# 25 Object Serialization

- Motivation
  - objects must be independent from application life time
  - objects must be exchanged between applications

# 25.1 Object Streams

- Object streams can read/write objects from/to a byte stream
- Class **`java.io.ObjectOutputStream`**
  - ◆ **`void writeObject(Object o)`**: transitive serialization of an object
- Class **`java.io.ObjectInputStream`**
  - ◆ **`Object readObject()`**: reading a serialized object

# 25.2 Example

- Save a String and a **`Date`** object

```
FileOutputStream f = new FileOutputStream("/tmp/objects");
ObjectOutput s = new  ObjectOutputStream(f);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- Read the objects

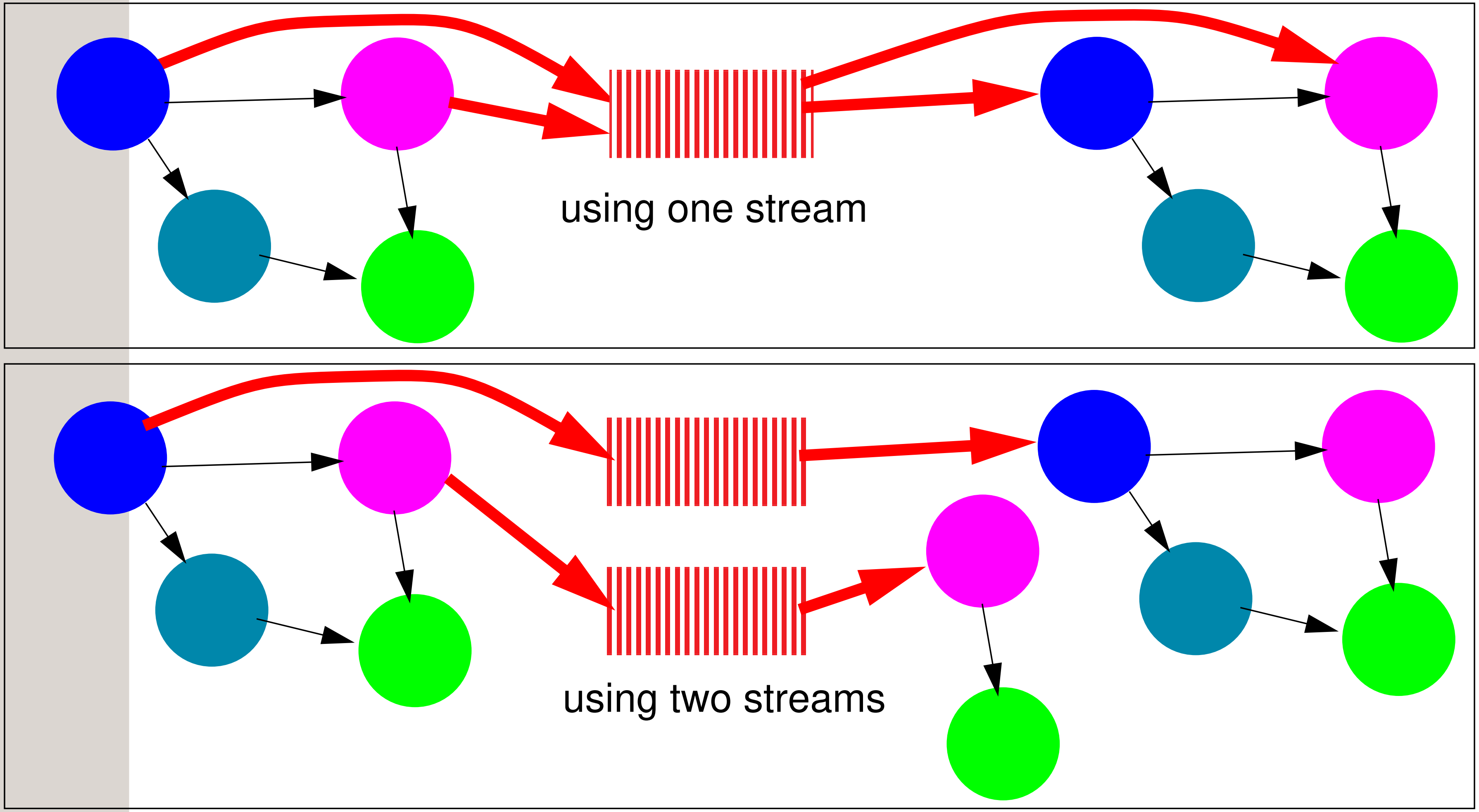
```
FileInputStream in = new FileInputStream("/tmp/objects");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

# 25.3 Interfaces

- Marker interface **`java.io.Serializable`**:
  - Instance variables are saved automatically
  - declare variable as **`transient`** to exclude it from saving
- Interface **`java.io.Externalizable`**:
  - Object can control its serialization
  - contains methods
    - **`writeExternal(ObjectOutput out)`**
    - **`readExternal(ObjectInput in)`**

# 25.4 Problems

- write all related objects to the same stream, otherwise objects are duplicated when read in



# 25.4 Problems / Hints

- to preserve the exact state: write object graph atomically
- to write the new state: clone the object or reset the object stream
- classes are not saved: they must be available if you later read in the object
- statics are not saved

# 25.5 Version Control of Classes

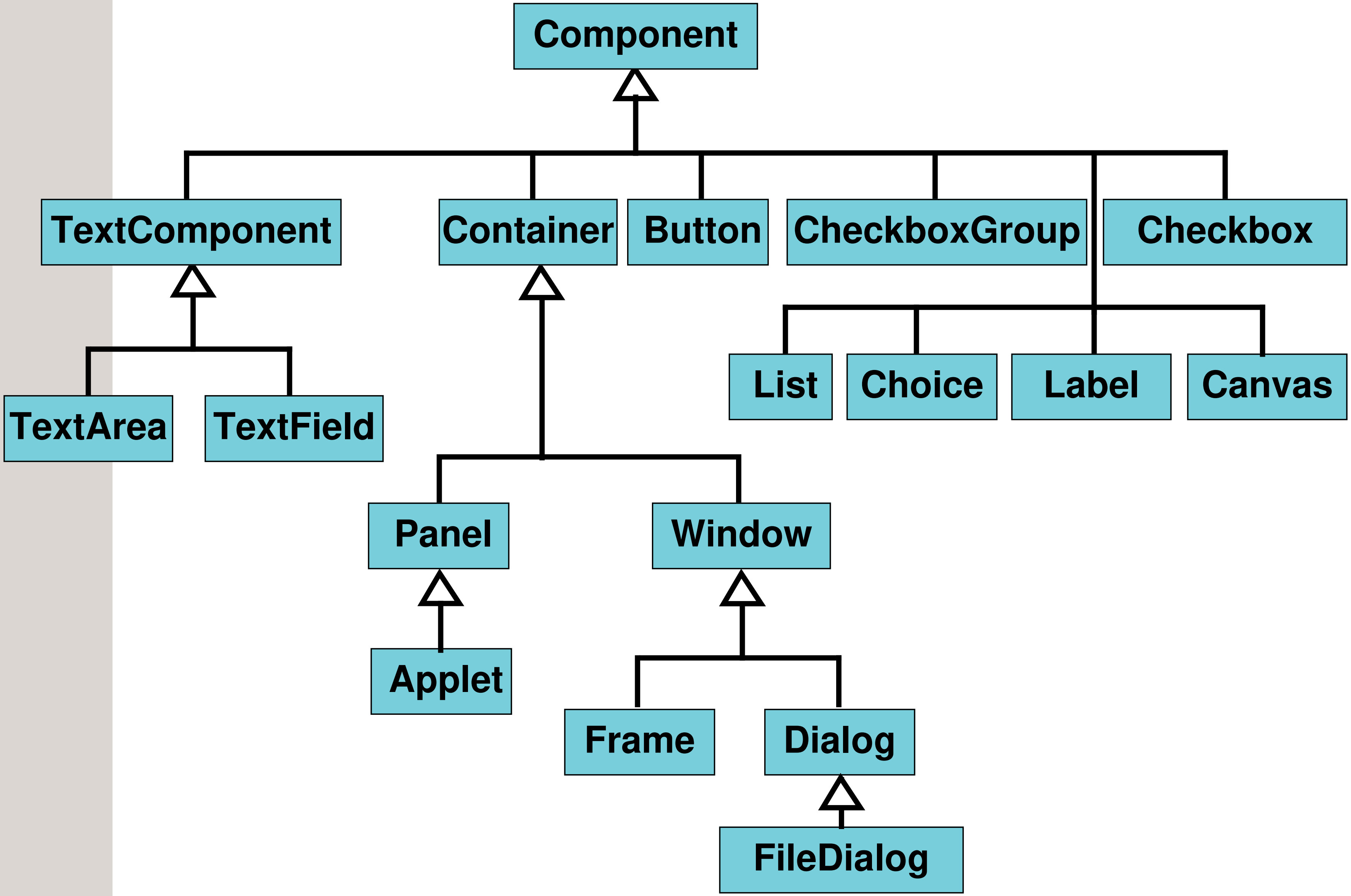
- serialized objects must be read with the "right" class
- serialized object contains a class reference that contains the class name and a version number
- version number is created by hashing over class name, interfaces, names of instance variables and methods
- Problem: small changes at the class make old serialized objects unreadable
- Solution
  - a class may contain its version number:

