opening a New Window in Swing

```
public class HelloWorldMain {

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

            @Override
            public void run() {
                JFrame frame = new HelloWorldFrame();
                frame.setTitle("Hello World");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

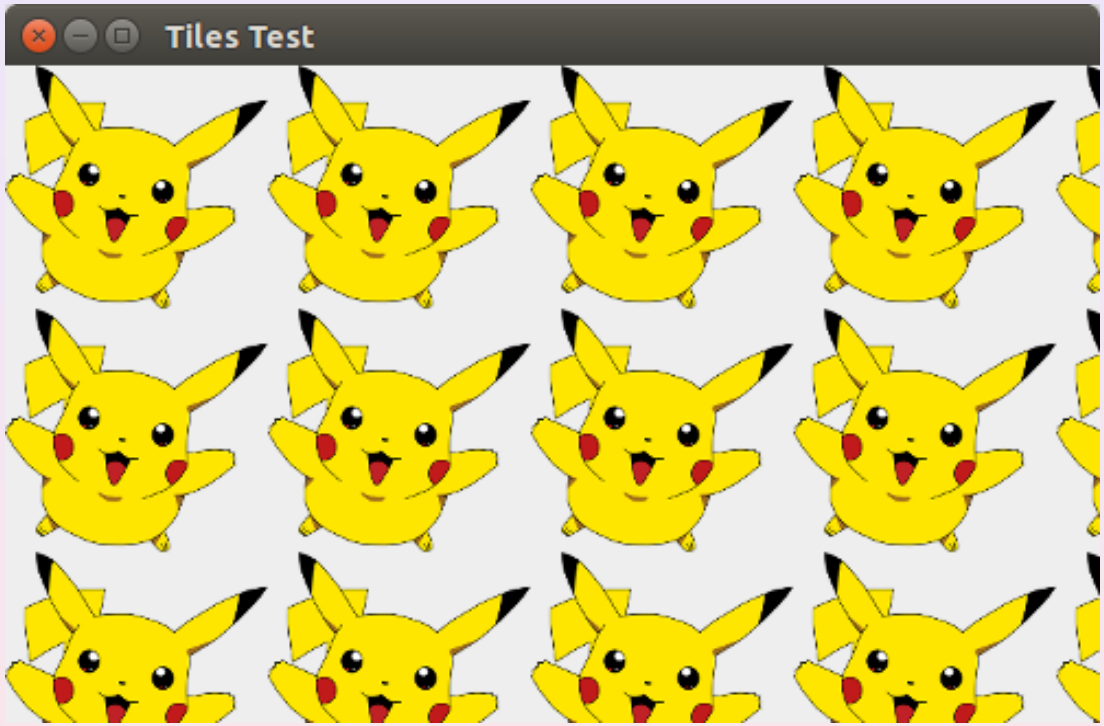
# A Frame Implements a Window and Contains Components

```
public class HelloWorldFrame extends JFrame {

    public HelloWorldFrame() {
        add(new HelloWorldComponent());
        pack();
    }
}
```

# A Component Knows how to Draw Itself

```
public class HelloWorldComponent extends JComponent {
    public final static int MESSAGE_X = 75;
    public final static int MESSAGE_Y = 100;
    public final static int DEFAULT_WIDTH = 300;
    public final static int DEFAULT_HEIGHT = 200;

