

Programming Paradigms
159.272
Swing 101
Introduction to GUI with Swing

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Java Tutorial trail "Creating a GUI With JFC/Swing"

<http://docs.oracle.com/javase/tutorial/uiswing/>

Overview

- user interfaces in Java
- swing components and containers
- layout managers
- event handling
- inner anonymous classes
- simple components: labels, buttons, checkboxes and lists
- dialogs
- advanced components: tables and trees
- pluggable look & feel

User Interfaces in Java: AWT

- the main use case to develop object-oriented programming (Smalltalk!) was to facilitate the development of graphical user interfaces (GUIs)
- GUIs lend themselves to OO development: user interface components can easily be modelled as sets of interacting objects
- the first version of Java already had support for GUI development: the abstract windowing toolkit (AWT)
- AWT classes are in the packages starting with `java.awt`
- in the AWT, components (like buttons) are wrappers around components offered by the OS (like Window)

User Interfaces in Java: Swing

- for this reason, Java 1.2 (1998) added an alternative library to build GUIs: ***swing***
- swing components are natively drawn by Java, and are not mapped to platform components
- this creates a lot of flexibility, but make GUI applications somehow slower than native applications

User Interfaces in Java: Others

- several other GUI libraries for Java are available
- JavaFX is technology focusing on animations and rich internet applications (similar to Flash), it is expected to become part of the standard JDK in version 8
- SWT is a library that uses a design similar to AWT, but only supports a few selected platforms
- SWT is mainly developed by IBM, and used in Eclipse
- there are bindings for platform-independent non-Java libraries like GTK and QT - these bindings provide Java APIs to use these libraries

Swing Containers and Components

- swing components are defined by classes in the `javax.swing` package
- swing components are subclasses of `javax.swing.JComponent`
- swing containers are components that can contain other components

Creating an Empty Window

```
import javax.swing.*;  
..  
JFrame frame = new JFrame();  
frame.setVisible(true);
```

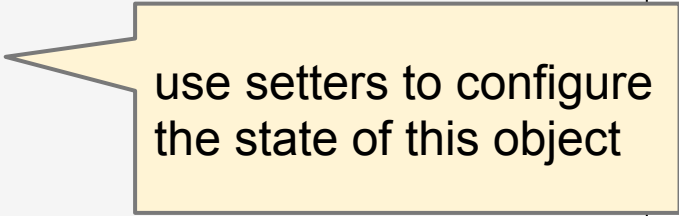
- creates and opens a minimalistic JFrame in the left upper corner of the screen

examples:

<https://bitbucket.org/jensdietrich/oop-examples/src/60c1c898b9e035be9355ba9fea66e079e64f8e9c/swing101/?at=master>

Creating a Better Empty Window

```
import javax.swing.*;  
..  
JFrame frame = new JFrame();  
frame.setTitle("Hello World");  
frame.setLocation(100, 100);  
frame.setSize(300,300);  
frame.setVisible(true);
```



use setters to configure
the state of this object

- creates and opens a JFrame with a title sized 300x300 at position 100x100

JFrame Lifecycle

```
..  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
..
```