    ```
    static final long serialVersionUID =
    1164397251093340429L;
    ```
  - version number can be computed using **`serialver`**
  - $\Rightarrow$ change version number only after incompatible changes

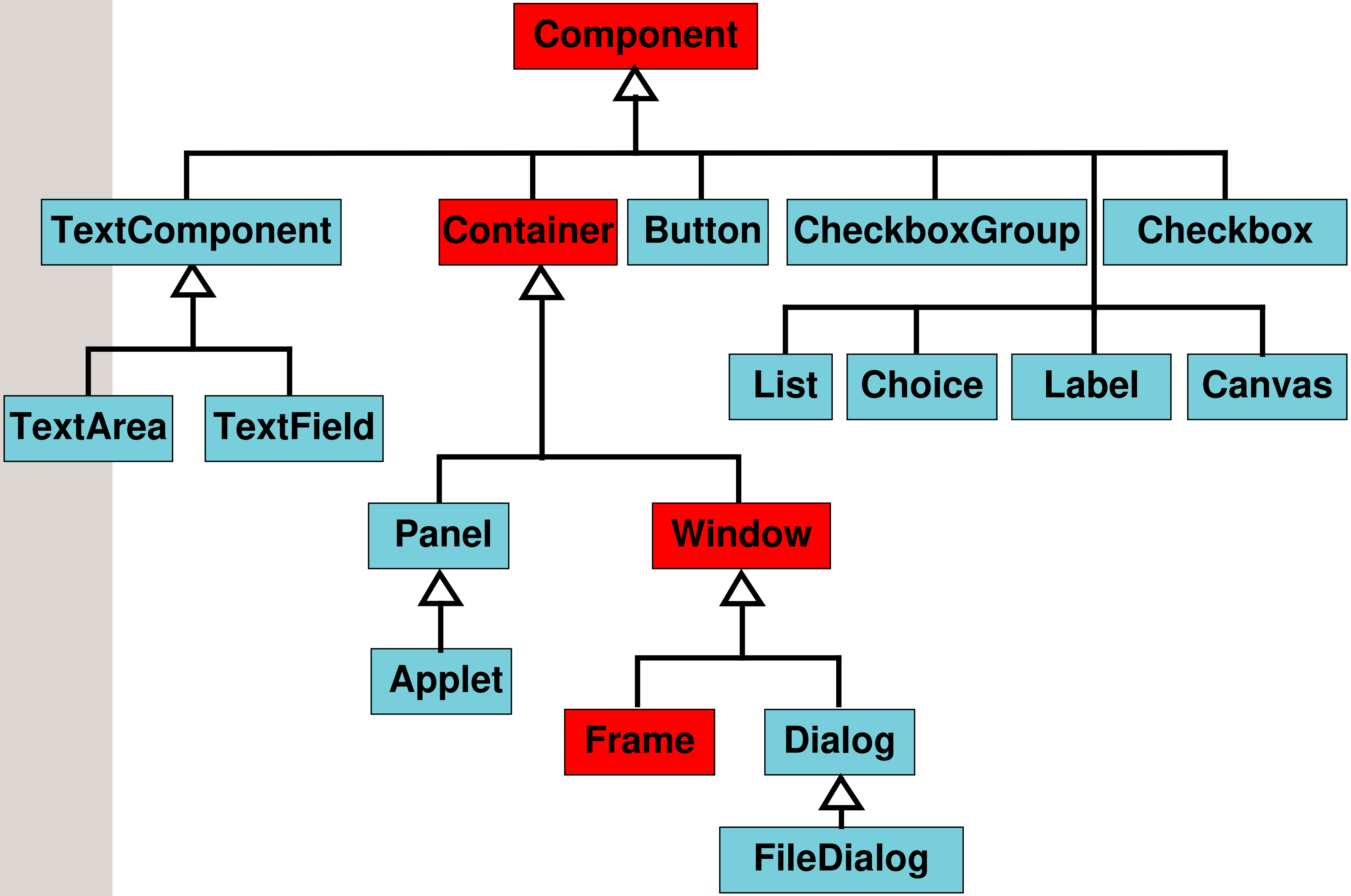
# 26 Abstract Window Toolkit (AWT)

- Contains classes to construct graphical user interfaces (GUIs)
- Classes are arranged in
  - a platform-independent package (**`java.awt.*`**)
  - a platform-dependent package (**`java.awt.peer.*`**)
- Use only classes from **`java.awt.*`** !
  - Button
  - List
  - Label
  - TextField
  - ...

# 26.1 AWT Class Diagram



# AWT Class Diagram



# 26.2 Container - Window - Frame

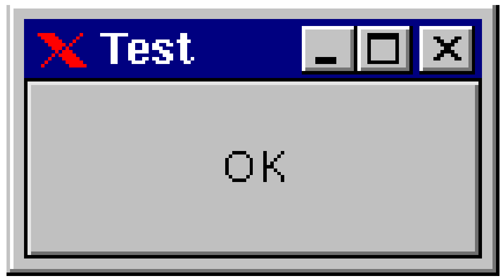
- **`Container`** is subclass of **`Component`**
  - contains **`Component`** objects
  - **`add(Component c)`**: adds a component to the container
- **`Window`** is subclass of **`Container`**
  - **`pack()`**: resize + validate
  - **`show()`**: show the window
  - **`Window`** is not used directly
- **`Frame`** is subclass of **`Window`**
  - used for top-level windows

# 26.3 Example: Frame with Button

```
import java.awt.*;

public class Simple extends Frame {
  public Simple(String title) {
    super(title);
    Button b = new Button("OK");
    add(b);
    pack();
    show();
  }

  public static void main(String a[]) {
    Simple s = new Simple("Test");
  }
}
```

