# Praktische Softwaretechnologie

Baher S. Salama and Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation (RISC)

## Java 1.1 Event Model

Karoly.Bosa@jku.at

- Events represent the actions that the user performs
- AWT package: `java.awt.event`
- Components produce events in response to user interaction
- Events can be intercepted by event listeners (implementations of interface `EventListener`)
- Listeners have to "register" to specific events in order to receive them.
- There are many different types of events (and corresponding listeners):
  - `ActionEvent`   `ActionListener`
  - `MouseEvent`   `MouseListener`
  - `WindowEvent`   `WindowListener`
  - `KeyEvent`   `KeyListener`
  - ...
- A listeners that wants to receive the events of a particular component, has to be added to this component's listeners
- Events are instances of `AWTEvent`

## AWTEvent Class

Karoly.Bosa@jku.at

- Superclass of all types of events

**Constructor:**

`AWTEvent(Object source, int id)`

Creates a new event. `source` is a reference to the object that initiated the event. `id` is an integer that represents the type of the event. This constructor is rarely used since events are generated automatically.

**Important instance methods:**

`int getID()`

Returns the ID of the event which represents the event type.

`Object getSource()`

Returns a reference to the object that initiated the event.

`void consume()`

When an event is consumed, it is not sent to the peer object.

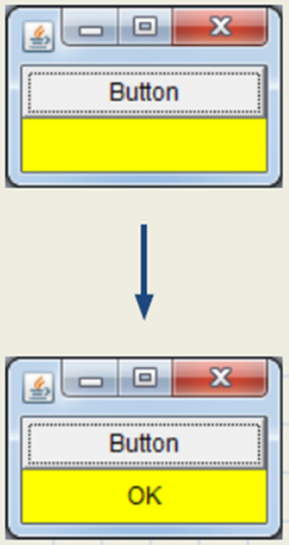
## Events Example

Karoly.Bosa@jku.at

- Frame with 1 button and 1 label
- When the button is pressed, the label should display "OK"
- Two parts: Frame code and Event handling code
- Event handling code needs to have a reference to the label
- When button is pressed it produces an ActionEvent

```
class MyListener implements ActionListener {
     Label l;
     public MyListener(Label l) {
          this.l = l;
     }
     public void actionPerformed(ActionEvent e) {
          l.setText("OK");
     }
}
```

```
...
          Label l = new Label("", Label.CENTER);
          l.setBackground(Color.YELLOW);
          Button b = new Button("Button");
          b.addActionListener(new MyListener(l));
          add(b);
          add(l);
```



## ActionListener and ActionEvent

Karoly.Bosa@jku.at

- `ActionListener` interface Must be implemented by classes that want to handle `ActionEvent`s
- Contains only one method signature:
  - `public void actionPerformed(ActionEvent e);`
- This method is invoked automatically when the button to which the listener is attached is pressed.
- The event-handling code should be written inside this method.
- The method receives an `ActionEvent` instance, which can be used to get more information about the event.
- `ActionEvent` is a subclass of `AWTEvent`
- **Important instance methods of `ActionEvent`:**

  **`Object getSource()`**

  Returns a reference to the component that initiated the event.

  **`String getActionCommand()`**

  The command associated with the object that initiated the event.

## WindowListener Interface

- A listener interface for responding to window events such as:
  - Window opened, closing, closed, iconified, ...
- Defined as follows:

```
public interface WindowListener extends EventListener {

    public void windowActivated(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowOpened(WindowEvent e);
}
```

- A WindowListener implementation must implement all the methods.
- A window listener is added to a Frame using the method:
  - `Frame.addWindowListener(WindowListener l)`

## WindowListener Example

Karoly.Bosa@jku.at

- **Create a blank Frame that exits the program when the close button is pressed.**
- Without a `WindowListener`, the close button is unresponsive.
- When the close button is pressed, the method `windowClosing()` is invoked.

```
import java.awt.event.*;
class ExitListener implements WindowListener {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    public void windowActivated(WindowEvent e) { }
    public void windowClosed(WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowIconified(WindowEvent e){ }
    public void windowOpened(WindowEvent e) { }
}
```