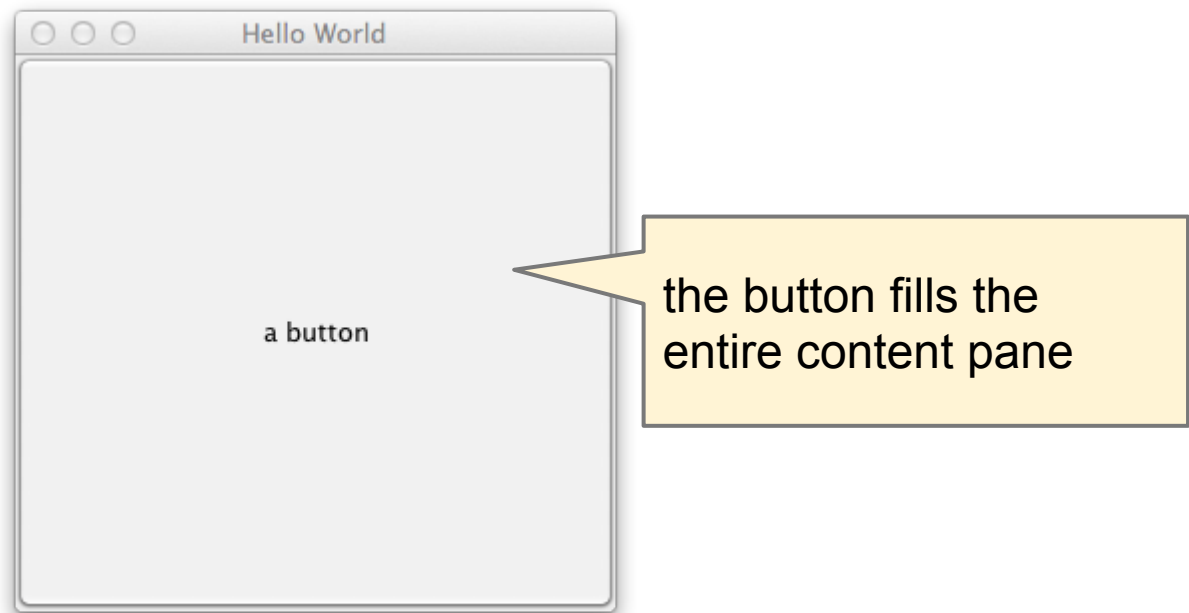
- when the window is closed, the JVM is not terminated!
- to control this, set a **close operation**
- if `JFrame.EXIT_ON_CLOSE` is set, the JVM will terminate

Adding Components

```
JFrame frame = new JFrame();  
frame.setTitle("Hello World");  
frame.setLocation(100, 100);  
frame.setSize(300, 300);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.getContentPane().add(new JButton("a button"));  
  
frame.setVisible(true);
```

- now a new button is added
- note that the button is not directly added to the frame, but to its **content pane** - this is the main area of the frame not including the decoration (title bar)

Adding Components ctd



Layout Managers

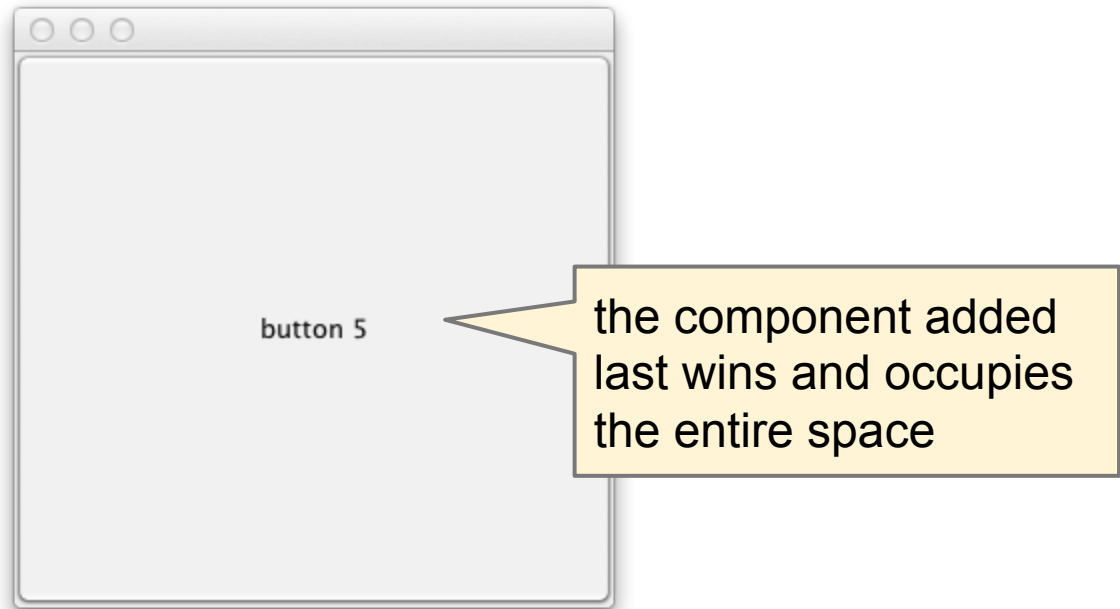
- when multiple components are added, the space of the content pane has to be shared
- hardcoding size and position for each component is possible, but discouraged
- often users resize windows, and then the location and size of components must be recomputed
- for this reason, Java uses **layout managers**
- layout managers decide how to position components, and how to distribute surplus space not needed by components