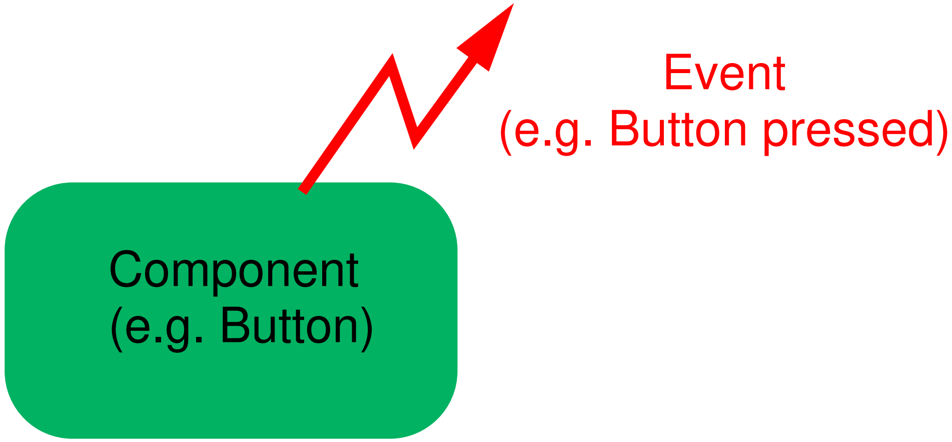
*button label*

*window title*

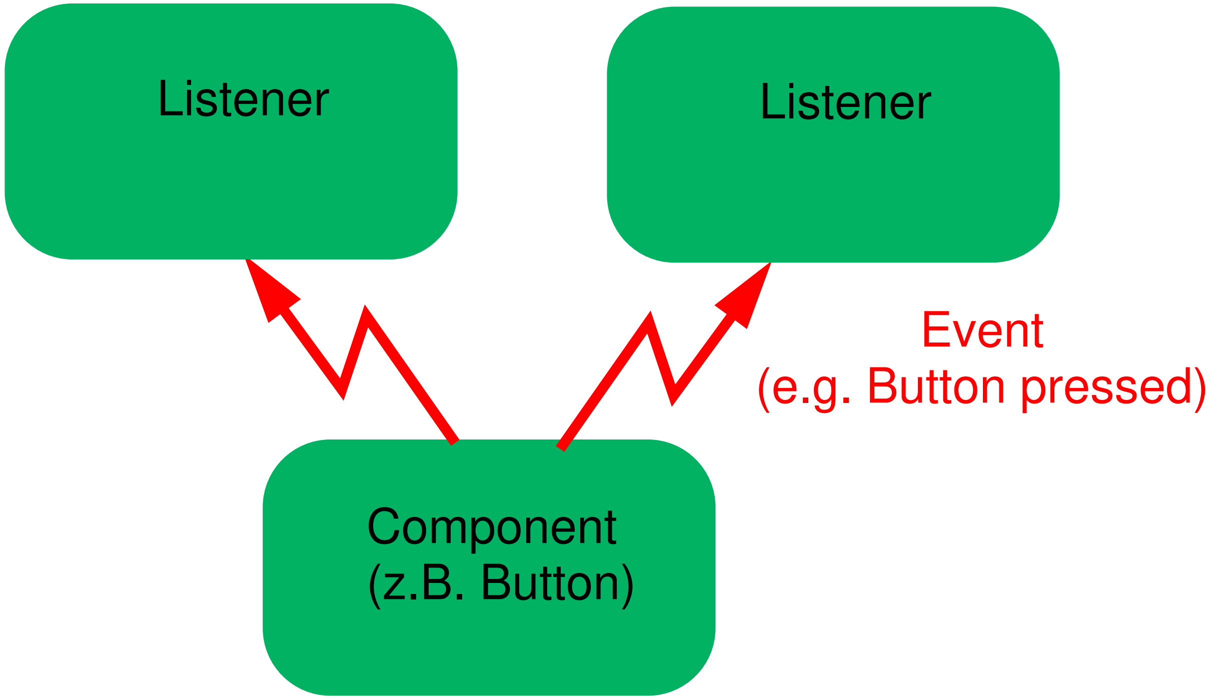
# 26.4 Events (JDK 1.1)

- User actions generate events (mouse movements, keyboard input, etc.)
- Every component can fire an event



# 26.4 Events

- Every object can be registered as an event receiver *if it implements a Listener interface*



# 26.4 Event Handling

- Events are defined in package **`java.awt.event`**
- Two kinds of events
  - High Level (push button, input text, ...)
  - Low Level (move mouse, press key, ...)

# 26.5 Events Example

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

public class SimpleEvt extends Frame {
  public SimpleEvt(String title) {
    super(title);
    Button b = new Button("OK");
    add(b);
    OKListener l = new OKListener();
    b.addActionListener(l);

    pack();
    show();
  }
  public static void main(String a[]) {
    SimpleEvt s = new SimpleEvt("Test");
  }
}
class OKListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK was pressed.");
  }
}
```

# 26.6 High-Level Events

- `ActionEvent`: User interacts with component
- `AdjustmentEvent`: User moves scrollbar slider
- `ItemEvent`: User selects component
- `TextEvent`: Contents of text component changed

# 26.7 Low-level Events

- `ComponentEvent`: component moved, resized, ...
- `FocusEvent`: component got/lost input focus
- `InputEvent`: abstract superclass of KeyEvent and MouseEvent
  - `KeyEvent`: Key was pressed, released, etc.
  - `MouseEvent`: Mouse was moved, mouse button pressed, etc.
- `ContainerEvent`: state of container changed
- `WindowEvent`: Window opened, closed, iconized, etc.

# 26.8 Events Example with Inner Class

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

public class SimpleEvt extends Frame {
  public SimpleEvt(String title) {
    super(title);
    Button b = new Button("OK");
    add(b);
    OKListener l = new OKListener();
    b.addActionListener(l);

    pack();
    show();
  }
  public static void main(String a[]) {
    SimpleEvt s = new SimpleEvt("Test");
  }

  class OKListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("OK was pressed.");
    }
  }
}
```

# 26.9 Events Example with Anonymous Inner Class

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

public class SimpleEvt extends Frame {
  public SimpleEvt(String title) {
    super(title);
    Button b = new Button("OK");
    add(b);

    ActionListener l = new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.out.println("OK was pressed.");
       }
    };

    b.addActionListener(l);

    pack();
    show();
  }

  public static void main(String a[]) {
    SimpleEvt s = new SimpleEvt("Test");
  }
}
```

# 26.10 Events Example with Anonymous Inner Class

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

public class SimpleEvt extends Frame {
  public SimpleEvt(String title) {
    super(title);
    Button b = new Button("OK");
    add(b);

    b.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
          System.out.println("OK was pressed.");
       }
    });

    pack();
    show();
  }

  public static void main(String a[]) {
    SimpleEvt s = new SimpleEvt("Test");
  }
}
```

# 26.11 Event Adpaters

- Event Adapters can be used to map an event to an arbitrary method invokation
- Useful for:
  - event queues
  - event filters
  - demultiplexing of multiple event sources to one listener
- **`java.awt.event`** package provides some simple adapters:
  - **`MouseAdapter`**, **`ComponentAdapter`**, ...
  - contain empty handlers
  - subclass these adapters and override some of the handler methods

# 26.12 Mouse Handling

- MouseListener, MouseMotionListener, MouseEvent, MouseAdapter, MouseMotionAdapter
- MouseListener is used to react to mouse buttons
- MouseMotionListener is used to react to mouse movements
- the MouseEvent object contains information about
  - the mouse position (**`getX(), getY()`**)
  - whether some modifier keys are pressed (**`isShiftDown(), isControlDown()`**, ...)

# Layoutmanager

- Problem: Arranging components in a container
- Solution #1: absolute positioning
  - Problem: User cannot change window size
- Solution #2: using a layout manager
- Principle:
  - A rectangular container contains components and other containers.
  - Every container has a layout manager.
  - The layout manager is responsible for computing the component positions.
- The most simple container is **`Panel`** with layout manager **`FlowLayout`**.