    @Override
    protected void paintComponent(Graphics g) {
        g.drawString("Hello World!", MESSAGE_X, MESSAGE_Y);
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}
```

# A Second Example of the Use of Swing



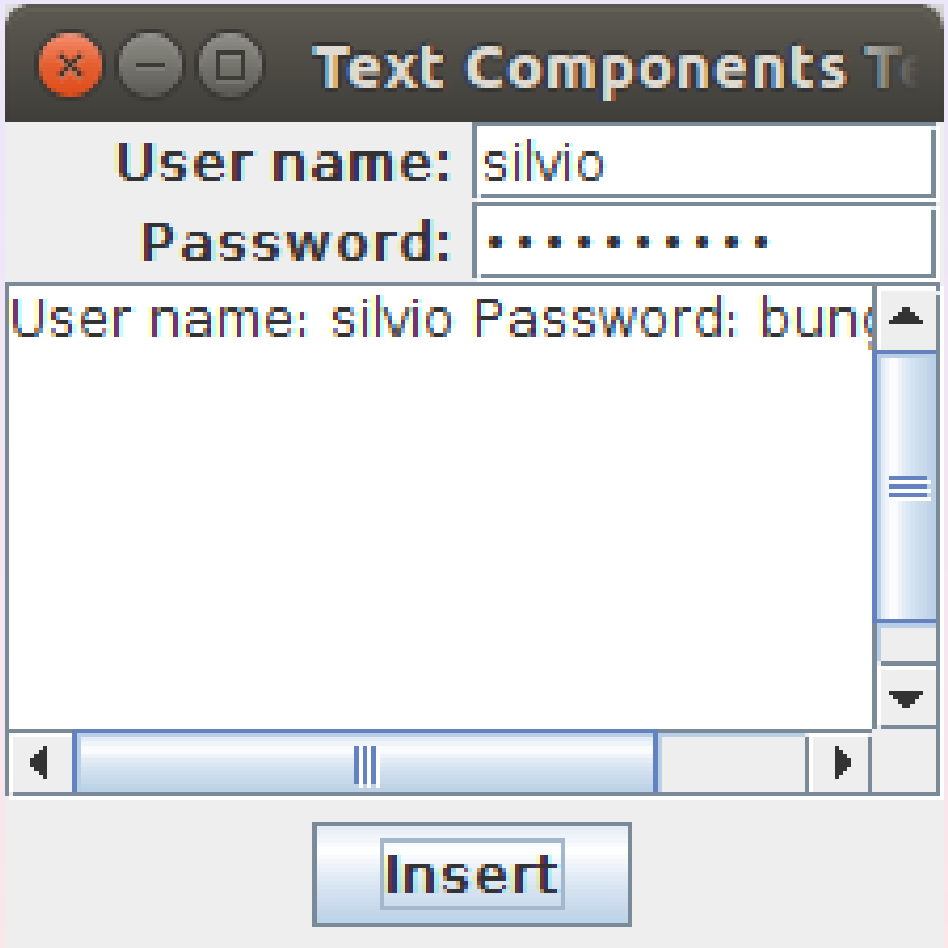
# A Second Example of the Use of Swing

```
public class TilesComponent extends JComponent {
    public final static int DEFAULT_WIDTH = 500;
    public final static int DEFAULT_HEIGHT = 300;
    private final Image image;

    public TilesComponent() {
        image = new ImageIcon("img/pika.png").getImage();
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
```

# A Second Example of the Use of Swing

```
@Override
protected void paintComponent(Graphics g) {
    if (image == null)
        return;

    int imageWidth = image.getWidth(this);
    int imageHeight = image.getHeight(this);

    g.drawImage(image, 0, 0, null);

    for (int i = 0; i <= getWidth(); i+= imageWidth)
        for (int j = 0; j <= getHeight(); j+= imageHeight)
            g.copyArea(0, 0, imageWidth, imageHeight, i, j);
}
```

# A Third Example of the Use of Swing: Library Components



# TextFields, Labels, Container Panels...

```
public class TextComponentFrame extends JFrame {
    public static final int TEXTAREA_ROWS = 8;
    public static final int TEXTAREA_COLUMNS = 20;

    public TextComponentFrame() {
        JTextField textField = new JTextField();
        JPasswordField passwordField = new JPasswordField();

        JPanel northPanel = new JPanel();
        northPanel.setLayout(new GridLayout(2, 2));
        northPanel.add(new JLabel("User name: ", JLabel.RIGHT));
        northPanel.add(textField);
        northPanel.add(new JLabel("Password: ", JLabel.RIGHT));
        northPanel.add(passwordField);

        // a frame has by default the border layout
        add(northPanel, BorderLayout.NORTH);
```

# Text Areas, Scroll Bar Decorators...

