Graphical interfaces & event-driven programming

Lecture 12 of TDA 540
Object-Oriented
Programming

Jesper Cockx
Fall 2018

Chalmers University of Technology — Gothenburg University

Last week: Inheritance in Java

Inheritance

Syntax
```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```

The reserved word extends denotes inheritance.

Subclass

Superclass

Declare instance variables that are added to the subclass.

Declare methods that are added to the subclass.

Declare methods that the subclass overrides.

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices

   public void addChoice(String choice, boolean correct) { . . . }

   public void display() { . . . }
}
```

Syntax 9.1
© John Wiley & Sons, Inc. All rights reserved.

Polymorphism

If `S` is a subtype of `T`, an expression of type `S` can be used *wherever* an expression of type `T` is expected.

Inheritance: constructors

Syntax
```
public ClassName(parameterType parameterName, . . .)
{
    super(arguments);
    . . .
}
```

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>;
}
```

The superclass constructor is called first.

The constructor body can contain additional statements.

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

Syntax 9.2

© John Wiley & Sons, Inc. All rights reserved.

Interfaces

Syntax

Declaring:

```
public interface InterfaceName
{
   method declarations
}
```

Implementing:

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

```
public interface Measurable
{
   double getMeasure();
}

public class BankAccount implements Measurable
{
   . . .

   public double getMeasure()
   {
      return balance;
   }
}
```

Interface methods are always public.

Interface methods have no implementation.

A class can implement one or more interface types.

Other BankAccount methods.

Implementation for the method that was declared in the interface type.

Syntax 9.4

© John Wiley & Sons, Inc. All rights reserved.

Kahoot: Inheritance in Java

Graphical user interfaces

GUIs: Graphical user interfaces

Design Preview [ContactEditorUI]

Name

First Name: Last Name:

Title: Nickname:

Format: Item 1

E-mail

E-mail Address: Add

Item 1
Item 2
Item 3
Item 4
Item 5

Edit

Remove

As Default

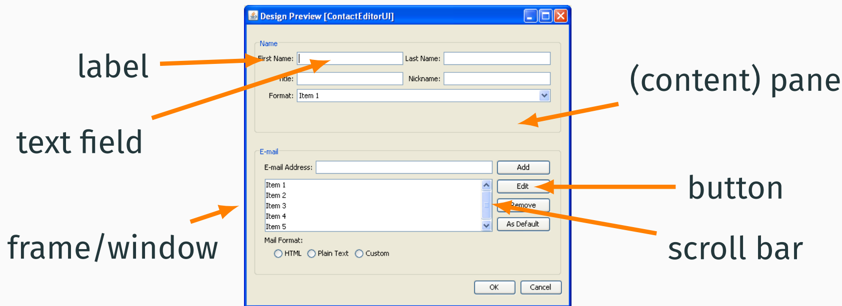
Mail Format:

HTML Plain Text Custom

OK Cancel

Graphical components

A GUI consist of (graphical) components:



Each component corresponds to an object in Java.

GUIs and object-oriented programming

GUI programming is a domain where object-oriented programming shines:

- Graphical components (windows, buttons, scroll bars, ...) are modelled by classes.
- Relations between components are captured by inheritance.
- Polymorphism supports flexible reuse of the different components, without worrying about implementation details.

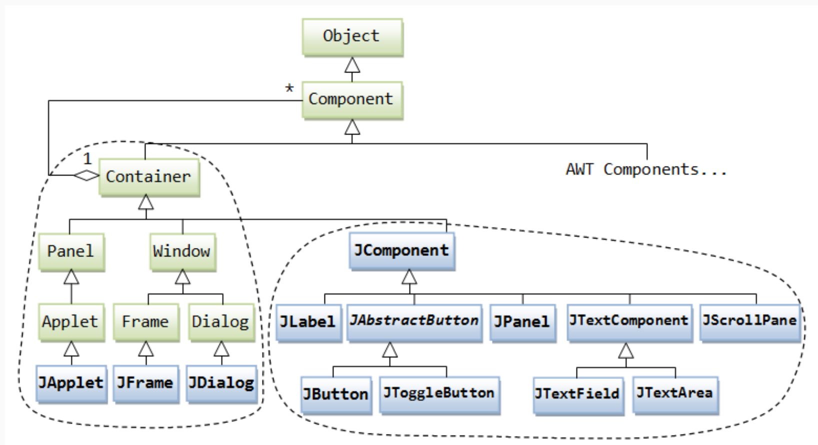
AWT vs. Swing

Java has two main GUI libraries:

- AWT (Abstract Windowing Toolkit)
  - First Java GUI toolkit
  - Native implementation ($\Rightarrow$ very fast)
  - Looks different depending on system
- Swing
  - Java implementation
  - Looks the same on all systems
  - Implemented on top of AWT

We will use mostly Swing, which is newer and has some advantages over AWT.

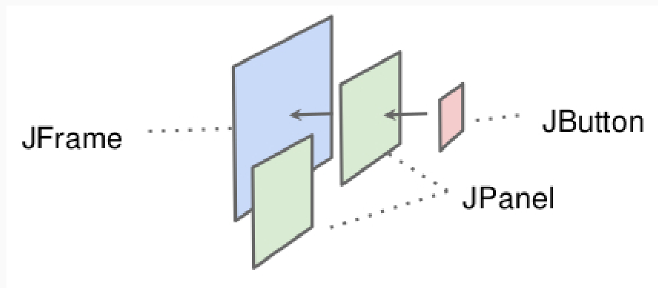
Overview of Swing classes



Swing is huge: 18 packages, 100s of classes.

The structure of a Swing GUI

- Top-level: `JFrame` or `JDialog`
- Secondary components: `JPanel`
- Atomic components: `JButton`, `JTextField`, `JTable`, `JScrollBar`, ...



The `JFrame` class

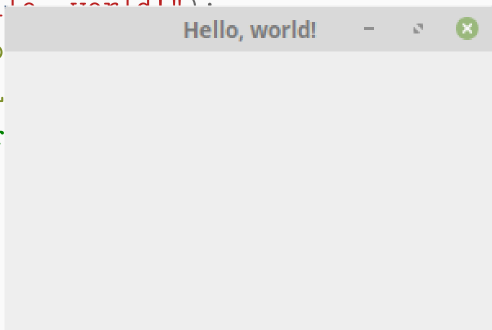
A frame represent one window of a GUI.