# 27.1 Layoutmanager

- FlowLayout (from left to right, line by line)
- GridLayout (aligned in a grid)
- BorderLayout (aligned at the container borders)
- CardLayout (components are cards)
- GridBagLayout (constraints for layout in the grid)

# 27.2 FlowLayout

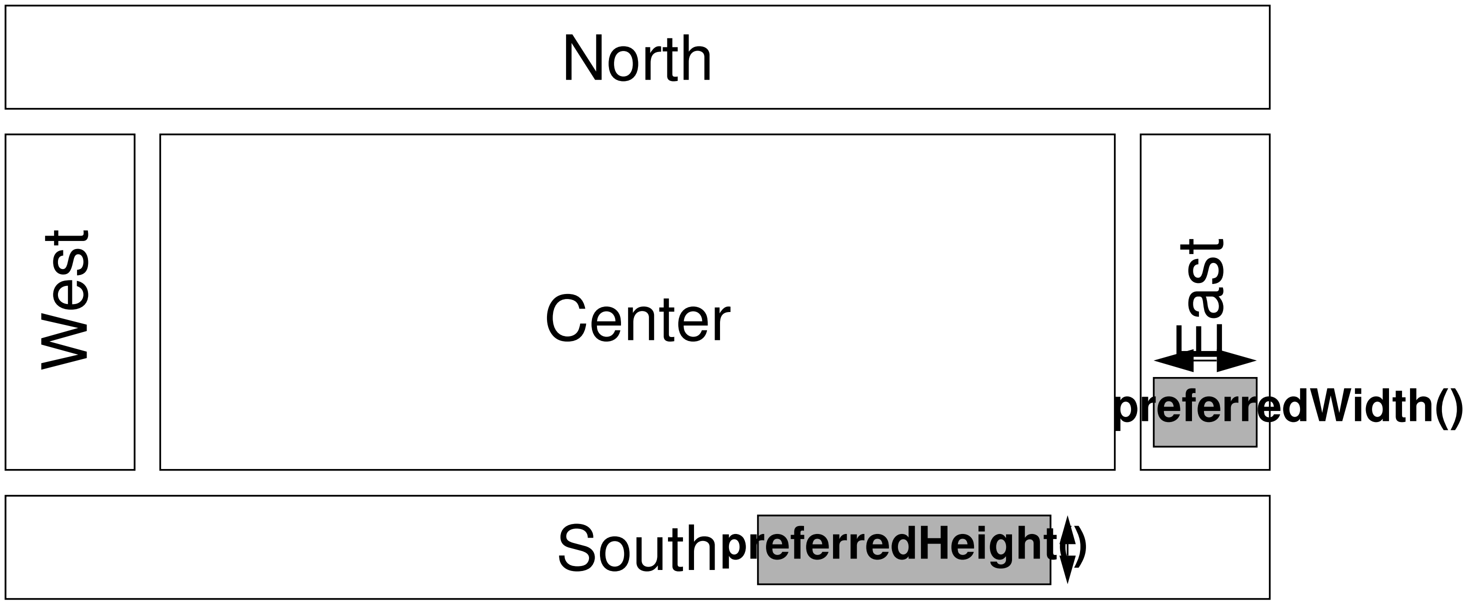
- Default LayoutManager of **`Panel`** and **`Applet`**
- Similar to a text processor with word wrap

# 27.3 FlowLayout Example

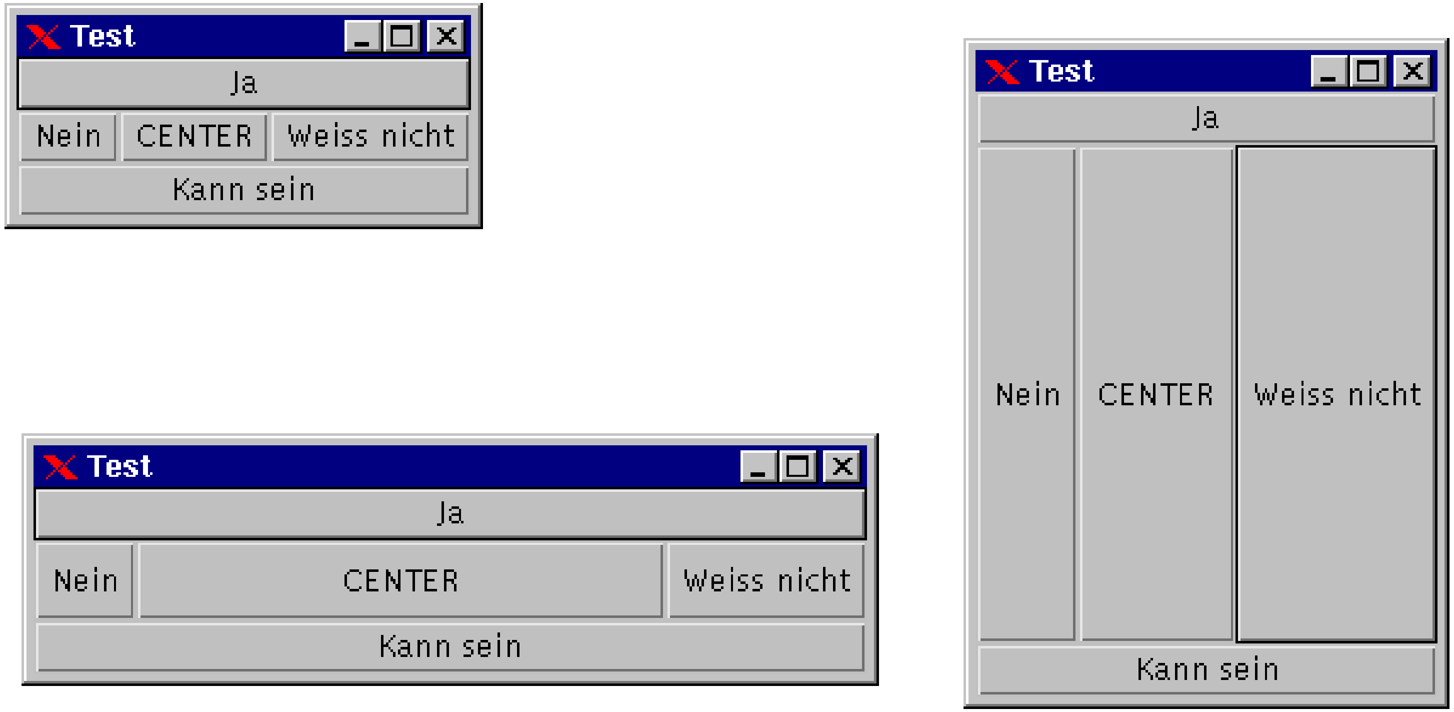
```
import java.awt.*;

public class Flow extends Frame {
  public Flow(String title) {
    super(title);

    setLayout(new FlowLayout());

    add(new Button("Ja"));
    add(new Button("Nein"));
    add(new Button("Weiss nicht"));
    add(new Button("Kann sein"));

    pack();
    show();
  }

  public static void main(String a[]) {
    new Flow("Test");
  }
}
```

# 27.4 BorderLayout

- Default LayoutManager of **`Frame`**
- 5 Components:



# 27.5 BorderLayout Example

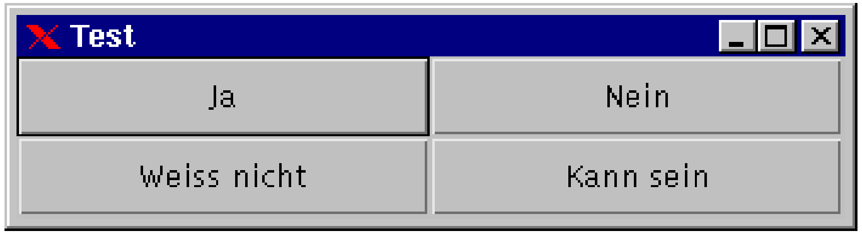


# 27.6 BorderLayout Example

```
import java.awt.*;

public class Border extends Frame {
  public Border(String title) {
    super(title);

    setLayout(new BorderLayout());

    add(new Button("Ja"), BorderLayout.NORTH);
    add(new Button("Nein"), BorderLayout.WEST);
    add(new Button("Weiss nicht"), BorderLayout.EAST);
    add(new Button("Kann sein"), BorderLayout.SOUTH);

    add(new Button("CENTER"), BorderLayout.CENTER);

    pack();
    show();
  }

  public static void main(String a[]) {
    new Border("Test");
  }
}
```

# 27.7 GridLayout

- Alignes components in a grid


# 27.8 GridLayout Example

```
import java.awt.*;

public class Grid extends Frame {
  public Grid(String title) {
    super(title);

    setLayout(new GridLayout(2,2));

    add(new Button("Ja"));
    add(new Button("Nein"));
    add(new Button("Weiss nicht"));
    add(new Button("Kann sein"));

    pack();
    show();
  }

  public static void main(String a[]) {
    new Grid("Test");
  }
}
```

