Swing

CST Part 1A, Easter 2004

Tim Harris

tim.harris@cl.cam.ac.uk

Introduction to Swing

This handout aims to introduce the terminology used in the Swing libraries for implementing user interfaces. It cannot provide a full reference, so I would suggest looking at some of the following sources:

- Progamming documentation is available on the web. `http://java.sun.com/j2se/1.4.1/docs/api/`
- This includes the Java language specification + details about Java Bytecode and the Java Virtual Machine at `http://java.sun.com/j2se/1.4.1/docs/`
- A set of examples are all in `$CLTEACH/tlh20/swing-examples-2003` on the PWF Linux system
- There's a local newsgroup `ucam.cl.java` (`nntp://ucam.cl.java` in most web browsers, or look at stand-alone news clients e.g. `trn`)

Graphics

- Basic rendering primitives are available on instances of `Graphics`, e.g. using Java applets:

```
import java.awt.*;

public class E1 extends java.applet.Applet
{
  public void paint (Graphics g) {
    g.drawLine (0, 0, 100, 100);
  }
}
```

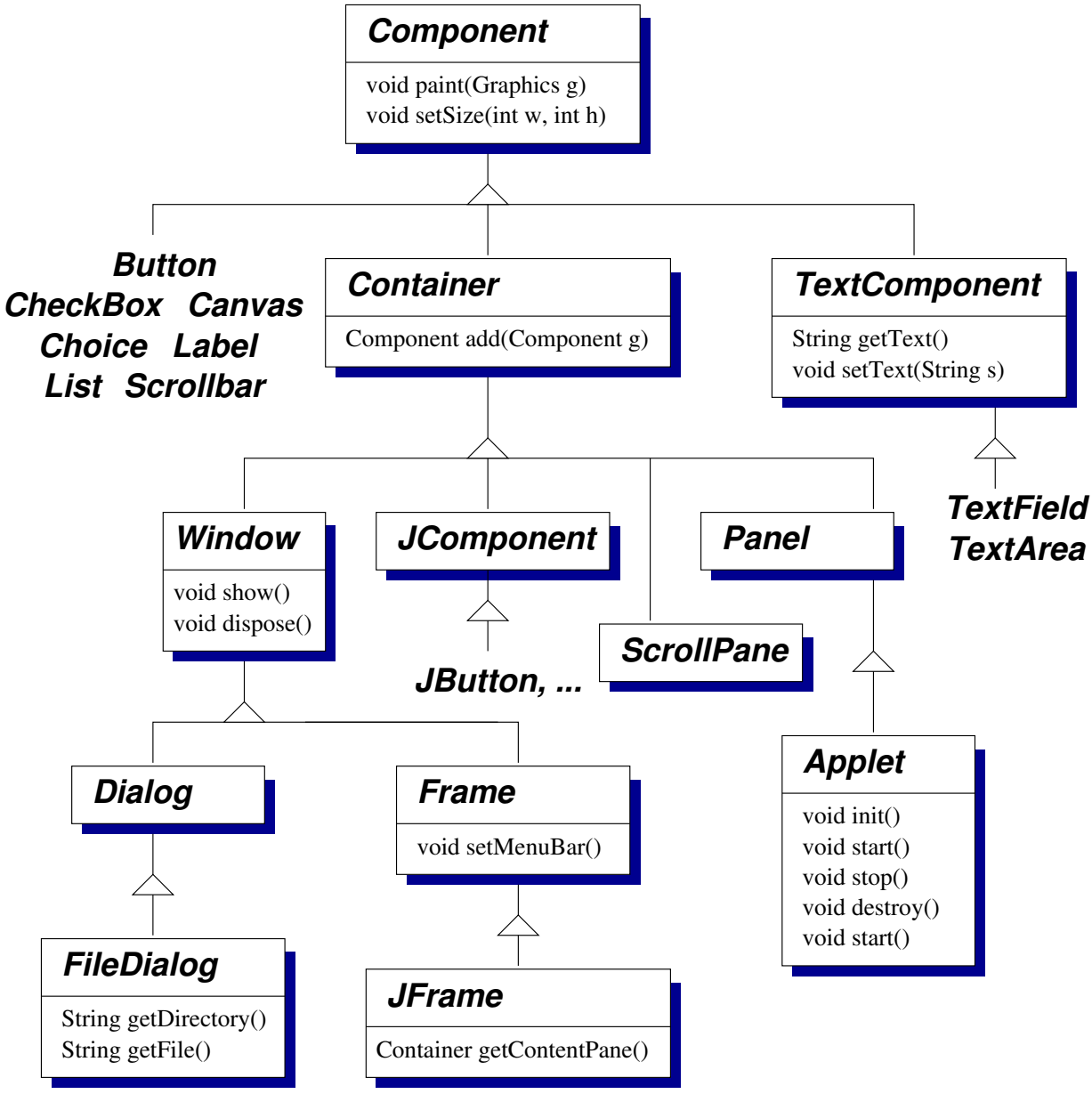
- Simple primitives are available – `setColor`, `copyArea`, `drawLine`, `drawArc`...

