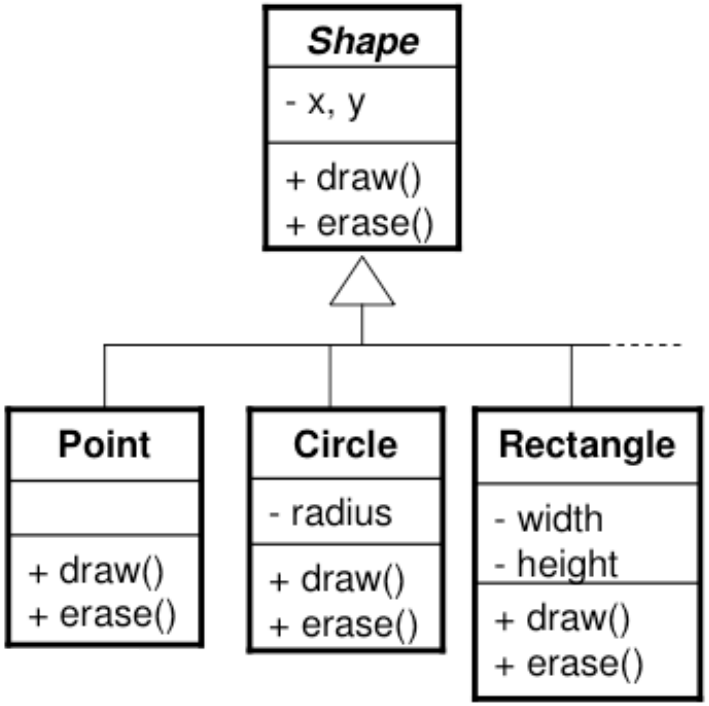
Abstract classes
vs.
Interfaces

Abstract class

- Sometimes we don't want objects of a base class to be created
- Examples:
  - *Animal:* Cat, Cow, Dog,…
    - An Animal object makes no sense
  - *Shape:* Point, Rectangle, Triangle, Circle
    - What does a generic Shape look like?
- Solution: make it an abstract class
- Cannot be instantiated



Abstract class

- gathers all the common properties of its derived classes
- may contain data members
- may contain fully implemented methods
- may contain abstract methods

Abstract class

```
abstract class Shape {
   protected int x, y;
   Shape(int _x, int _y) {
      x = _x;
      y = _y;
   }
}

class Circle extends Shape {
   private int r;
   public Circle(int _x, int _y, int _r) {
      super(_x, _y);
      r = _r;
   }
   ...
}
```

```
Shape s1 = new Circle();
Shape s = new Shape(10, 10) //  compile error
```

Abstract class: objects cannot be instantiated

Concrete class: objects can be instantiated

Abstract methods

- Sometimes we want a method in base class to serve as the common interface of subclasses' versions only:
    - i.e. it should never be called

      `Animal.makeASound()`. It contains only dummy code
- Solution: make it an abstract method
- An abstract method has only a declaration and no method body (i.e. no definition):

```
abstract void f();
```

- The class containing an abstract method MUST be qualified as abstract
- An abstract method must be overridden and defined in a derived class so that objects of that class can be created (concrete class)

Abstract method

```java
abstract class Shape {
   protected int x, y;
   Shape(int _x, int _y) {
      x = _x;
      y = _y;
   }
   abstract public void draw();
   abstract public void erase();
   public void moveTo(int _x, int _y) {
      erase();
      x = _x;
      y = _y;
      draw();
   }
}
```

```java
class Circle extends Shape {
   private int r;
   public Circle(int _x, int _y, int _r) {
      super(_x, _y);
      r = _r;
      draw();
   }
   public void draw() {
      System.out.println("Draw circle at ("+x+","+y")");
   }
   public void erase() {
      System.out.println("Erase circle at ("+x+","+y")");
   }
}
```