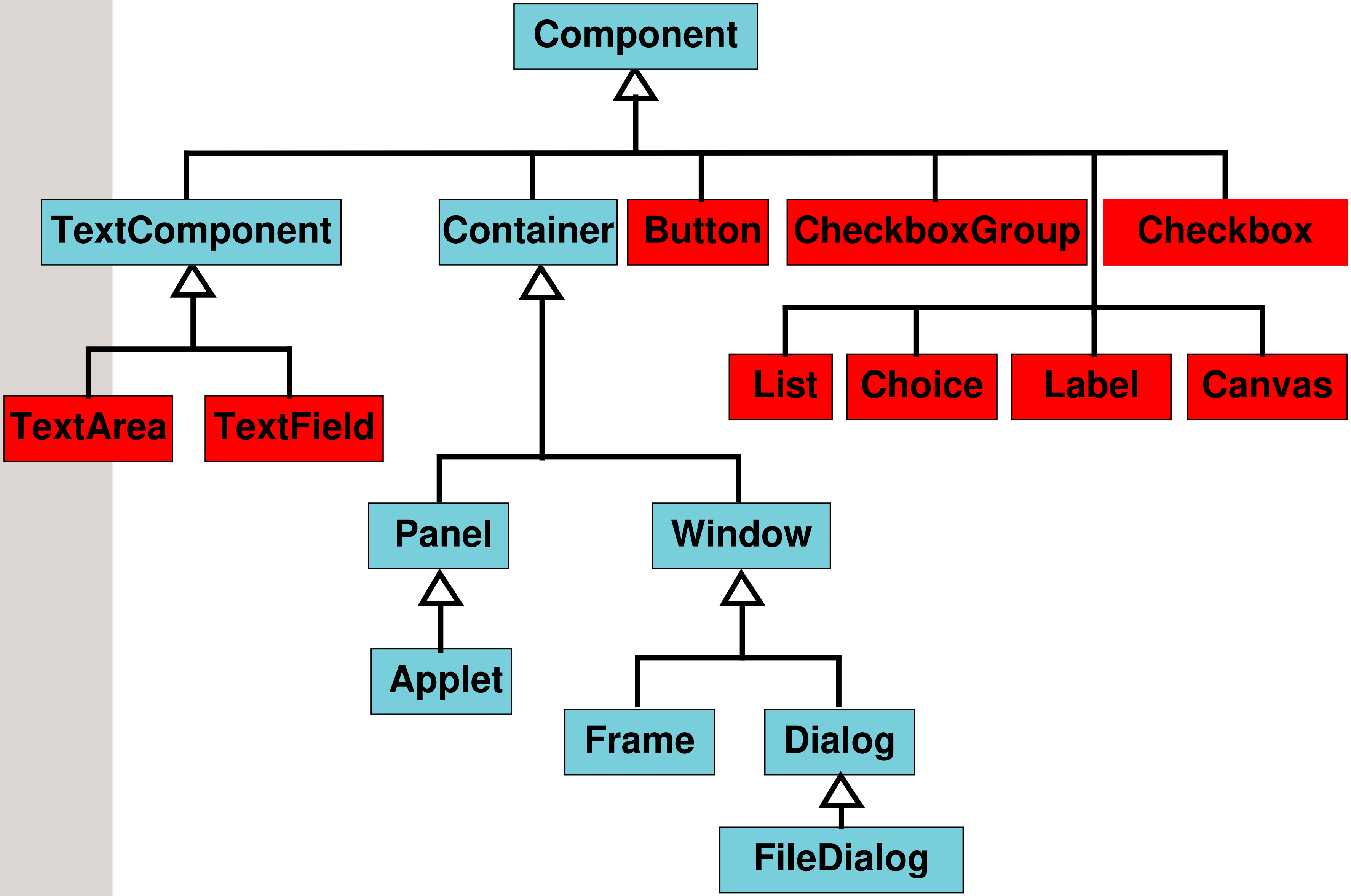
# 27.9 CardLayout

- only one component is visible at a time
- normally components are containers
- **`CardLayout`** is used to create a slide-show effect
- **`show(Container parent, String name)`** : show component with **`name`** in Container **`parent`**
- Methods **`first, next, previous, last`** iterate through the container

# 27.10 CardLayout Example

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

public class Card extends Frame {
  public Card(String title) {
    super(title);

    final CardLayout cardLayout = new CardLayout();
    setLayout(cardLayout);

    Button button = new Button("Go");
    add("Card1", button);
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        cardLayout.show(this, "Card2");
      }
    });

    ...
  }
}
```

# 28 AWT Components



# 28.1 Some important components

- **`Button`**: normally used together with ActionListener

```
Button b = new Button("OK");
b.addActionListener(new ActionListener() {
        public void actionPerformed() {...   }
};
```

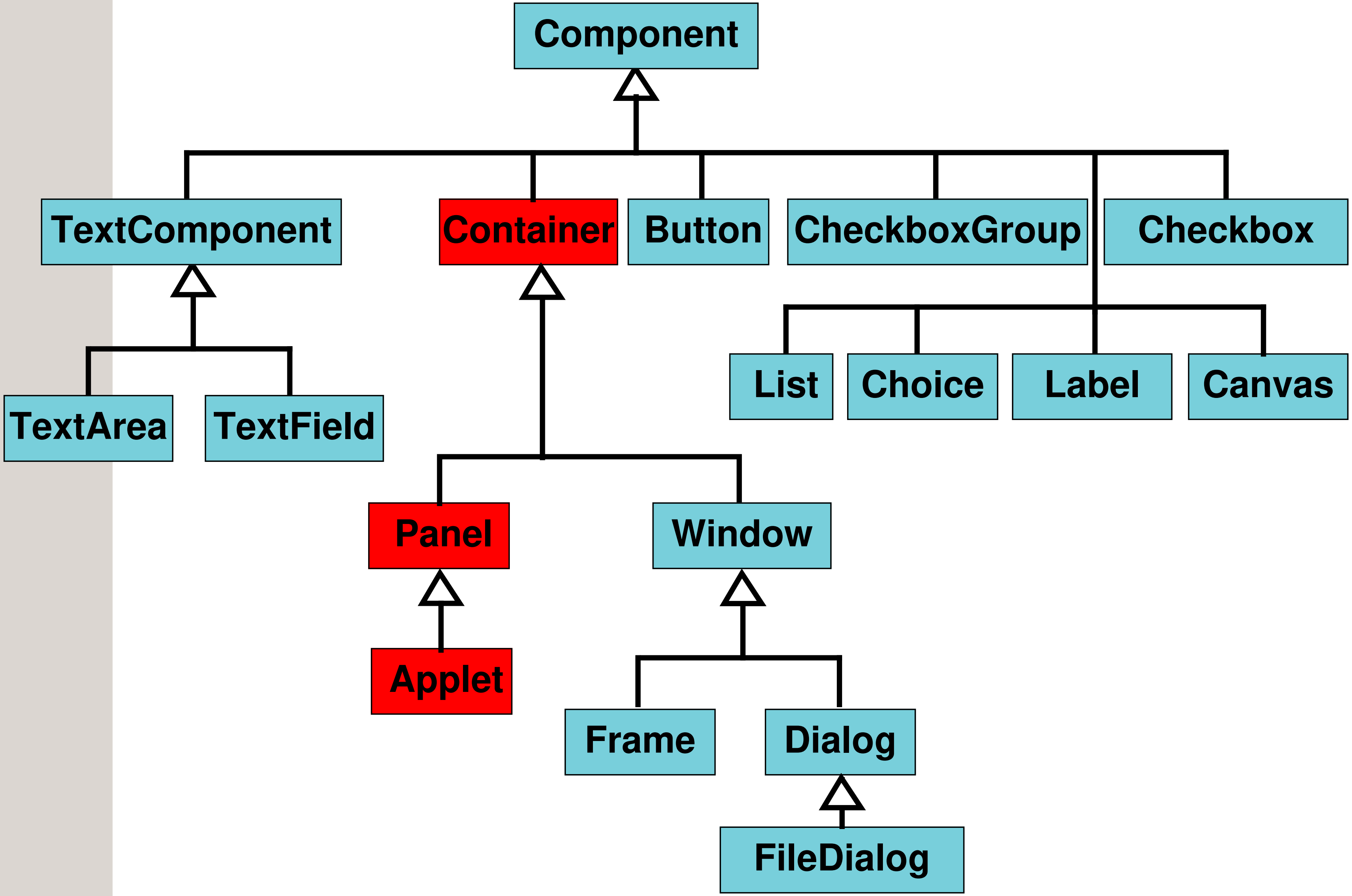
- **`TextField`**: used to get user input, normally used together with Button (as trigger)