```
JTextArea textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);

add(scrollPane, BorderLayout.CENTER);
```

# Interactive Components

```
JButton insertButton = new JButton("Insert");
insertButton.addActionListener(actionListener);
```

A listener specifies the behavior of the click on the button:

```
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

The Hollywood Principle: Don't call me, I'll call you

# Listeners as Explicit Classes (Name Pollution)

```
    JButton insertButton = new JButton("Insert");
    insertButton.addActionListener
        (new MyListener(textField, textArea, passwordField));
}

private class MyListener implements java.awt.event.ActionListener {
    private JTextField textField;
    private JTextArea textArea;
    private JPasswordField passwordField;

    private MyListener(JTextField textField,
            JTextArea textArea, JPasswordField passwordField) {
        this.textField = textField;
        this.textArea = textArea;
        this.passwordField = passwordField;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        textArea.append("User name: " + textField.getText() +
            " Password: " + new String(passwordField.getPassword()) + "\n");
    }

}
```

# Listeners as Inner Classes (Still too Long)

```
class MyListener implements java.awt.event.ActionListener {
    @Override
    public void actionPerformed(ActionEvent event) {
        textArea.append("User name: " + textField.getText() +
            " Password: " + new String(passwordField.getPassword()) + "\n");
    }
}

JButton insertButton = new JButton("Insert");
insertButton.addActionListener(new MyListener());
```

# Listeners as Anonymous Inner Classes (OK-ish)

```
JButton insertButton = new JButton("Insert");
insertButton.addActionListener(new java.awt.event.ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        textArea.append("User name: " + textField.getText() +
            " Password: " + new String(passwordField.getPassword()) + "\n");
    }
});
```

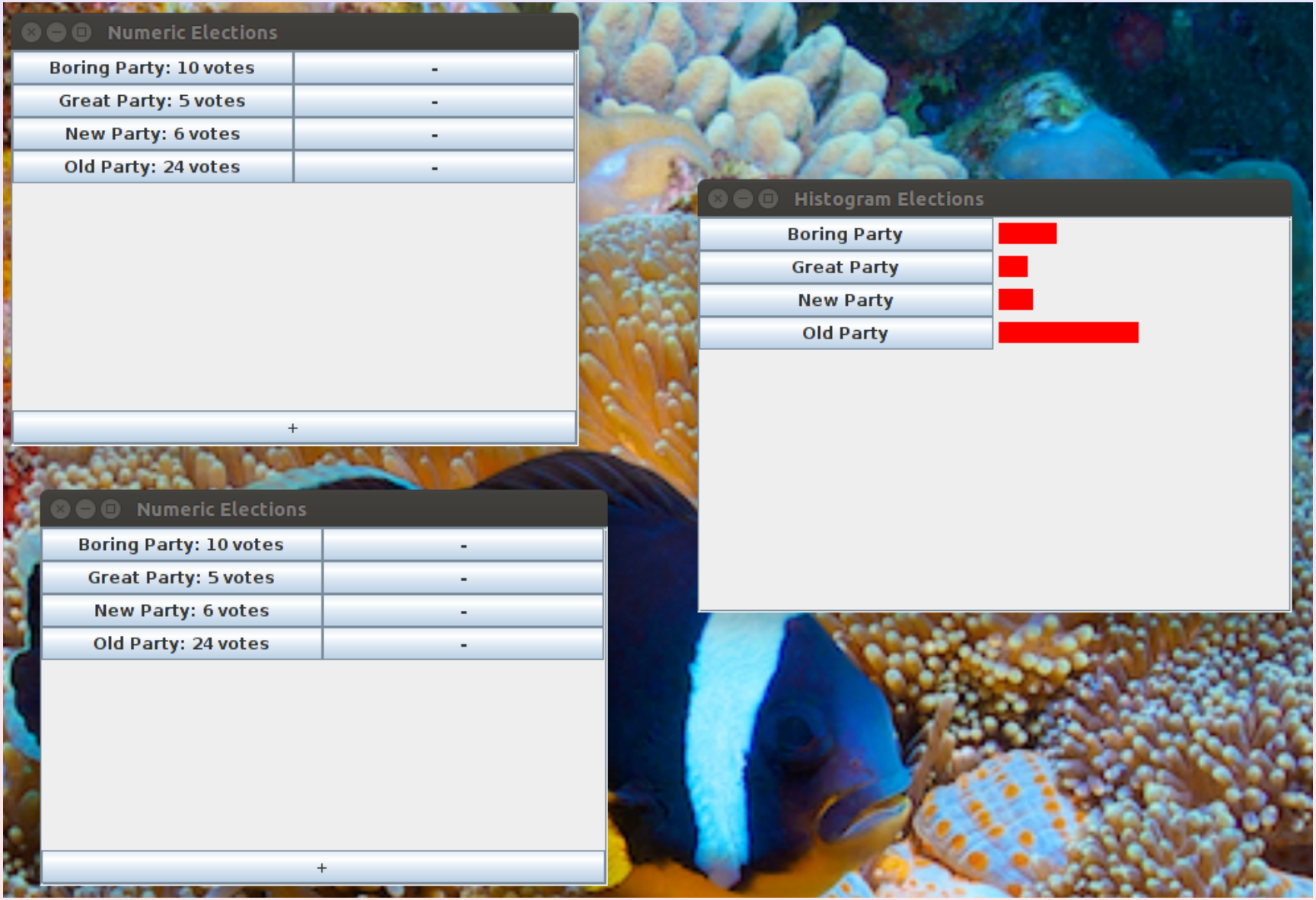
# Listeners as Lambdas (only Java 8) (great!)

```
JButton insertButton = new JButton("Insert");
insertButton.addActionListener(
    event -> textArea.append("User name: " + textField.getText() +
        " Password: " + new String(passwordField.getPassword()) + "\n"));
```