Interfaces

- Java does not support multiple inheritance
  - What if we want an object to be multiple things?
- Specify the form (specification, behavior) of something (a concept)
  - NOT providing an implementation
  - interface's methods define the contract / protocol (signature) to be respected for this concept
- Based on the "has-a" relationship
- Useful when you want to use code written by others

Interface

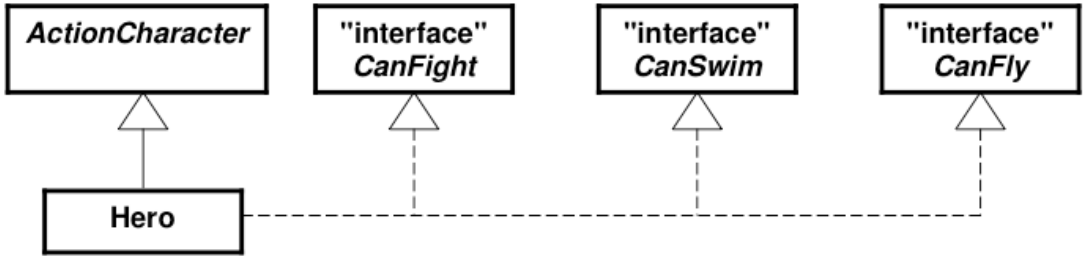
- A special type of class - a “pure” abstract class:
  - No data (only *static* or *final*)
  - Defines a set of abstract methods (prototypes)
    - does not provide the implementation for the prototypes, only the definition (signature)
- A class can implement *any number* of interfaces
  - it will respect a “contract”: implement *all* its methods
  - NB: implementation <> inheritance

Interface example

```
public interface Comparable<T>{
   public int compareTo(T o);
}
public class Name implements Comparable<Name>{
    ...
    public int compareTo(Name n){
        ... //implementation of compareTo
    }
}
```

Implementing interfaces

- Implementations provide complete methods:




```
interface CanFight {
    void fight();
}
interface CanSwim {
    void swim();
}
interface CanFly {
    void fly();
}
class ActionCharacter {
    public void fight() {...}
}
```

```
class Hero extends ActionCharacter
implements CanFight, CanSwim, CanFly {
    public void swim() {...}
    public void fly() {...}
}
```

Interface vs. Abstract class

- specify the form of a concept: not implementing it
- cannot have data members, only constants
- lightweight to implement
- multiple-implementations
- based on "has-a" (composition)

- incomplete class (may have partial implementations) which needs a specialization (derivation)
- can have data members
- base class: used to initialize a hierarchy of classes
- single-inheritance
- based on "is-a" (inheritance)

Java Swing

Evolution of Java GUI

- Java 1: AWT (`java.awt.*`) built in 30 days, and it shows
- Java 2: Swing (`javax.swing.*`) extends/replaces AWT; very different, vastly improved
- This lecture covers Swing only
- Terminology:
  - Control: generic term for the manipulable elements of the screen: button, scrollbar, text, menu
  - Container: window containing controls or other containers
  - Component: collective name for Controls and Containers

Swing vs. AWT

- AWT: *heavyweight*
  - primitive components
  - window handling, buttons, etc.
- Swing: *lightweight*
  - no native code, platform independent
  - pluggable look and feel
  - Java 2D API
  - drag and drop
  - its components (`JScrollBar, JButton, JTextField, ...`) replace the AWT ones, which are simpler but more rudimentary
  - its components are drawn in an AWT canvas
  - event handling done by AWT
  - all components' name start with `J*`

Basic principles

- Graphical components
  - ex: frames, buttons, drawings, etc.

- Events and actions
  - ex: press a button, click the mouse, etc.