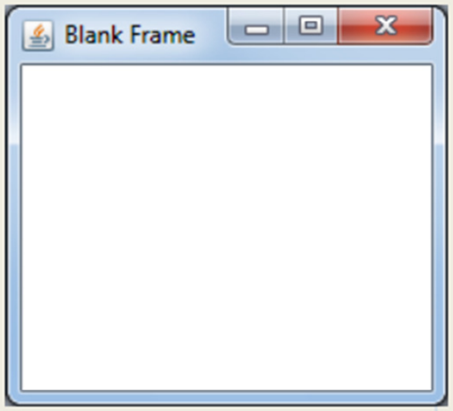
- In order to implement the interface, all interface methods have to implemented, even the unused ones.

## WindowListener Example (continued)

Karoly.Bosa@jku.at

- The listener needs to be attached to a Frame using the method:
  - `void addWindowListener(WindowListener l)`

```
import java.awt.*;
import java.awt.event.*;

public class BlankFrame extends Frame {

    public BlankFrame() {
        super("Blank Frame");
        setSize(220,200);
        addWindowListener(new ExitListener());
        setVisible(true);
    }

    public static void main(String[] args) {
        new BlankFrame();
    }
}
```



## MouseListener Interface

- Interface for listeners that handle mouse events
- Defined as follows:

```
public interface MouseListener extends EventListener {

    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

## Adapters

- Many listener interfaces, such as `WindowListener` and `MouseListener`, declare a large number of methods.
- In many cases, only one (or a few) of these methods is needed.
- But a class that implements an interface must implement its methods.
- Lots of redundant code
- Solution: **Adapters**
  - Abstract classes that implement all the methods of an interface.
- Example: `MouseAdapter`

```
public abstract class MouseAdapter
        implements MouseListener, MouseWheelListener, MouseMotionListener {

    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    void mouseDragged(MouseEvent e) { }
    void mouseMoved(MouseEvent e) { }
    void mouseWheelMoved(MouseWheelEvent e) { }
}
```

## Adapters (continued)

Karoly.Bosa@jku.at

- Instead of implementing the listener, extend the adapter
- Override the needed methods, other methods need not be reimplemented
- Example:

**Listener class:**

```
import java.awt.event.*;
public class ConciseMouseListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        System.out.println("The mouse was pressed: " + e);
    }
}
```

**Adding the listener:**

```
import java.awt.*;
...
    Button b = new Button();
    b.addMouseListener(new ConciseMouseListener());
...
```

## Event Class Hierarchy

Karoly.Bosa@jku.at

- AWTEvent
  - ActionEvent
  - AdjustmentEvent
  - ComponentEvent
    - ContainerEvent
    - FocusEvent
    - InputEvent
      - KeyEvent
      - MouseEvent
    - PaintEvent
    - WindowEvent
  - ItemEvent
  - TextEvent

## Listener Class Hierarchy

- EventListener
    - ActionListener
    - AdjustmentListener
    - ComponentListener
    - ContainerListener
    - FocusListener
    - ItemListener
    - KeyListener
    - MouseListener
    - MouseMotionListener
    - TextListener
    - WindowListener

## Java Applets

Karoly.Bosa@jku.at

- Small Java programs that can be embedded in an HTML page
- Defined as a class that extends `java.applet.Applet`.
- Applet is a subclass of `java.awt.Panel` (inherits all the methods of Panel)
- Class hierarchy of `Applet`:
  - java.lang.Object
    - java.awt.Component
      - java.awt.Container
        - java.awt.Panel
          - **java.applet.Applet**

## Developing Applets

Karoly.Bosa@jku.at

- Create a subclass of `Applet`
- Instead of constructor, override the method `void init()` with the initialization code.
- If needed, override the method `void start()` with code that should be executed when the applet "plays".
- An applet can be treated as a normal panel.
  - Layout can be assigned
  - Components (or other containers) can be added and removed
  - Listeners can be attached to the components
  - etc.
- Extra classes can be created, which are used by the applet
  - E.g.: Listeners, other panels or back-end classes
  - Classpath includes Java Standard library and the folder or package of the applet

## Embedding Applets in HTML

- Applets are embedded in HTML pages using the `<applet>` tag.
- The applet tag can take the following parameters:
  - `code`: URL of the .class file containing the Applet class
  - `width` and `height`: specify the dimensions of the applet
  - `archive`: Optionally, the URL of a JAR file containing the applet classes. If `archive` is specified, then `code` is the name of the main class file.
- Example without archive:

```
<applet code="/path/to/binary/MyApplet.class" width="150" height="
100">
</appet>
```

- Example with archive:

```
<applet code="MyApplet.class" archive="/path/to/archive.jar"
        width="150" height="100">
</appet>
```

## Parameterizing Applets

Karoly.Bosa@jku.at

- HTML code can pass runtime parameters to the applet
- This allows the applet to be customized without having to rewrite code.
- Parameters are passed inside the `<applet>` tag using `<param>` tags.
- The `<param>` tag has only two parameters:
  - `name`: the name of the parameter, and
  - `value`: the value of the parameter
- Both parameters are treated as Strings
- Applet accesses paramters using the method:
  - `String getParameter(String name)`
    - Returns the parameter value with the given `name`, or `null`
- Example:

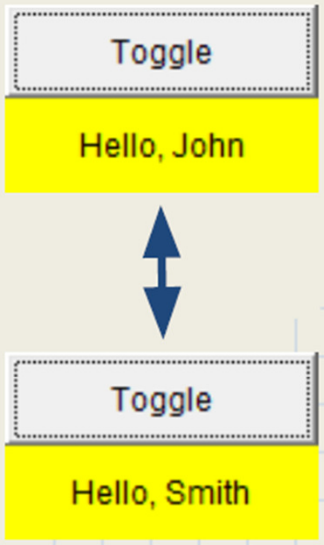
```
<applet code="MyApplet.class" width="150" height="100">
    <param name="greeting" value="Hello" />
    <param name="adressee" value="World" />
</applet>

getParameter("greeting")  ->   "Hello"
```

## Applets Example

Karoly.Bosa@jku.at

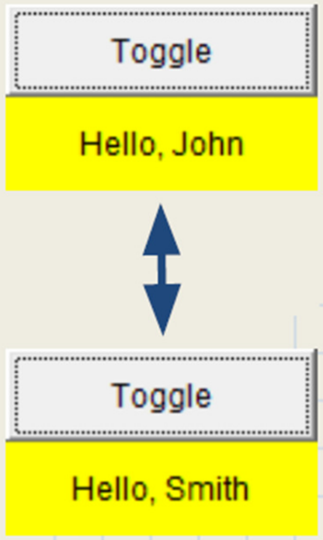
- An applet with a button and a label
- Takes 2 parameters: person1, person2
- Label initially displays "Hello, <person1>"
- When button is pressed, toggles between person1 and person2

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class GreetingApplet extends Applet
        implements ActionListener {
    boolean flag;
    Label l;
    String person1, person2;

    public void toggleGreeting() {
        l.setText("Hello, " + (flag?person1:person2));
        flag = !flag;
    }

    public void actionPerformed(ActionEvent e) {
        toggleGreeting();
    }
    ...
```



## Applets Example (Continued)

```
    ...
    public void init() {
        flag = true;
        person1 = getParameter("person1");
        person2 = getParameter("person2");
        setLayout(new GridLayout(2,1));
        l = new Label("",Label.CENTER);
        l.setBackground(Color.YELLOW);
        toggleGreeting();
        Button b = new Button("Toggle");
        b.addActionListener(this);
        add(b);
        add(l);
    }
}
```



**Applet HTML tag**

```
<applet code="GreetingApplet.class" width="100" height="70"
>
    <param name="person1" value="John" />
    <param name="person2" value="Smith" />
</applet>
```

## Security Restrictions

- By default, applets are loaded in "Sandbox" mode
- This mode offers a number of restrictions of what the applet can do.
- In sandbox mode, an applet:
  - cannot access client resources, e.g. file system, executables,
  - cannot contact a 3rd party server (however, it may contact the server from which it originated)
  - cannot load native libraries
  - can only read secure system properties, all other properties are forbidden
- Applets can request to run in privileged mode only if they are signed.
- In privileged mode, none of these restrictions apply.