Using No Layout

```
JFrame frame = new JFrame();  
frame.setLocation(100, 100);  
frame.setSize(300, 300);  
addComponents(frame);  
frame.setVisible(true);
```

- `addComponents()` adds 5 buttons with labels button 1 .. button 5 to the content pane

Using No Layout ctd



Using FlowLayout

```
JFrame frame = new JFrame();  
frame.setLocation(100, 100);  
frame.setSize(300, 300);  
frame.getContentPane().setLayout(  
    new FlowLayout()  
);  
addComponents(frame);  
frame.setVisible(true);
```


Using FlowLayout ctd



components are arranged in a row, and a new line is started if there is not enough space in a row
within a row components are centred (but this can be configured)



when the window is resized, components are rearranged: now they all fit within a row
note that components retain their size !

Using GridLayout

```
JFrame frame = new JFrame();  
frame.setLocation(100, 100);  
frame.setSize(300, 300);  
frame.getContentPane().setLayout(  
    new GridLayout(3, 2, 10, 10)  
);  
addComponents(frame);  
frame.setVisible(true);
```

- the parameters mean the following: 3 rows, 2 columns, 10 pixels horizontal gap, 10 pixels vertical gaps between components

Using GridLayout ctd



components are arranged in a 3x2 grid with 10 pixels space in between



when the window is resized, the position of the components does not change, but the size does

Using BorderLayout

```
JFrame frame = new JFrame();  
frame.setLocation(100, 100);  
frame.setSize(300, 300);  
frame.getContentPane().setLayout(  
    new BorderLayout(10, 10)  
);  
addComponents(frame);  
frame.setVisible(true);
```

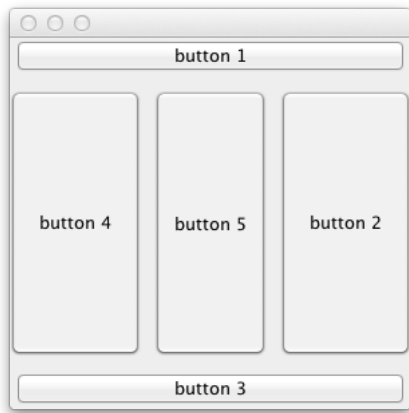
- the parameters mean the following: 10 pixels horizontal gap, 10 pixels vertical gaps between components

Using BorderLayout ctd

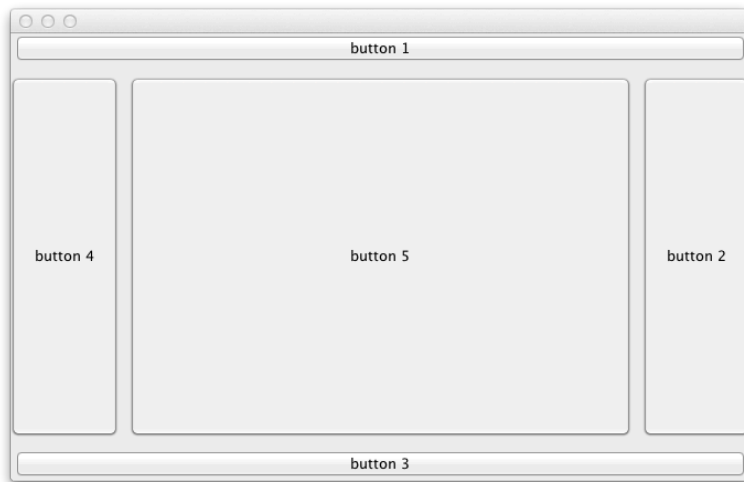
```
static void addComponents(JFrame frame) {  
    JButton button1 = new JButton("button 1");  
    frame.getContentPane().  
    add(button1, BorderLayout.NORTH);  
    ..  
}
```

- to use a BorderLayout, components have to be added with additional **layout constraints**
- the layout constraints are static variables in BorderLayout: CENTER, EAST, WEST, NORTH and SOUTH

Using BorderLayout ctd



in a border layout up to 5 components can be arranged: one in the centre, four around it (north,west,east and south)



when the window is resized, the components retain their positions, all surplus space is given to the component in the middle

Layout Managers ctd

- there are several other layout managers in Java
- `CardLayout` stacks components - only the component on top is visible and occupies the entire visible space
- `GridBagLayout` is a very flexible version of `GridLayout` - components are still arranged in a grid, but constraints can be used to define how components are aligned and how additional space is distributed
- `GridBagLayout` also supports cell merging - components occupying more than one cell