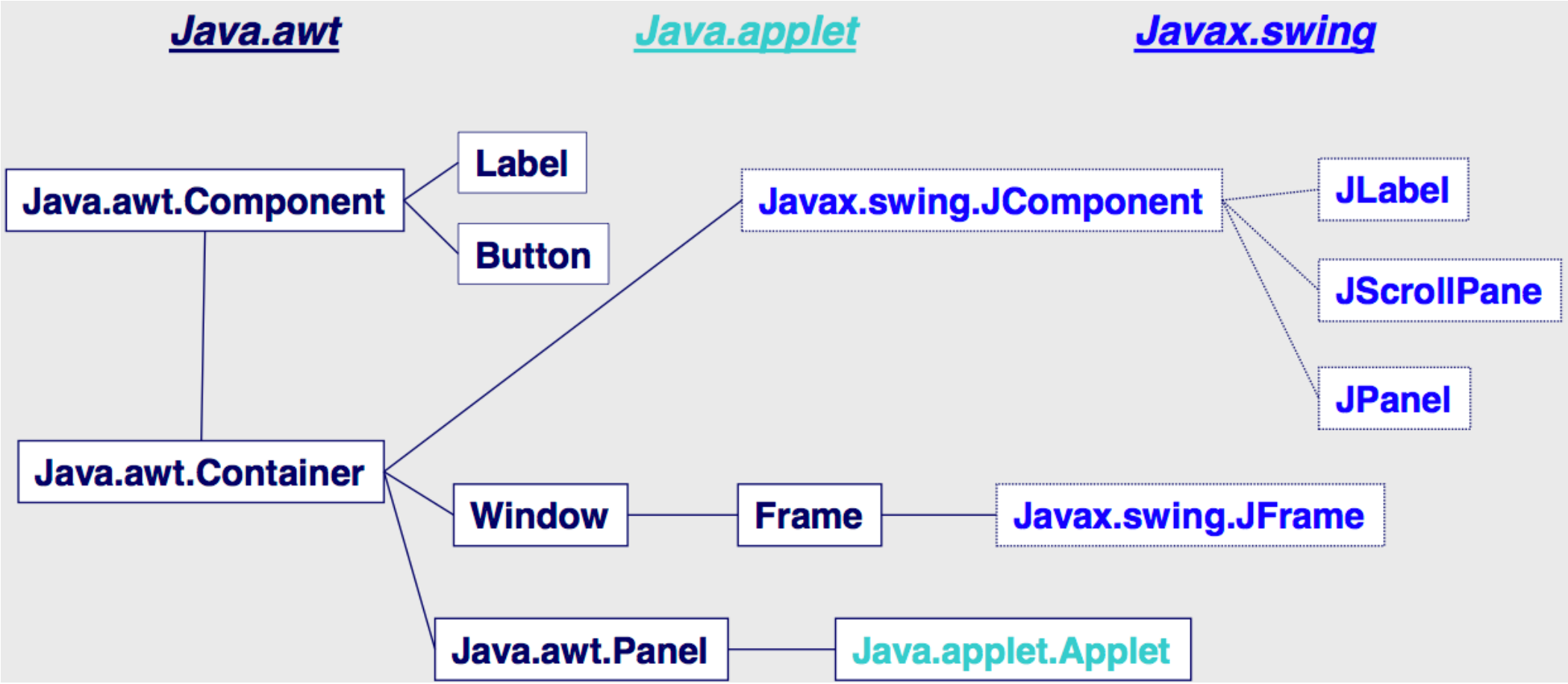
Basic principles

- Define the graphical components
  - instance of the API classes
- Place them (layout management) in a container
- Define actions associated to events (listeners) and associate them to the graphical components: *event-driven*