```
JFrame frame = new JFrame();
frame.setSize(300,200);
frame.setTitle("Hello, world!");
frame.setDefaultCloseOperation(
  JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

The `JFrame` class

A frame represent one window of a GUI.

```
JFrame frame = new JFrame();
frame.setSize(300,200);
frame.setTitle("Hel
frame.setDefaultClo
  JFrame.EXIT_ON_CL
frame.setVisible(tr
```



The `JPanel` class

A panel can contain several components:
buttons, labels, text fields, ...

The components in a frame are organized into panels.

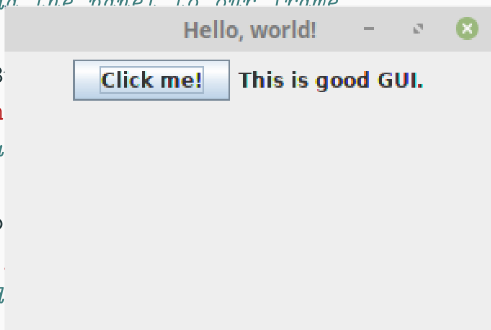
```
JPanel panel = new JPanel();
frame.add(panel); // add the panel to our frame

JButton button = new JButton();
button.setText("Click me!");
panel.add(button); // add button to the panel

JLabel label = new JLabel();
label.setText("This is good GUI.");
panel.add(label); // add label to the panel
```

The `JPanel` class

A panel can contain several components:
buttons, labels, text fields, ...

The components in a frame are organized into panels.

```
JPanel panel = new JPanel();
frame.add(panel); // add the panel to our frame

JButton button = new JB
button.setText("Click m
panel.add(button); // a

JLabel label = new JLab
label.setText("This is
panel.add(label); // ad
```



Customizing `JFrame` with inheritance

A GUI often consists of many components.