# Buttons

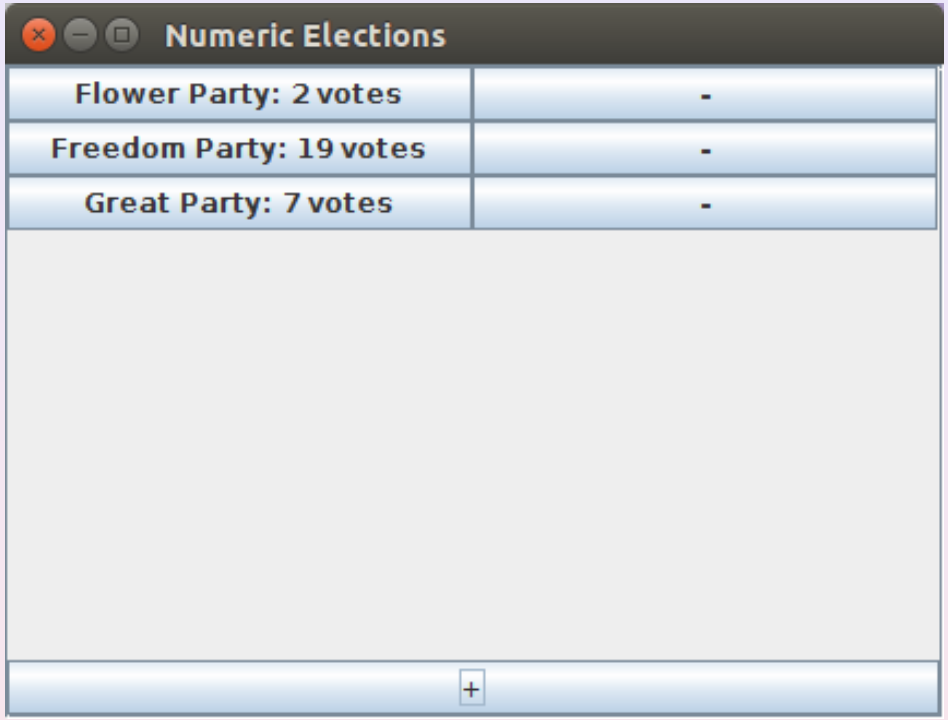
```
JPanel southPanel = new JPanel();
JButton insertButton = new JButton("Insert");
// a panel has by default the flow layout
southPanel.add(insertButton);
insertButton.addActionListener(
    event -> textArea.append("User name: " + textField.getText() +
        " Password: " + new String(passwordField.getPassword()) + "\n"));

add(southPanel, BorderLayout.SOUTH);

pack();
```