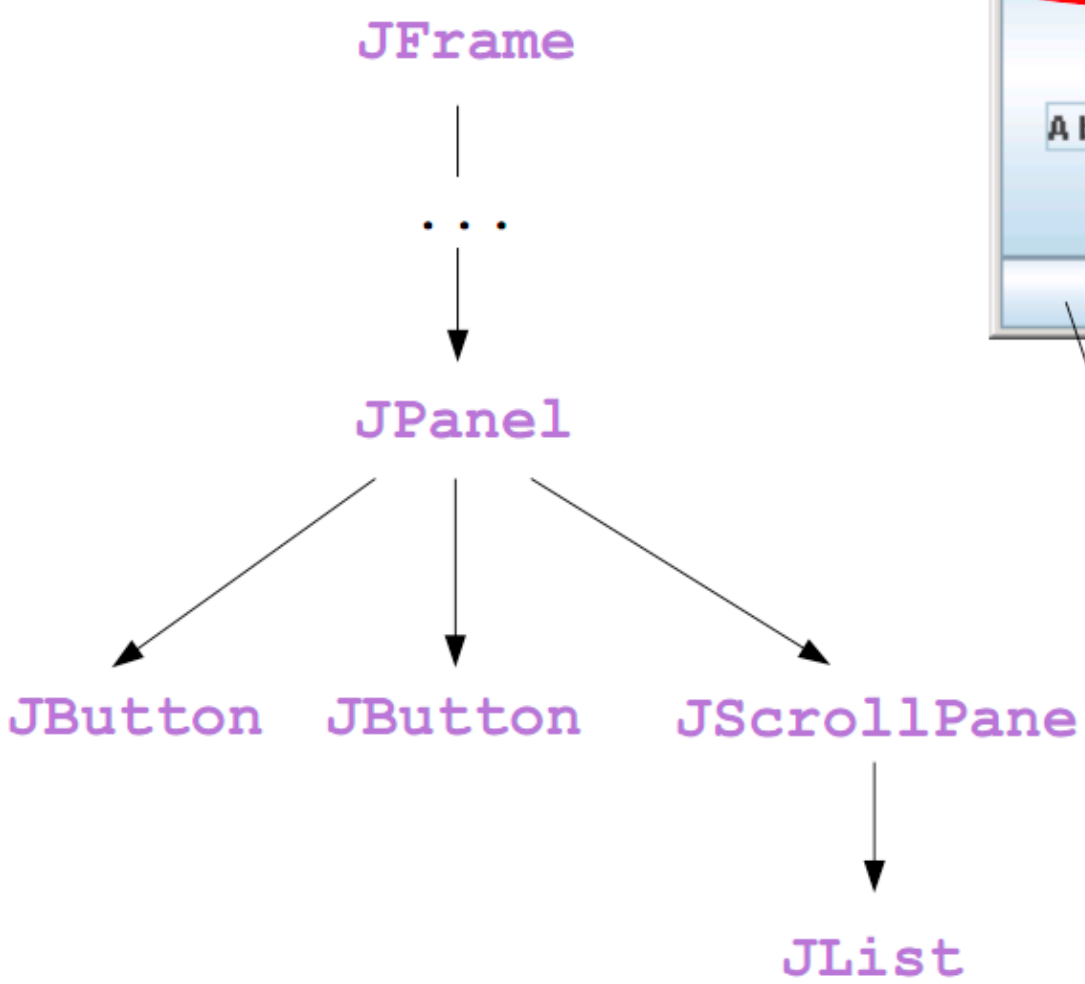
Swing hierarchy

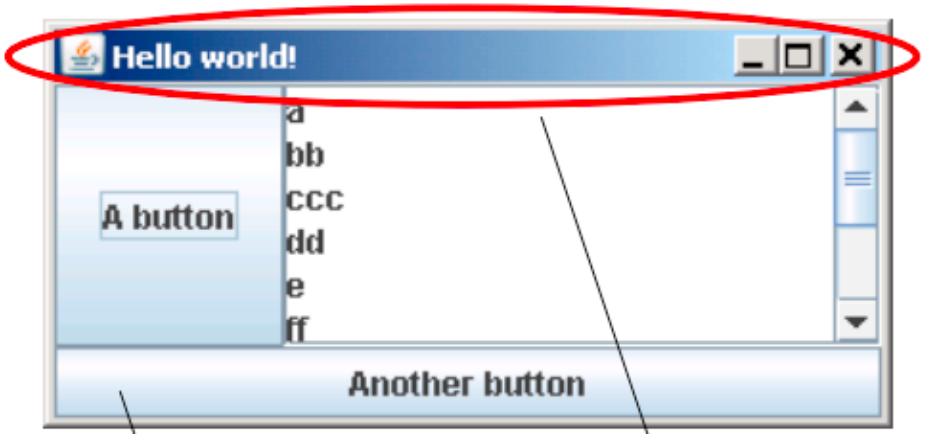
- A Swing interface is a tree having as root a system object (*heavyweight*) - container:
  - `JFrame`: normal system window
  - `JWindow`: non decorated system window
  - `JDialog`: dialog box
  - `JApplet`: an applet display area within a browser window
- The root contains objects handled by Java (*lightweight*)