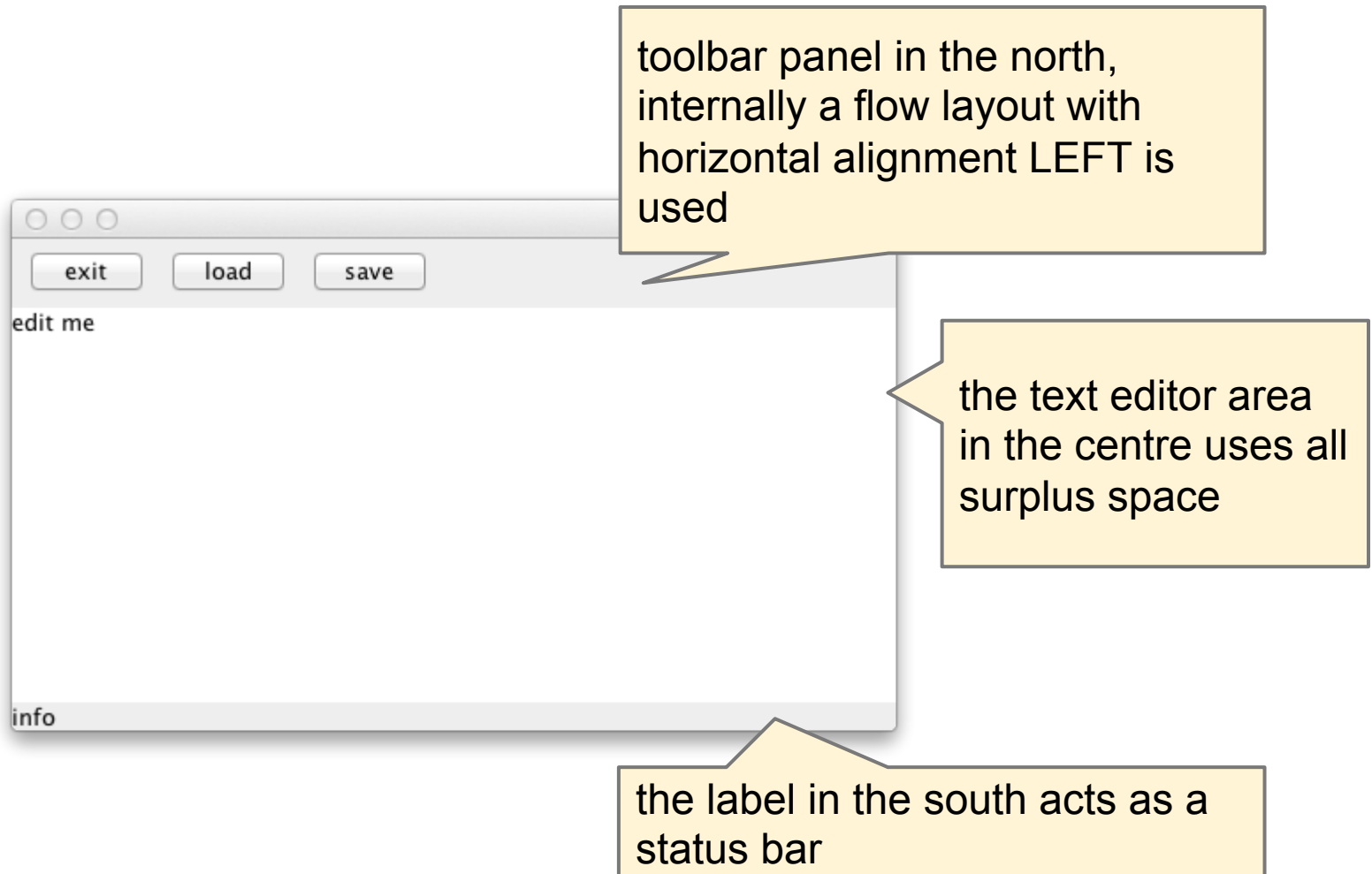
Nested Containers

- layouts are applied to a panel (the content pane), not to the window
- panels can be nested
- nested panels with different layouts can be used to construct custom layouts