- More abstractly, an instance of `Graphics` represents the component on which to draw, a translation origin, the clipping mask, the font, etc

- Translation allows components to assume that they're placed at (0,0)

(Notice the running similarity between these basic functions as X11 / Motif...)

Components



- See the “SwingExamples” demos for illustrations of how to use many of these

Components (2)

- In general a graphical interface is built up from **components** and **containers**
- *Components* represent the building blocks of the interface, for example buttons, check-boxes or text boxes
- Each kind of component is modelled by a separate Java class (e.g. `javax.swing.JButton`). Instances of those classes provide particular things in particular windows – e.g. to create a button bar the programmer would instantiate the `JButton` class multiple times
- As you might expect, new kinds of component can be created by sub-classing existing ones – e.g. sub-classing `JPanel` (a blank rectangular area of the screen) to define how that component should be rendered by overriding its `paintComponent` method:

```
public void paintComponent (Graphics g) {
  super.paintComponent (g);
  ...
}
```

Containers

- *Containers* are a special kind of component that can contain other components – as expected, the abstract class `java.awt.Container` extends `java.awt.Component`
- Containers implement an `add` method to place components within them
- Containers also provide top-level windows – for example `javax.swing.JWindow` (a plain window, without title bar or borders) and `javax.swing.JFrame` (a 'decorated' window with a title bar etc)
- Other containers allow the programmer to control how components are organized – in the simplest case `javax.swing.JPanel`
- In fact, `java.applet.Applet` is actually a sub-class of `Panel`

Containers (2)

- A common design technique is to develop a **spatial hierarchy** of nested containers
- Components are organized within a container under the control of a **layout manager**
- `BoxLayout` is particularly useful: it places a series of components horizontally or vertically
- `Box` provides static methods to create special invisible components:
  - **Rigid-area** components which have a fixed size
  - **Struts** which have fixed height or wdith (used to space out other components)
  - **Glue** which expands/contracts if the window is resized and nothing else can change, e.g.:

```
1       cp.setLayout (
2             new BoxLayout (cp, BoxLayout.X_AXIS));
3       cp.add (Box.createHorizontalStrut (10));
4       cp.add (left);
5       cp.add (Box.createHorizontalGlue ());
6       cp.add (right);
7       cp.add (Box.createHorizontalStrut (10));
```

Receiving input

- An **event-based** mechanism is used for delivering input to applications, broadly following the observer pattern

- Different kinds of event are represented by sub-classes of `java.awt.AWTEvent`. These are all in the `java.awt.event` package. e.g. `MouseEvent` is used for mouse clicks, `KeyEvent` for keyboard input, etc.

- The system delivers events by invoking methods on a **listener**. e.g. instances of `MouseListener` are used to receive `MouseEvent`:

```
public interface MouseListener
       extends EventListener
{
  public void mouseClicked(MouseEvent e);
  ...
}
```

  Components provide methods for registering listeners with them, e.g. `addMouseListener` on `Component`

- `AWTEvent` has a `getSource()` method, so a single listener can disambiguate events from different sources. Sub-classes add methods to obtain other details – e.g. `getX()` and `getY()` on a `MouseEvent`

Input using inner classes

- Anonymous inner classes provide an effective way of handling some forms of input, e.g.

```
addActionListener (new ActionListener () {
  public void actionPerformed (ActionEvent e)
  {
    ...
  }
});
```

- A further idiom is to define inner classes that extend **adapter classes** from the `java.awt.event` package. These provide 'no-op' implementations of the associated interfaces

- The programmer just needs to override the methods for the kinds of event that they are interested in: there is no need to define empty methods for the entire interface

```
addMouseMotionListener
        (new MouseMotionAdapter () {
  public void mouseDragged (MouseEvent e)
  {
    ...
  } // no need to define mouseMoved
};
```