(Part of) the Swing components hierarchy



An example





system style, as `JFrame` is heavyweight

Swing style, as `JButton` is lightweight

JFrame

- Setting up a frame:

```
JFrame frame = new
JFrame("HelloWorldSwing");

frame.setDefaultCloseOperation(JFrame.EX
IT_ON_CLOSE);
```

- Adding a component to a frame:

```
frame.add(component);
```

- Lay the components according to some rules and make them visible:

```
frame.pack();

frame.setVisible(true);
```

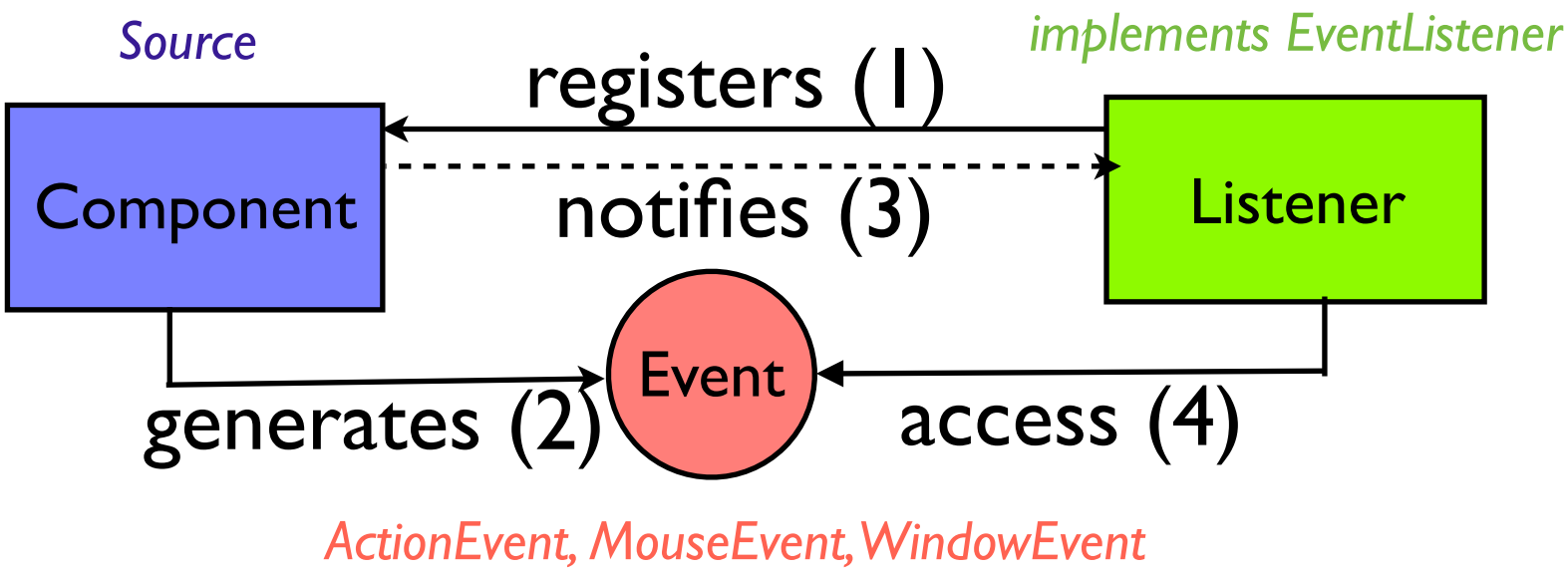
Layout management

- Use different layout managers to control the size and position of the components
- Setting the layout manager:

```
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());
```