Example: Nested Containers

```
JFrame frame = new JFrame();
frame.setLocation(100, 100);
frame.setSize(500,300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().setLayout(new BorderLayout());
JPanel toolbar = new JPanel();
toolbar.setLayout(new FlowLayout(FlowLayout.LEFT));
toolbar.add(new JButton("exit"));
toolbar.add(new JButton("load"));
toolbar.add(new JButton("save"));
frame.getContentPane().add(toolbar,BorderLayout.NORTH);
frame.getContentPane().add(
    new JTextArea("edit me"),BorderLayout.CENTER);
frame.getContentPane().add(new JLabel("info"),BorderLayout.SOUTH);
frame.setVisible(true);
```

Example: Nested Containers etc



Borders

- the layout can be further refined with borders
- all components support the `setBorder` method
- borders are created using the static methods in `javax.swing.BorderFactory`
- example: etched borders, empty borders (add only space around a component), line borders with and without titles, composite borders combining two border types

Implementing Frames

```
public class MyFrame extends JFrame {  
    private JButton exitButton = new JButton("exit");  
    public MyFrame() {  
        super();  
        init();  
    }  
    private void init() {  
        this.getContentPane().setLayout(new FlowLayout());  
        this.getContentPane().add(exitButton);  
        this.setLocation(100,100);  
        this.setSize(300,200);  
    }  
}
```

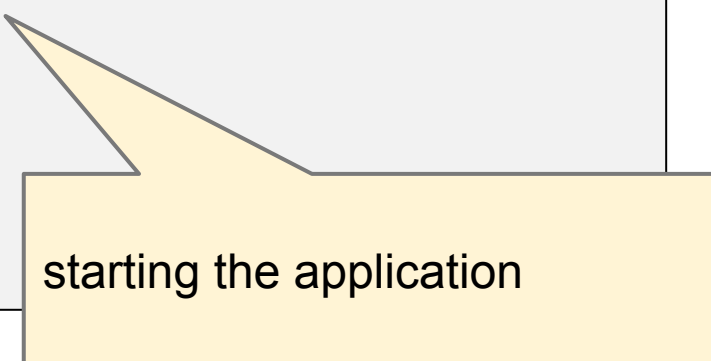
often GUI applications are written by subclassing JFrame

components can be referenced as instance variables - this facilitates event handling

the frame is initialised in the constructor or a method called from the constructor

Implementing Frames ctd

```
public class MyFrame extends JFrame {  
    ..  
    public static void main(String[] args) {  
        MyFrame frame = new MyFrame();  
        frame.setVisible(true);  
    }  
}
```



starting the application

Event Handling

- when users interact with GUI components (clicking or moving the mouse, keystrokes etc), they generate **events**
- these events must be mapped to code
- in Java, this is done through event handlers
- components have methods following this pattern:
`void add*Listener(<ListenerInterface>)`
- the listener interface defines method(s) that are invoked if an event occurs
- these methods usually have a single parameter for an object that represents the event that occurred

Example: ActionListener

interface to listen to
button click events

```
public class MyFrame extends JFrame implements ActionListener
{
    private JButton exitButton = new JButton("exit");
    ...
    private void init() {
        ...
        this.exitButton.addActionListener(this);
    }
    ...

    @Override
    public void actionPerformed(ActionEvent e) {
        this.dispose();
    }
}
```

this is the listener!
start listening to button
click events

code executed when
button is clicked: this
closes the window

Example: ActionListener ctd

```
public class MyFrame extends JFrame implements ActionListener
{
    private JButton exitButton = new JButton("exit");
    ...
    private void init() {
        ...
        this.exitButton.addActionListener(this);
    }
    ...

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == this.exitButton) {
            this.dispose();
        }
    }
}
```

note that this method must be public - this contradicts encapsulation

if multiple buttons exist, conditional code must be used to respond to the click of the correct button

Anonymous Inner Classes

- to simplify situations like this, Java supports **anonymous inner classes**
- an action listener can be instantiated by directly instantiating the interface `ActionListener`
- the constructor must be followed by a block that defines all interface methods
- this implicitly defines a new class named `OuterClassName$<counter>`, e.g. `MyFrame$1`
- this is the class that is actually instantiated

Example: Anonymous Inner Classes

```
public class MyFrame extends JFrame {  
    private JButton exitButton = new JButton("exit");  
    ...  
    private void init() {  
        ...  
        this.exitButton.addActionListener(  
            new ActionListener() {  
                @Override  
                public void actionPerformed(ActionEvent e) {  
                    dispose();  
                }  
            }  
        );  
    }  
    ...  
}
```

this implicitly defines a new class !

note that dispose() is a member of the outer class (MyFrame) !