# A Serious Swing Application!



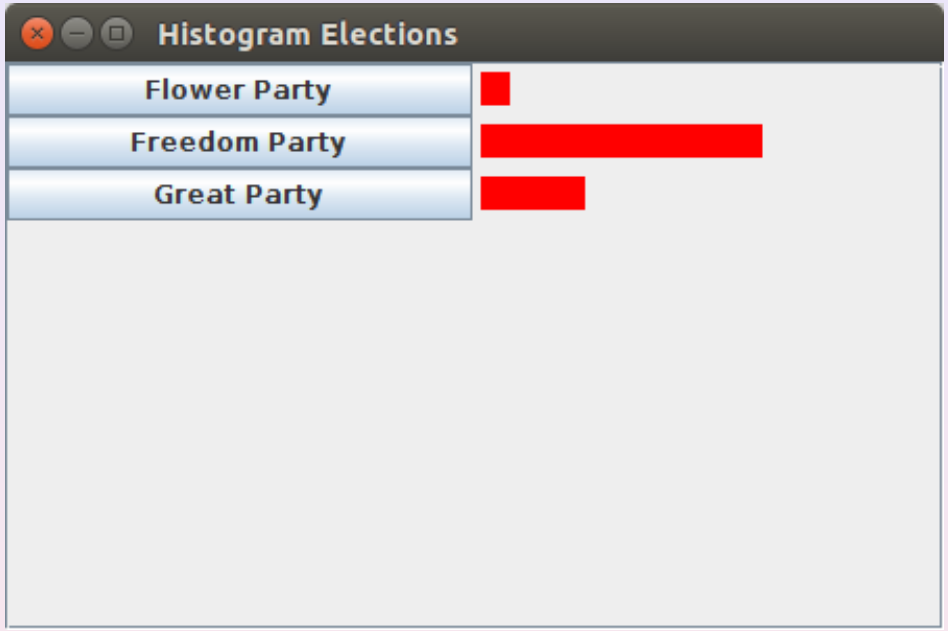
# Elections: First View

We will develop a system with two interfaces, for handling election charts




- by clicking on a party name one can increase its votes
- by clicking on the minus sign, one can remove a party
- by clicking on the plus sign, one can add a new party

# Elections: Second View




- by clicking on a party name one can increase its votes
- no provision for adding/removing parties