```
TextField t = new TextField();
...
String s = t.getText();
```

- **`Canvas`**: used for drawing, normally subclassed

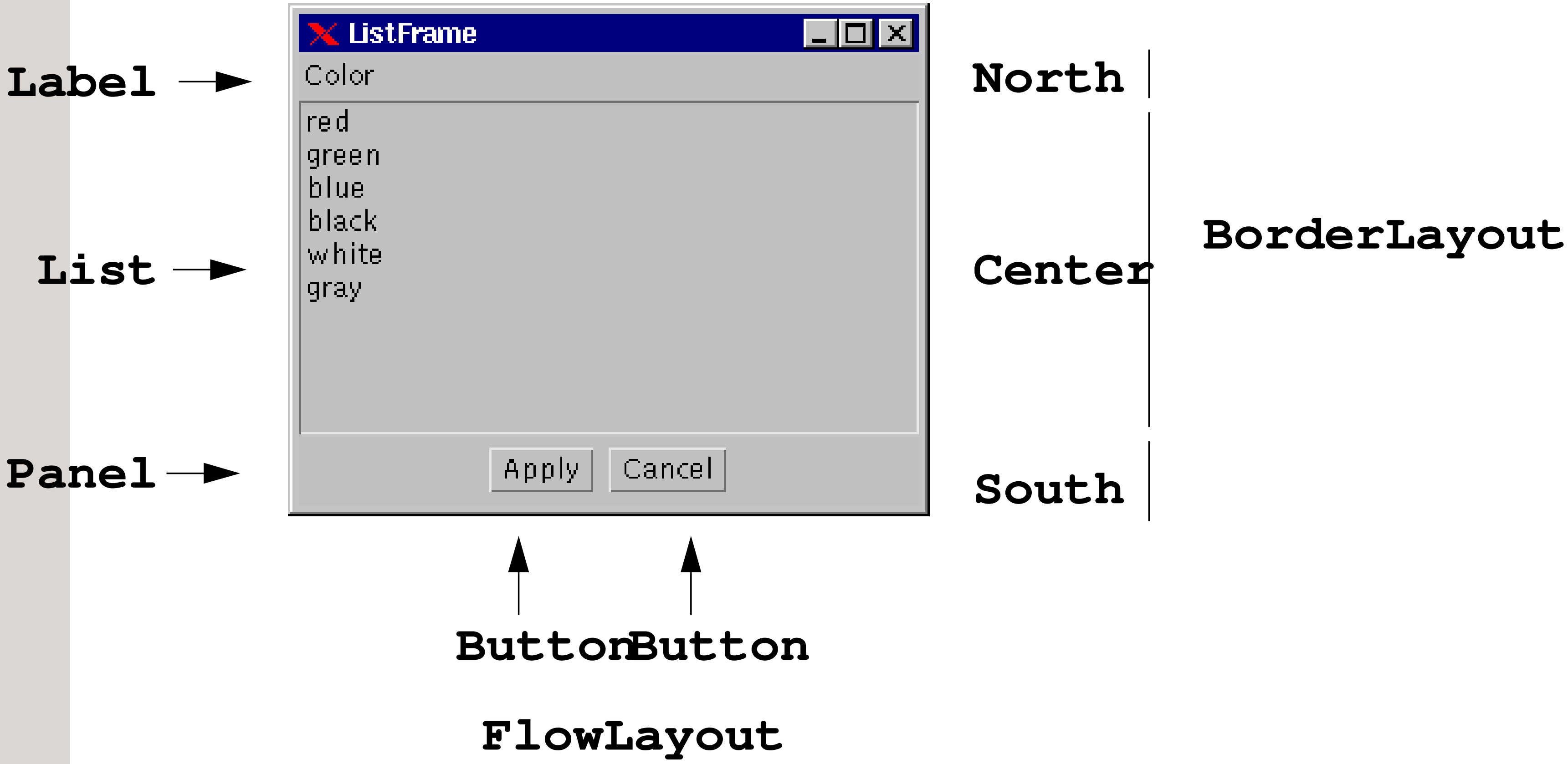
```
class MyDrawingClass extends Canvas {
        public void paint(Graphics g) { ... }
}
```

# 28.2 Using Container Components



# 28.3 Composing Panels

- Combine a panel from other panels



# 28.4 Composing Panels

```
import java.awt.*;
public class ListFrame extends Frame {
  List list;
  public ListFrame(String name) {
    super(name);                              // Frame constructor
    setLayout(new BorderLayout());            // new frame layout manager
    list = new List(3,true);                  // create a List
    add(new Label("Color"),BorderLayout.NORTH); // Label in north area
    Panel ctrlPanel=new Panel();              // create control panel
    ctrlPanel.setLayout(new FlowLayout());    // new Panel layout manager
    ctrlPanel.add(new Button("Apply"));       // add Apply and Cancel
    ctrlPanel.add(new Button("Cancel"));      // buttons to control panel
    add(ctrlPanel, BorderLayout.SOUTH);       // control panel at south
    add(list, BorderLayout.CENTER);           // List at center
    pack();                                   // compute positions
    show();                                   // show frame
  }
  public static void main(String a[]) {
    ListFrame frame = new ListFrame("ListFrame");
    frame.list.addItem("red");
    ...
  }
}
```

# 29 AWT Graphics

- Drawing
- Images

# 29.1 Drawing

- `java.awt.Graphics` has methods to draw
- is passed to the `paint` method

```
import java.awt.*;

public class DrawTest extends java.applet.Applet {
  public void paint(Graphics g) {
    g.setColor(Color.red);
    g.drawOval(10,10,70,100);
  }
}
```

# Images

- a **`Image`** object contains a GIF, JPEG, etc.
- you can draw images using the **`drawImage`** method of **`Graphics`**

```
import java.awt.*;