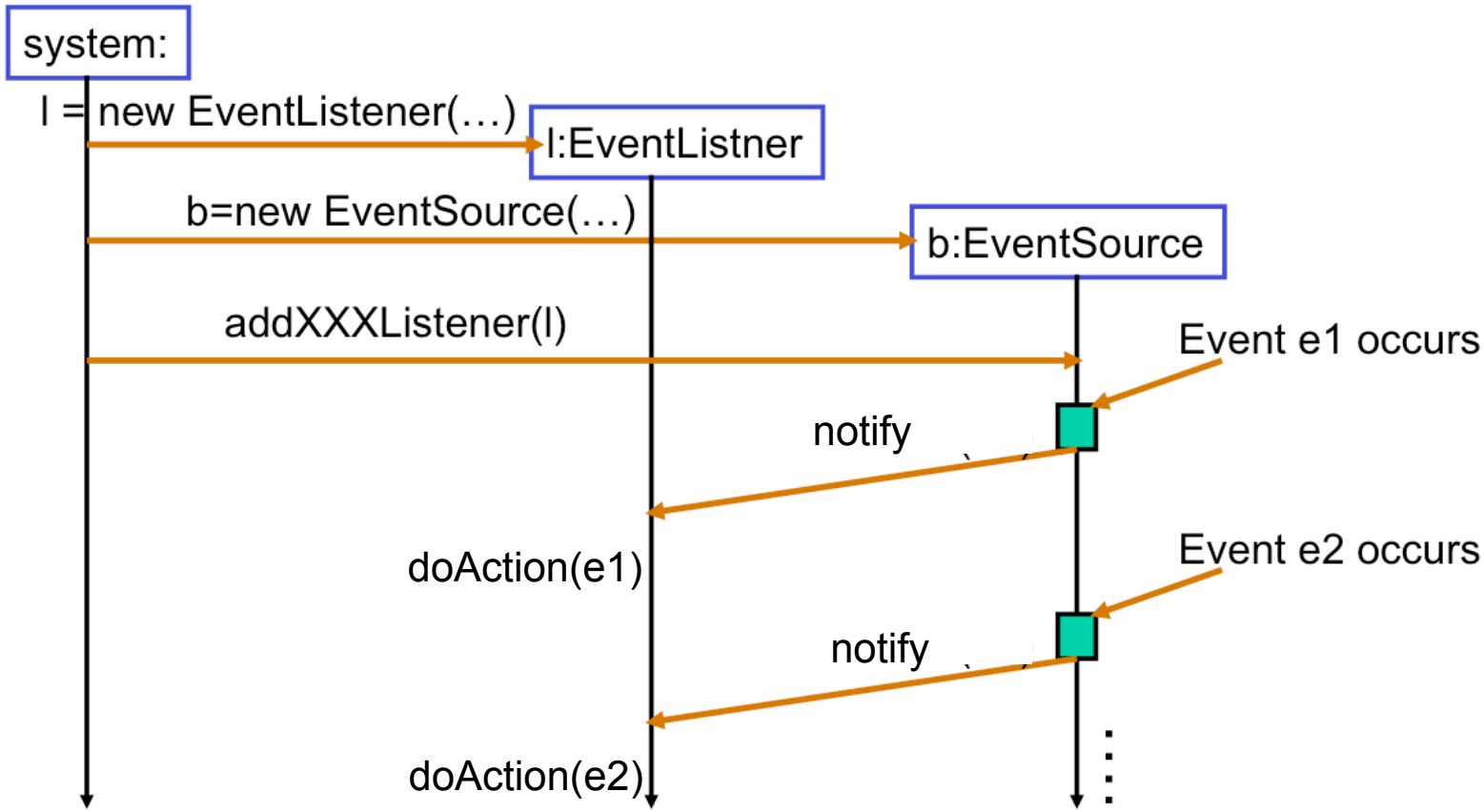
- Providing hints about a component:
  - provide size hints: `setMinimumSize(Dimension), setPreferredSize(Dimension)`
  - provide alignment hints: `setAlignmentX(float), setAlignmentY(float)`
- Putting space between components:
  - the layout manager: can specify `hgap` and `vgap`.
  - putting invisible components
  - empty border: best for components that have no default border (`JPanel`, `JLabel`)