# Separation of Concerns

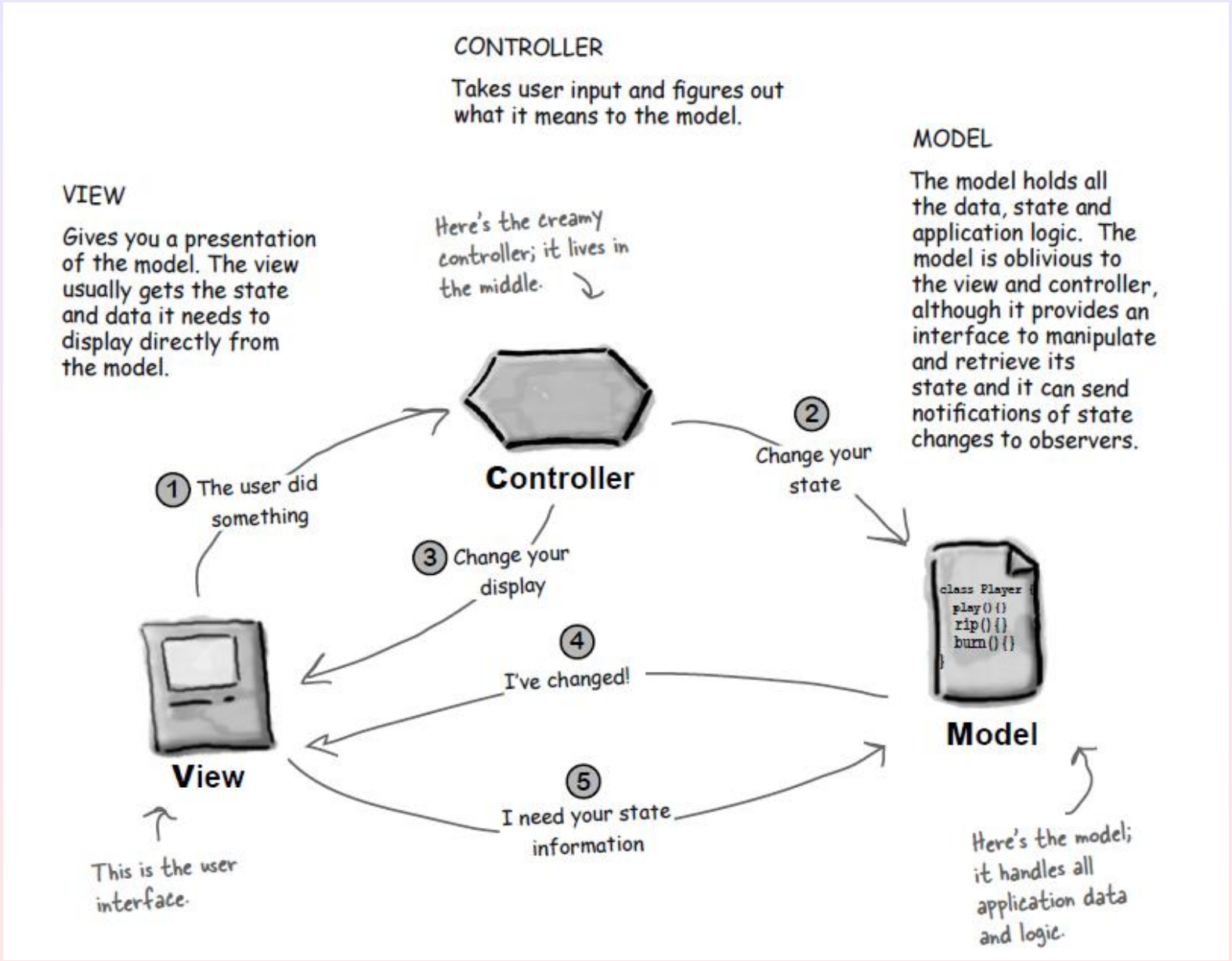
The graphical interface should be kept separate from the logic:

- for distinct versions
- for desktop
- for Android
- for special accessibility

Data should be kept separate from the logic:

- faster on desktop
- more compact on mobile
- kept in a database
- accessible through a web interface

# The Model-View-Controller Design Pattern



# The Organization into Packages

- `model`: data representation
- `view`: game views
- `controller`: data/view coordination

In order to clarify which thread can call which method, we can annotate the latter as follows:

- `@UiThread`: for methods that can only be executed by the EDT
- `@WorkerThread`: for methods that cannot be executed by the EDT

Current compilers do not check such annotations, but static analyzers are starting checking them

# The MVC Triple 1/2

```
@ThreadSafe
public class MVC {
    public final Model model;
    public final Controller controller;
    private final CopyOnWriteArrayList<View> views = new CopyOnWriteArrayList<>();

    public MVC(Model model,Controller controller) {
        this.model = model;
        this.controller = controller;

        model.setMVC(this);
        controller.setMVC(this);
    }

    public void register(View view) {
        this.views.add(view);
    }

    public void unregister(View view) {
        this.views.remove(view);
    }
```