To organize these components, we can create a subclass of `JFrame` with all components as (private) attributes:

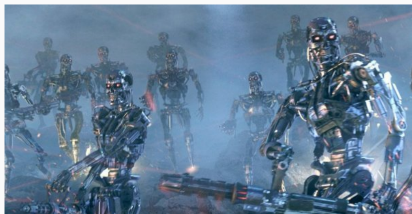
```
class MyFrame extends JFrame {
  private JPanel panel;
  private JButton button1;
  // ...
}
```

Customizing `JFrame` with inheritance

```
public class HelloFrame extends JFrame {
    private JPanel panel; private JButton button; private JLabel label;

    public HelloFrame() {
        setSize(300,200);
        setTitle("Hello, world!");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        panel = new JPanel();
        button = new JButton();
        button.setText("Click me!");
        label = new JLabel();
        label.setText("This is good GUI");
    }

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

Event-driven programming

Event-driven programming



Before:
the *computer* is in control of the program

Now:
the *user* controls the flow of the program himself.

Event-driven programming

In event-driven programming:

- The program listens for external events
- The user decides which events to trigger

Examples of events:

- User clicks a button
- User enters some text
- Mouse moves over a certain area
- ...

The publish/subscribe model

In the publish/subscribe model, components (buttons, ...) act as publishers of events, and special objects called *event handlers* listen and respond to these events.

1. The event handler subscribes to a publisher.
2. When an event is triggered, the publisher notifies all subscribed listeners.
3. When it is notified, the event handler executes some code in response to the event.

The publish/subscribe model in Java

1. Event handlers (= listeners) implement the `ActionListener` interface.

```
public interface ActionListener {
  void actionPerformed(ActionEvent event);
}
```

2. Components (= publishers) offer a method to subscribe to its events.

```
public class JButton { // ...
  addActionListener(ActionListener l) { ... }
}
```

3. When an event is triggered, the component notifies its listeners by calling `actionPerformed`.

Publish/subscribe example

```java
// An action listener that responds to button clicks
class ClickListener implements ActionListener {
  public void actionPerformed(ActionEvent actionEvent) {
    System.out.println("That tickles!");
  }
}

// ...

JButton button = new JButton();
button.setText("Click me!");
panel.add(button);

ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Changing the state of the GUI in response to an event

In theory: publishers and handlers act independently.

In practice: handlers often need to change parts of the GUI.

$\Rightarrow$ the handler needs access to the GUI state.

A button that changes text

```
public class CountClicks implements ActionListener {
  private JButton button;
  private int count;

  CountClicks(JButton button) {
    this.button = button;
    count = 0;
  }

  // whenever a click occurs
  public void actionPerformed(ActionEvent evt) {
    // increment counter
    count++;
    // change the button's text
    button.setText("You clicked " + count + " time(s)");
  }
}
```

Some events and listener interfaces

LISTENER	METHODS	COMPONENTS
ActionListener	actionPerformed	JButton, JComboBox, JTextField, ...
FocusListener	focusGained focusLost	JComponent
MouseListener	mouseClicked mouseClicked mouseEntered mouseExited ...	JComponent
KeyListener	keyPressed keyReleased keyTyped	JComponent
InputMethodListener	caretPositionChanged inputMethodTextChanged	JTextComponent

Inner classes

Inner classes

To avoid passing GUI components to the `ClickListener`, you can move `ClickListener` to *inside* the main class.

```
public class ClickMeFrame extends JFrame {
  private JButton button;

  class ClickListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      button.setText("That tickles!");
    }
  }
  // ...
}
```

This is called an inner class in Java.

Inner classes

An inner class is a class defined in the body of another class.

Each object of the inner class is linked to an object of the surrounding class, and can access its private attributes and methods.

Inner classes cannot include static members (except for constants).

Inner class example

```
public class Top {
  int a;
  class I {
    int three() {
      return 3;
    }
    int getA() {
      return a;
    }
  }
}
```

```
Top t = new Top();
Top.I i = t.new I();
int x = i.three(); // x == 3
t.a = 4;
Top.I j = t.new I();
int y = j.getA();  // y == 4
```

Why use inner classes?

As an alternative to inner classes, we could make `ClickMeFrame` implement `ActionListener`:

```
class ClickMeFrame extends JFrame implements ActionListener {
    public void actionPerformed(ActionEvent event) { ... }

    public ClickMeFrame() {
      JButton button = new JButton();
      button.addActionListener(this);
      // ...
    }
}
```

But this doesn't work very well with multiple events...

$\Rightarrow$ Use a dedicated handler for each event.

