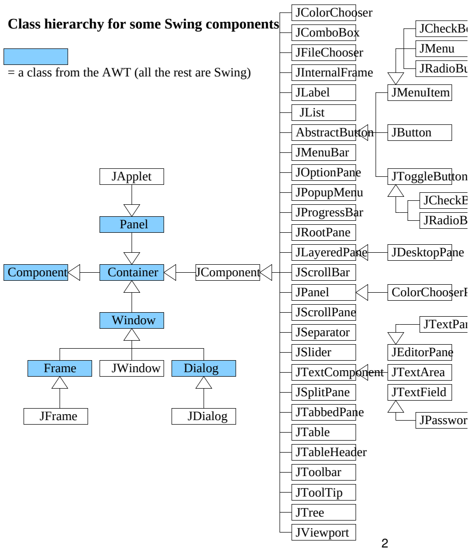
Lecture 37

We take a birds-eye view of a good number of Swing's different components and the way they fit into a class hierarchy. In particular, we draw the distinction between top-level containers (which are 'heavyweight components') and subclasses of `JComponent` (which are 'lightweight components').

We discuss some of the methods that are common to all `JComponent`s. And we investigate some of the ways you can customise the appearance of `JComponents`. This includes discussion of colours, fonts, borders and images.

The ideas are used in a burger bar program. This program again keeps model (application), view (presentation) and controller (translation) separate.



Containers in a GUI

- GUI programs have at least one *top-level container:* `JFrame`, `JApplet`, `JDialog`
- Other components are added to the *content pane* of the top-level container (and positioned by layout managers)
- Some of the other components may be *general-purpose intermediate containers* used for grouping/layout, e.g. `JPanel`, `JScrollPane`, `JSplitPane`, `JTabbedPane`, `JToolBar`, `JInternalFrame`, `JLayeredPane`, `JRootPane`
- (Top-level containers are 'heavyweight components', rendered as windows on the screen; other components are 'lightweight components', drawn ('painted') onto the heavyweight components)

Methods inherited into components

- Methods to customise appearance (`get/setBorder/ Font/ Foreground/ Background`)
- Methods affecting component state (setting tooltips, enabling/disabling, making visible/invisible)
- Methods to register listeners (for general events such as mouse being moved over component)
- Methods for displaying the component ('painting')
- Methods to add other components (`add, remove`)
- Methods for layout (`get/setLayout/ PreferredSize`)
- Methods dealing with absolute sizes & positions (`get/setSize/ Location/ Bounds`)

Custom painting. You don't really need it, do you?

- Suppose you think you want to draw something 'special' on the screen
- Before you resort to custom painting, be sure that what you want is really not catered for by Swing
- Avoiding custom painting…
- You can choose from numerous components, e.g. `JSeparator`; `JTree`; `JTable`; `JEditorPane` & `JTextPane` (for text, incl. HTML & RTF, with mixed fonts, embedded images), …
- You can add any component to any other (since they're all containers), e.g. add a `JTextArea` to a `JButton`

Avoiding custom painting

- You can set background & foreground colours of your components (`setForeground`, `setBackground`)
- You can set the borders of your components
- You can choose the font for text (e.g. on `JButton`s, `JLabel`s, etc.)
- You can display images ('icons') on several Swing components, e.g. `JButton`s, `JLabel`s
- You might be able to replace the existing appearance by some other appearance by exploiting Swing's pluggable look-and-feel
- If you still think you need custom painting, see next lecture

Fonts

- `Font`'s constructor takes three arguments:
  - the font name: use logical font names ("Serif", "SansSerif", "Monospaced", "Dialog" or "DialogInput");
  - the font style (`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` or `Font.BOLD | Font.ITALIC`);
  - the font size (an `int`, the numbers of 'points', 1/72 of an inch)
- E.g. a fairly common font used in books:

```
Font f = new Font(Serif, Font.PLAIN, 12);
```

- Your platform may not have all fonts; the system's default will be substituted

`Font` information

- If `comp` is a component and `f` is a font:

```
comp.getFont();    comp.setFont(f);
```

- If `f` is a `Font`, some obvious methods:

```
f.getName();  f.getStyle();  f.getSize();
f.isPlain();  f.isBold();    f.isItalic()
```

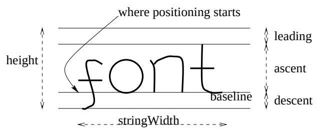
- For further info about a font, class `FontMetrics` defines objects that give us info on the dimensions of `Font` objects

The `FontMetrics` class (simplified)

- If `comp` is a component and `f` is a font, one way to get a fontmetrics object:

```
FontMetrics fm = comp.getFontMetrics(f);
```

- Send, e.g. `getAscent`, `getDescent`, `getLeading`, `getHeight`, to `fm`



- `stringWidth` gives width (in points), e.g.:

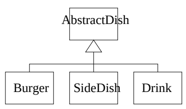
```
int w = fm.stringWidth("font");
```


`ImageIcon`s

- `JLabel`s, `JButton`s & some others can contain objects that implement the `Icon` interface
- `ImageIcon` implements `Icon` and can be used for JPEG or GIF images (including animated GIFs)
  - Constructor specifies file or URL to load from: `new ImageIcon("fileName")`, `new ImageIcon("http://...")` (but actual loading only happens just prior to being displayed)
  - Single-frame images: by default, a `MediaTracker` object ensures they're fully loaded before they are displayed (not good for large images)
  - Multi-frame images (animated GIFs): by default, parts of the image may be displayed before it's fully loaded; an `ImageObserver` makes sure more is displayed as it gets loaded

Derek's Burger Bar

- A `BurgerBar` object has instance variables for
  - its bill-of-fare: this is a `List` of foodstuffs that are available; the foodstuffs are created from this class hierarchy:



  - a `DefaultListModel` object (to keep track of the current customer's order)
- Three-part architecture: application (`BurgerBar`, `AbstractDish`, etc.), presentation (`BurgerBarGUI`) and translation (two anonymous classes in `BurgerBarGUI` and `AddItemToOrderListener`)

JList

Some fragments from the Burger Bar classes:

```
currentOrder = new DefaultListModel();

list = new JList(currentOrder);

public void addItemToOrder(
                        AbstractDish theItem)
{  currentOrder.add(0, theItem);
}
```

You can create a `JList` from a `Vector`, array of `Object`s or a class that implements `ListModel`, e.g. `DefaultListModel`.

If you use the last of these options, *when you update the data structure, e.g. insert/remove, the update automatically propoagates to the the items in the `JList` on the screen!*