# The MVC Triple 2/2

```java
/**
 * A task that can be executed on all views currently registered.
 */
public interface ViewTask {

    /**
     * Applies the task to the given view.
     *
     * @param view
     */
    void process(View view);
}

/**
 * Applies the given task to all views currently registered.
 *
 * @param task
 */
public void forEachView(ViewTask task) {
    // Internal iteration, preferred since we do not need
    // to expose the modifiable set of views
    for (View view: views)
        task.process(view);
}
```

# The Starting Point of the Program

```
public class Main {
    public static void main(String[] args) {
        EventQueue.invokeLater(() -> {
            MVC mvc = new MVC(new Model(), new Controller());

            new NumericElectionsFrame(mvc).setVisible(true);
            new NumericElectionsFrame(mvc).setVisible(true);
            new HistogramElectionsFrame(mvc).setVisible(true);
        });
    }
}
```

# The Model

```
@ThreadSafe
public class Model {
    private MVC mvc;
    private final Map<String, Integer> votes = new HashMap<>();

    public void setMVC(MVC mvc) {
        this.mvc = mvc;
    }

    // 5: I need your state information
    @UiThread public Iterable<String> getParties() {

    @UiThread public int getVotesFor(String party) {

    // 2: change your state
    @UiThread public void addParty(String party) {

    @UiThread public void removeParty(String party) {

    @UiThread public void addVotesTo(String party, int howMany) {
}
```

# The Model: Implementation

```
// 5: I need your state information
@UiThread public Iterable<String> getParties() {
    // in alphabetical order
    return new TreeSet<>(votes.keySet());
}

@UiThread public int getVotesFor(String party) {
    return votes.get(party);
}

// 2: change your state
@UiThread public void addParty(String party) {
    votes.put(party, 0);
    mvc.forEachView(View::onModelChanged);
}

@UiThread public void removeParty(String party) {
    votes.remove(party);
    mvc.forEachView(View::onModelChanged);
}

@UiThread public void addVotesTo(String party, int howMany) {
    if (votes.containsKey(party)) {
        votes.put(party, votes.get(party) + howMany);
        mvc.forEachView(View::onModelChanged);
    }
}
```

# The Controller

```
@ThreadSafe
public class Controller {
    private MVC mvc;

    public void setMVC(MVC mvc) {
        this.mvc = mvc;
    }

    // 1: the user did something
    @UiThread public void askForNewParty(View view) {
        view.askForNewParty();
    }

    @UiThread public void addParty(String party) {
        mvc.model.addParty(party);
    }

    @UiThread public void removeParty(String party) {
        mvc.model.removeParty(party);
    }

    @UiThread public void registerVoteFor(String party) {
        mvc.model.addVotesTo(party, 1);
    }
}
```

# The View

```
public interface View {
    // 3: change your display
    @UiThread
    void askForNewParty();

    // 4: I've changed
    @UiThread
    void onModelChanged();
}
```

# The View: First Implementation 1/4

```
@ThreadSafe
public class NumericElectionsFrame extends JFrame implements View {
    private final MVC mvc;
    private final JPanel scores;

    @UiThread
    public NumericElectionsFrame(MVC mvc) {
        this.mvc = mvc;
        mvc.register(this);

        setPreferredSize(new Dimension(430, 300));
        setTitle("Numeric Elections");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.scores = buildWidgets();

        onModelChanged();
    }
```

# The View: First Implementation 2/4

```
private JPanel buildWidgets() {
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    JPanel scores = new JPanel();
    scores.setLayout(new GridLayout(0, 2));
    panel.add(scores, BorderLayout.NORTH);

    JButton addParty = new JButton("+");
    addParty.addActionListener(e -> mvc.controller.askForNewParty(this));
    panel.add(addParty, BorderLayout.SOUTH);

    add(new JScrollPane(panel));

    return scores;
}
```

# The View: First Implementation 3/4

```
@Override @UiThread
public void onModelChanged() {
    Model model = mvc.model;

    scores.removeAll();
    for (String party: model.getParties()) {
        JButton label = new JButton(party + ": " + model.getVotesFor(party) + " votes");
        label.addActionListener(e -> mvc.controller.registerVoteFor(party));
        scores.add(label);
        JButton remove = new JButton("-");
        remove.addActionListener(e -> mvc.controller.removeParty(party));
        scores.add(remove);
    }

    pack();
}

@Override @UiThread
public void askForNewParty() {
    new InsertPartyNameDialog(mvc.controller);
}
```