Anonymous inner classes

If `I` is an interface, we can declare an anonymous inner class and *immediately* create a single object of this class:

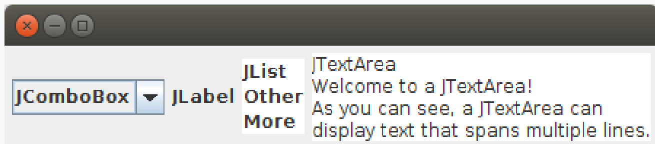
```
public interface I {
  int m();
}

I someObject = new I() {
  int m() {
    // ...
  }
}
```

Anonymous inner class example

```java
public class ClickMeFrame extends JFrame {
  private JButton button;

  public ClickMeFrame() {
    button = new JButton();
    ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        button.setText("That tickles!");
      }
    }
    button.addActionListener(listener);
  }
}
```

Some more GUI components

Partial overview of Swing components



- `JLabel`: simple text label or picture
- `JButton`: clickable button
- `JComboBox`: pull-down menu with mutually-exclusive options
- `JList`: list of selectable options
- `JTextField`: single-line text
- `JTextArea`: multi-line text
- `JScrollPane`: scroll bar
- `JToolBar`: list of clickable buttons
- `JOptionPane`: pop-up dialog box

`JTextField` and `JTextArea`

`JTextField` is a single-line text field.

`JTextArea` is a multi-line text area.

Methods (for both `JTextField` and `JTextArea`):

- `String getText()`
- `void setText(String text)`
- `void append(String text)`
- `void setEditable(Boolean isEditable)`

Live coding: a GUI for `toRobberSpeak` and `toPigLatin`

Implementing your own components

You can define your own components by creating a subclass of `JComponent` and overriding `paintComponent`:

```
public class MyComponent extends JComponent {
  public void paintComponent(Graphics g) {
    // ...
  }
}
```

`paintComponent` is called when frame is first shown, resized, or when repaint() is called.

Using the `Graphics` class

Graphics offers several methods for drawing and filling shapes: `drawRect`/`fillRect`, `drawOval`/`fillOval`, `drawLine`, `drawString`, ...

To change the color used by `Graphics`, call `g.setColor(Color c)`.

Live coding: displaying shapes in a GUI

Layout managers

Layout managers

A layout manager automatically determines the location of components within a panel.

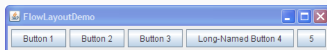
Each layout manager follows a different criterion to position components as they are added to a frame by calling `add`.

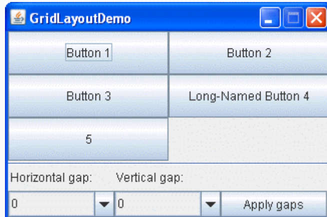
Layout managers provide flexibility:

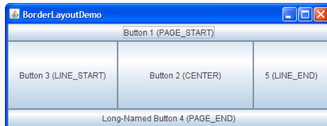
- no absolute positioning
- automatic rearrangement of components when the frame is resized

Some examples of layout managers

- `FlowLayout` (default in Swing): add components in rows
- `GridLayout`: add components in fixed grid
- `BorderLayout`: divide panel in 5 areas (north, south, east, west, center)
- `null`: no automatic layout, have to specify coordinates manually (not recommended)







Multithreading

Heavy computations in GUIs

When implementing a GUI, responsiveness is very important: we don't want the GUI to 'freeze' when doing a long computation.

```
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
     // encode 2-hour video in high resolution
   }
 });
```

Instead, we can run an expensive computation in the background by creating a new thread.

Multithreading with Swing

A `SwingWorker` (a thread in Java) can run a computation in the background without blocking the GUI:

- Create the worker:

```
SwingWorker<String,Object> worker =
  new SwingWorker<String,Object>() {
    public String doInBackground() {
      // ...
    }
  }
```

- Start computation: `worker.execute()`
- Check if computation is finished: `worker.isDone()`
- Get the result after completion: `worker.get()`

Multithreading example

```
public class MeaningOfLifeFinder {
    public static void main(String[] args) {
        final JLabel label = new JLabel();
        SwingWorker<String,Object> worker =
            new SwingWorker<String, Object>() {
                public String doInBackground() {
                    String theMeaning = findTheMeaningOfLife();
                    label.setText(theMeaning);
                    return theMeaning;
                }
            };
        worker.execute();
    }

    private static String findTheMeaningOfLife() {
        for (long i = 0; i < Long.MAX_VALUE; i++) { }
        return "42";
    }
}
```

What's next?

Next lecture:
Reasoning about program correctness.

To do:

- Read the book:
    - Today: chapter 10
    - Next lecture: no specific reading
- Hand in lab #6
- Start on lab #7