Anonymous Inner Classes and this

- note that the anonymous inner class calls `dispose()`
 - a private member of the outer class
- inner classes have access to `this` (the current instance of the inner class) but also to the outer `this`, and method lookup uses both

Event Handling in JavaScript

- other programming languages have a more elegant solution using functional features
- for instance, in JavaScript an event handler can be added as follows:

```
button.onClick = function() {  
    // do something  
}
```

- this motivates the use of functional features in Java, introduced in Java 8:

```
button.addActionListener(  
    e ->System.out.println("clicked"));
```

Swing Component Overview

- Swing contains many standard components and containers
- additional libraries (free and commercial) add more components to swing, e.g. charts
- for a good overview of the various components, and how to use them, use the **SwingSet demo** (distributed as part of the JDK, there is also a copy on stream)
- to start the swing demo, double click `SwingSet2.jar`, or run `java -jar SwingSet2.jar`

Simple Components - Overview

- **labels** (`JLabel`) are used to represent text labels (text property)
- labels can also display images (icon property)
- labels can render HTML (e.g. to highlight parts of the text)
- **check boxes** (`JCheckBox`) can be used to check options
- **radio buttons** (`JRadioButton`) can be used to make selections, they are often grouped together using a `ButtonGroup`, an `ItemListener` can be added to get notifications when the selection is changed

Simple Components - Overview ctd

- **text fields** (`JTextField`) are used to edit text
- **input verifiers** (`InputVerifier`) can be used to check whether the text entered is valid (e.g. if text needs to be converted to a number)
- **text areas** (`JTextArea`) and **editor panes** (`JEditorPane`) are other components that can be used to edit multi-line text
- editor panes also support editing of **rich text** (html and rtf)
- **lists** (`JList`) and **combo boxes** (`JComboBox`) can be used to make selections from a list of items
- lists, combo boxes, tables and trees support **cell renderers** - i.e. the items cannot just be text but other components as well (such as labels with icons)

Simple Containers Overview

- **simple panels** (`JPanel`) are plain containers that have layouts (`LayoutManager`) and contain other components (including nested containers)
- **scrolled panes** (`JScrollPane`) are containers with scroll bars - they are useful if there is not enough space to display the components contained
- **split panes** (`JSplitPane`) are containers with two components separated by a vertical or horizontal divider that can be moved by the user
- **tabbed panes** (`JTabbedPane`) are containers that organise components as notebooks with visible labels ("tabs" - usually on top)

Standard Dialogs

- `JOptionPane` has static methods to "pop up" message dialogs: warning, confirmation and information dialogs
- `JFileChooser` can be used to open dialogs to select files (instances of `java.io.File`)
- `JColorChooser` can be used to select colours (instances of `java.awt.Color`)

Advanced Components: Trees and Tables

- swing components work with a data model - a data structure that organises data in a way suitable for visualisation
- these models also emit events - the visual components listen to these events, and update the user interface if data change
- when using complex components like trees and tables, it is crucial to understand these models

Example: FileSystemViewer

```
public class FileSystemTreeModel implements TreeModel {
    public Object getRoot() {return new File(".");}
    public Object getChild(Object parent, int index) {
        File file = (File)parent;
        return file.listFiles()[index];
    }
    public int getChildCount(Object parent) {
        File file = (File)parent;
        return file.listFiles().length;
    }
    public boolean isLeaf(Object node) {
        File file = (File)node;
        return !file.isDirectory();
    }...
}
```