public class ImageTest extends java.applet.Applet {
  Image image;
  public void init() {
    image = getImage(getDocumentBase(), "image.gif");
  }
  public void paint(Graphics g) {
    g.drawImage(image,10,10,this);
  }
}
```

# 29.3 Images

- Load and show images
- Consider World-Wide-Web problems
  - slow connections
  - load asynchronously: All images are loaded in parallel.
- Principle:
  - **`Image`** objects contain a partially/completely loaded image
  - **`Applet.getImage()`** returns immediately, image starts loading if **`drawImage`** is called
  - **`MediaTracker`** object controls image loading

# 29.4 MediaTracker

**`void addImage(Image image, int id)`**

Control **`image`** loading. **`id`** can be used to group images.

**`void waitForID(int id)`**

Wait for medium with **`id`**.

**`boolean checkAll()`**

Returns **`true`** if all images/media are loaded.

**`boolean checkAll(boolean force)`**

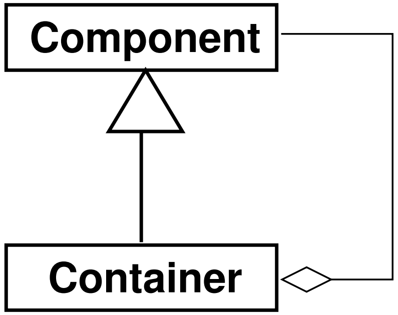
Returns true if all images are loaded. If **`force=true`** stat image loading.

# AWT Design Patterns

- AWT employs many design patterns
- understand the AWT looking at these patterns
- Patterns:
  - *Composite*: Component management in a Container
  - *Strategy*: Separating layout strategy from Container objects
  - *Bridge*: Separating platform independent from platform dependent AWT objects
  - *AbstractFactory*: Creating peer objects
  - *Singleton*: the abstract factory that creates peer objects
  - *Observer*: the listener mechanism

# 30.1 Composite Pattern

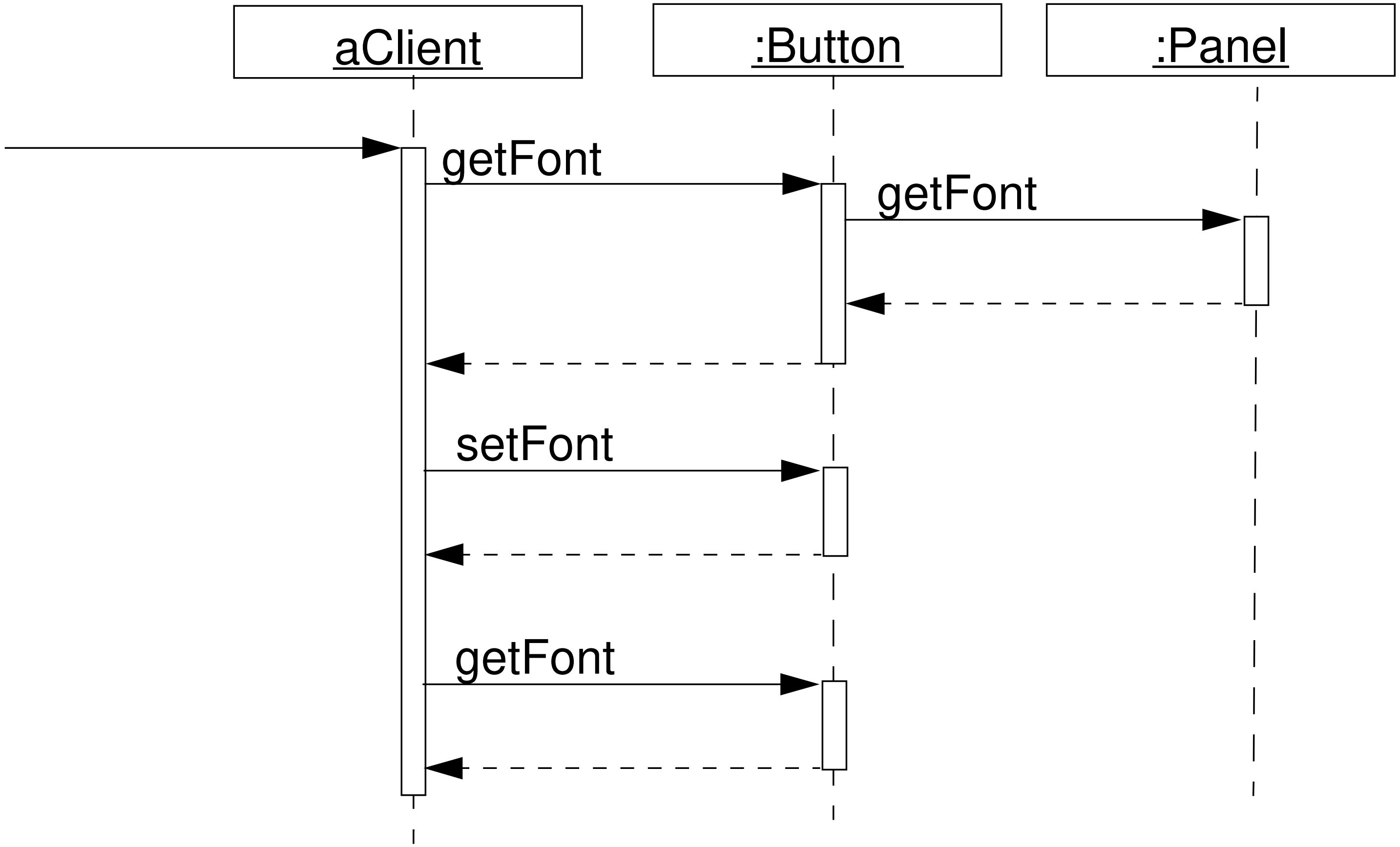
- Class diagram




- Roles
  - Component: supplies GUI service
  - Container: is composite, manages components(children), computes position

# 30.1.1 Interactionen during getFont

- Which font is used for a newly created button?

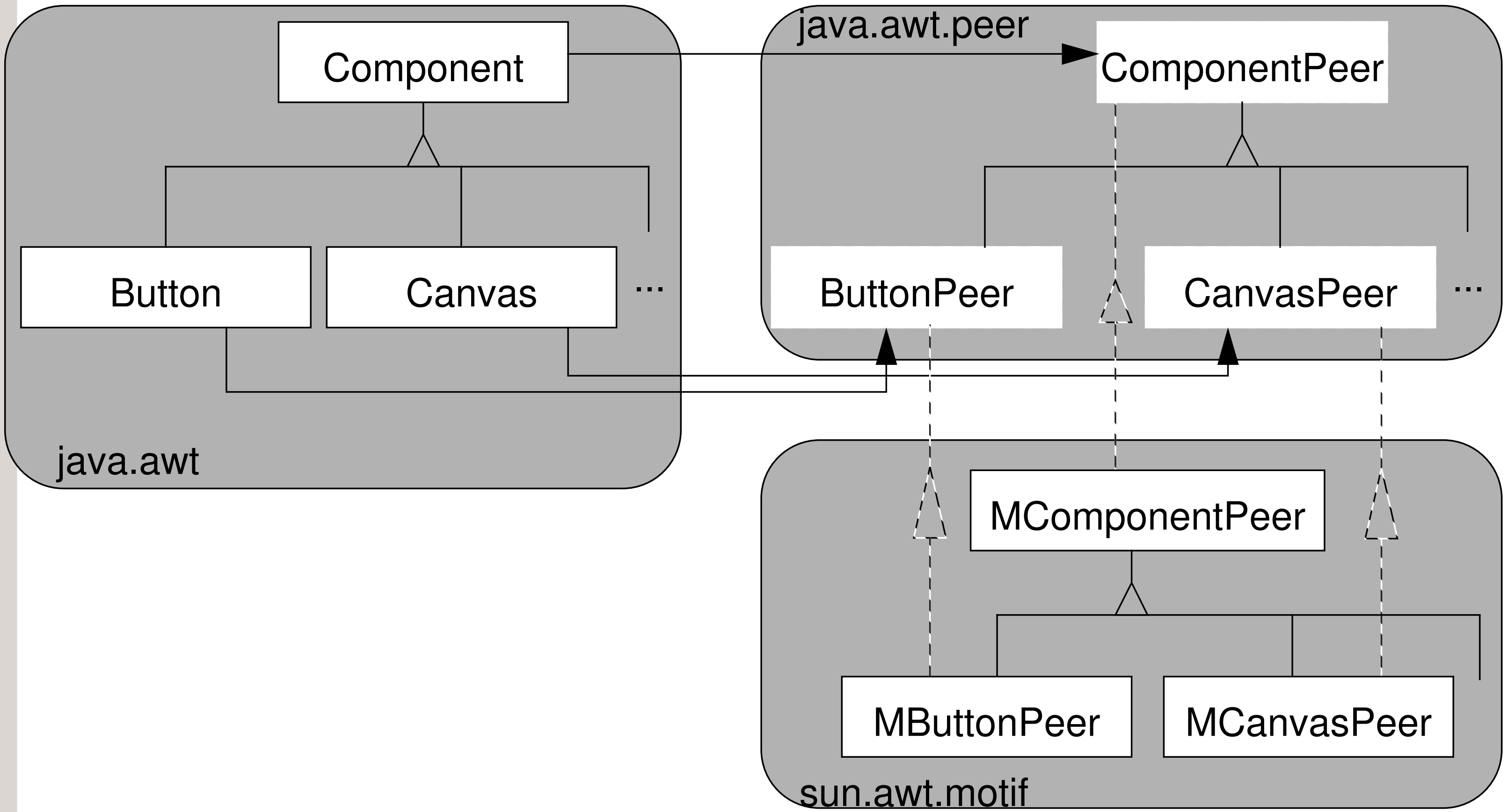



- Propagation of the font property from the parents to the children

# 30.2 Bridge Pattern

- Motivation
  - platform-independent user interface
- Roles:
  - *Component*: platform-independent visual component
  - *ComponentPeer*: platform specific component

## 30.2.1 Peers - Class Diagram



# 30.3 Factory

- Problem: Instantiation ties to a specific GUI