# The View: First Implementation 4/4

```
class InsertPartyNameDialog extends JDialog {

    @UiThread
    public InsertPartyNameDialog(Controller controller) {
        super((Dialog) null);

        setTitle("Insert Party Name");
        setLayout(new FlowLayout());
        add(new JLabel("Insert new party name: "));

        JTextField textField = new JTextField("nome partito");
        textField.addActionListener(e -> {
            String party = textField.getText();
            if (!party.isEmpty())
                controller.addParty(party);

            setVisible(false);
            dispose();
        });
        add(textField);

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

# A Custom Component for Histograms

```
public class Histogram extends JComponent {
    private final float percent;

    /**
     * Builds a component that represents a histogram.
     * The size is given as a percent (0..1) of the total width of 200 pixels.
     *
     * @param percent
     */

    public Histogram(float percent) {
        this.percent = percent;
    }

    @Override
    protected void paintComponent(Graphics g) {
        // this is true in Java 1.2 and later
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.RED);
        g2.fillRect(4, 4, (int) (200 * percent), 16);
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(208, 24);
    }
}
```

# The View: Second Implementation 1/3

```
@ThreadSafe
public class HistogramElectionsFrame extends JFrame implements View {
    private final MVC mvc;
    private final JPanel scores;

    @UiThread
    public HistogramElectionsFrame(MVC mvc) {
        this.mvc = mvc;
        mvc.register(this);

        setPreferredSize(new Dimension(450, 300));
        setTitle("Histogram Elections");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.scores = buildWidgets();

        onModelChanged();
    }
```

# The View: Second Implementation 2/3

```
private JPanel buildWidgets() {
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    JPanel scores = new JPanel();
    scores.setLayout(new GridLayout(0, 2));
    panel.add(scores, BorderLayout.NORTH);

    add(new JScrollPane(panel));

    return scores;
}
```

# The View: Second Implementation 3/3

```
@Override @UiThread
public void onModelChanged() {
    Model model = mvc.model;

    int totalVotes = 0;
    for (String party: model.getParties())
        totalVotes += model.getVotesFor(party);

    scores.removeAll();
    for (String party: model.getParties()) {
        JButton label = new JButton(party);
        label.addActionListener(e -> mvc.controller.registerVoteFor(party));
        scores.add(label);
        scores.add(new Histogram((float) model.getVotesFor(party) / totalVotes));
    }

    pack();
}

@Override @UiThread
public void askForNewParty() {
}
```

# Exercise 1: Generate Votes through an Election Server

Write and publish a servlet with a single method:

`sendvotes?howmany=XXX&parties=P1,P2,...,Pn`

that sends back `howmany` random votes for the given, comma-separated party names. Votes are reported back to the client as a sequence of `howmany` party names from `P1,P2,...,Pn`, each standing for a vote for that party:

```
flower party
great party
PCDD
great party
new party
flower party
great party
...
```

# Exercise 2: Import Votes from an Election Server

Modify both views for the election example, by adding a button *import votes from server* that accesses the previous servlet and registers the random votes generated by the server. This will modify the model and update the views

**You Need a Worker Thread**

The connection to the servlet cannot be executed in the EDT, since it might be slow and is potentially blocking. Spawn a thread instead and report the results back to the EDT at the end

Is it wise to generate an EDT event for each vote that gets read and registered to the model? Is it possible instead to batch all updates into a restricted number of EDT events?

# Exercise 3: Third Implementation of the Elections View

Implement a third version of the view, where the names of the parties are reported in the top part of the frame, in distinct colors, and below is reported a pie chart, with colors corresponding to those of the parties. No provision for adding or removing parties

See a pie chart Swing component at:
`http://blue-walrus.com/2012/09/simple-pie-chart-in-java-swing`