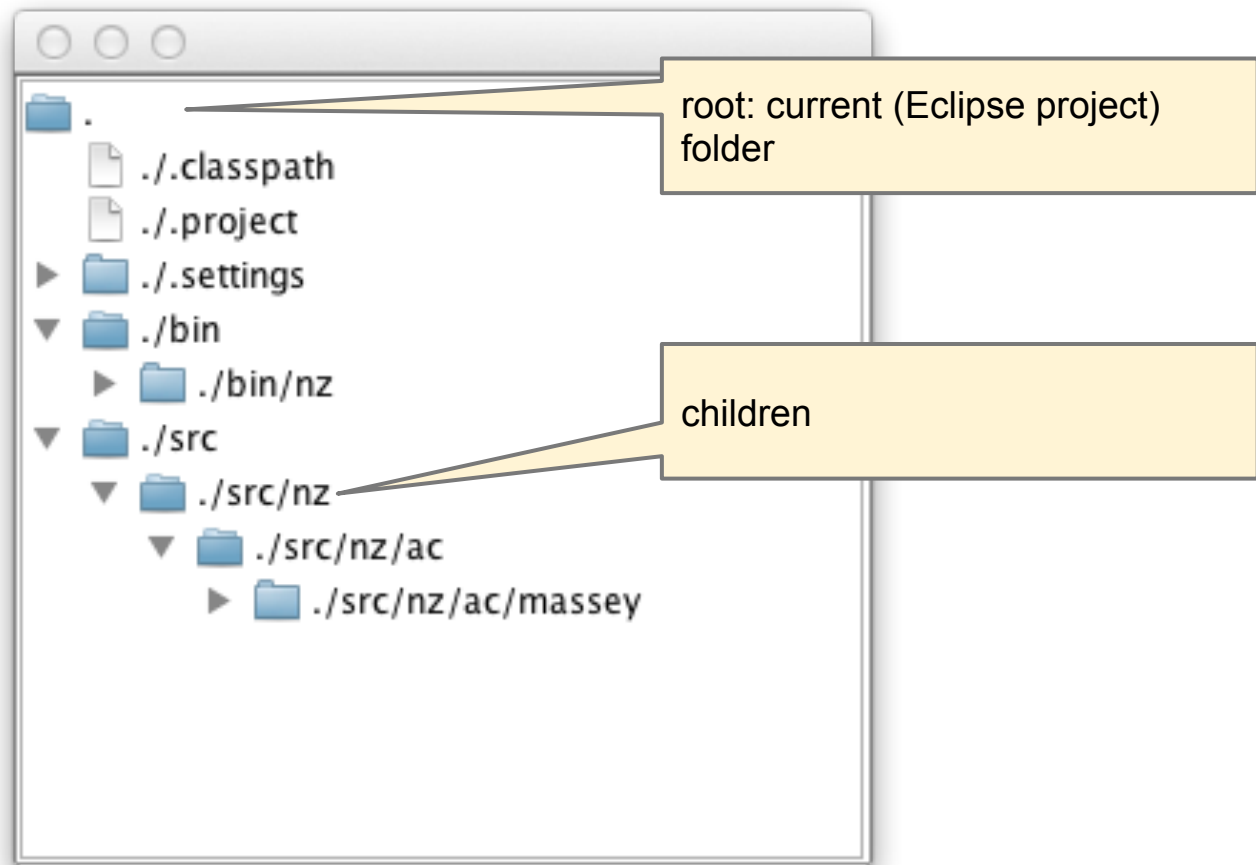
FileSystemTreeModel

- `FileSystemTreeModel` implements the `javax.swing.tree.TreeModel` interface
- is uses the API in `java.io.File` to describe the hierarchical structure of the file system

Example: FileSystemViewer ctd

```
public class FileSystemViewer extends JFrame {
    JTree tree = new JTree();
    public static void main(String[] args) {
        FileSystemViewer win = new FileSystemViewer();
        win.setVisible(true);
    }
    public FileSystemViewer() {
        super();
        init();
    }
    private void init() {
        this.getContentPane().add(tree);
        ..
        tree.setModel(new FileSystemTreeModel());
    }
}
```

Example: FileSystemViewer ctd



Pluggable Look & Feels

- swing does not use native components
- however, it is often desirable to make Java GUI applications look like native applications
- in swing, this is achieved through look and feels
- a look and feel is a class that implements `javax.swing.LookAndFeel`
- the look and feel can be changed at **runtime**:
`javax.swing.UIManager.setLookAndFeel (LookAndFeel)`

Pluggable Look & Feels

- look and feels for windows, mac and a platform-independent "Java look and feel" exist, and there are many free or commercial additional look and feels
- `UIManager` has methods to return the (current) system look and feel and the cross-platform look and feel
- see also:
<http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html>
<http://www.javootoo.com/> (listing of third party look and feels)