```
ButtonPeer b = new MButtonPeer();
ScrollbarPeer s = new MScrollbarPeer();
```

# 30.3 Factory

- Problem: Instantiation ties to a specific GUI

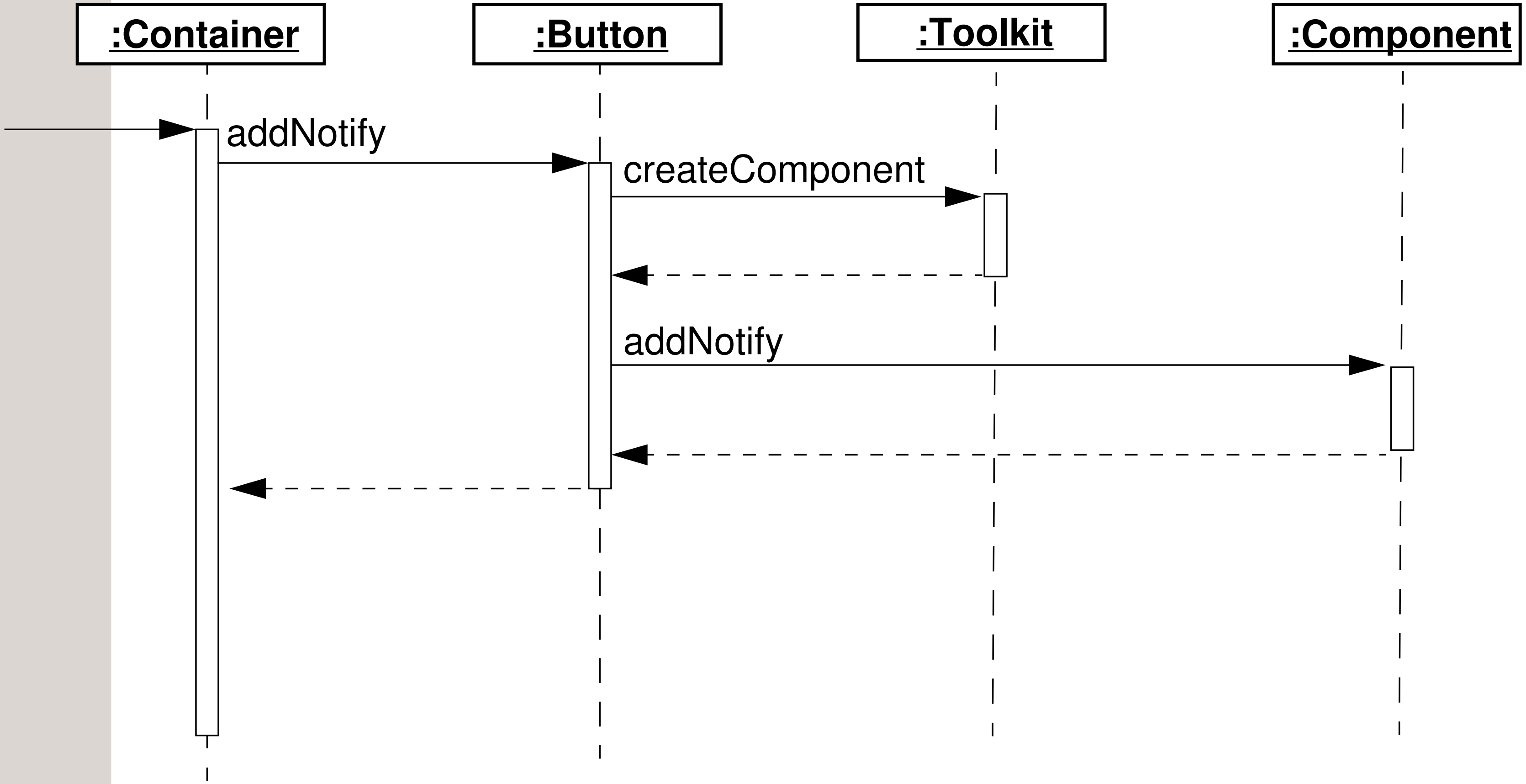
```
ButtonPeer b = new MButtonPeer();
ScrollbarPeer s = new MScrollbarPeer();
```

- Solution: Use a factory (Toolkit)

```
Button button = new Button();
ButtonPeer buttonPeer = toolkit.createButton(button);
```

# 30.3 Factory

- Sequence diagram: making a button visible



# 30.4 Singleton

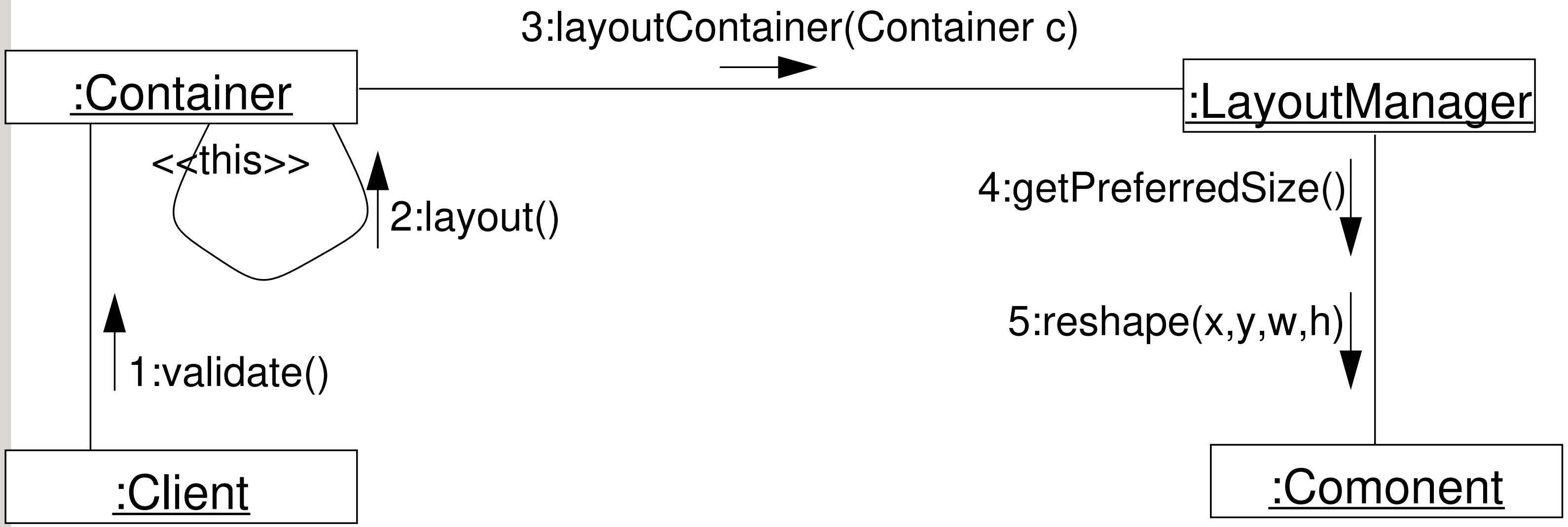
- Motivation
  - only one instance of Toolkit should be allowed
  - the class name is given as a property

```
abstract class Toolkit {
    Toolkit instance;

    private Toolkit() {}

    public Toolkit getDefaultToolkit() {
        if (instance == null) {
            Class tClass =Class.forName(toolkitName);
            instance = tClass.newInstance();
        }
        return instance;
    }
    ...
}
```

# 30.5 Strategy

- Separation of container and layout strategy
- Roles:
  - **`Container`**: wants to layout a component
  - **`LayoutManager`**: provides layout algorithm
  - **`Component`**: provides size information and is positioned

# 30.5.1 LayoutManager - Collaboration Diagram



# 30.6 Observer

- Example
  - Listener mechanism
  - **`Observer`**/**`Observable`** from **`java.util`**
- Observ *Subject* and react