Event notification



Java event model

- delegation (or forwarding) model



Event handling

- Every time the user types a character (`KeyEvent`) or pushes a mouse button (`MouseEvent`) etc., an event occurs:
  - stores info about what button, mouse position, etc.
  - added to an *event queue*
  - events in the event queue are sent to the objects registered to listen for them
  - **`<type>Event:`** `ComponentEvent`, `KeyEvent`, `MouseEvent`, `WindowEvent`, etc.
- Any object can be *notified* of the event:
  - registered as an *event listener* on the appropriate event source.
  - implements the appropriate interface (ex: `ActionListener`) that defines the methods associated to that event
  - **`<type>Listener:`** `ComponentListener`, `KeyListener`, `MouseListener`, `WindowListener`, etc.
- Possible to register multiple listeners to a single component

Events and listeners

Act that results in the event	Listener
User clicks a button, presses Return while typing in a text field, or chooses a menu item	`ActionListener`
User closes a frame (main window)	`WindowListener`
User presses a mouse button while the cursor is over a component	`MouseListener`
User moves the mouse over a component	`MouseMotionListener`
Component becomes visible	`ComponentListener`
Component gets the keyboard focus	`FocusListener`
Table or list selection changes	`ListSelectionListener`

How to implement an event handler

- Implement and instantiate an event listener:

```
public class MyClass implements XXXListener {…}
MyClass l = new MyClass(…);
```

- Register the event listener as an listener on event source:

```
EventSource.addXXXListener(l) ;
```

- From now on, every time an event `e` occurs, the event source object will trigger the appropriate `doXXXAction(e)` from `l`.
- Event-handling code executes in a single thread, the event-dispatching thread.
- Event handlers should *execute very quickly*. Otherwise, the program's perceived performance will be poor. If needing lengthy operation, starting up another thread

Listener vs. Adapter

- Example: the `MouseListener` interface

```
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e)
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseClicked(MouseEvent e);
```

- Any class that implements `MouseListener` has to define all its methods, although it may not use all of them
- Solution: `java.awt.event.MouseAdapter` implements `MouseListener` by defining an empty block for each method
  - any class that extends `MouseAdapter` can redefine only the useful methods
- Similar pattern for: `WindowListener` / `WindowAdapter`, `MouseMotionListener` / `MouseMotionAdapter`, `KeyListener` / `KeyAdapter`

Java 2D API

- A uniform rendering model for display devices and printers
- A wide range of geometric primitives, such as curves, rectangles, and ellipses, as well as a mechanism for rendering virtually any geometric shape
- Mechanisms for performing hit detection on shapes, text, and images
- A compositing model that provides control over how overlapping objects are rendered
- Enhanced color support that facilitates color management
- Control of the quality of the rendering through the use of rendering hints

Painting with Graphics

- `JComponent` has a few methods that can be overridden in order to draw special things:

```
public void paintComponent(Graphics g){}

public void repaint();
```

- `java.awt.Graphics` - API allows drawing of:
  - Text strings
  - Geometry
  - Bitmap (raster) images
- The `Graphics` class is the abstract base class for all graphics contexts that allow an application to draw onto components
- All coordinates that appear as arguments to the methods of this `Graphics` object are considered relative to the translation origin of this `Graphics` object prior to the invocation of the method.

Drawing

- A line:

```
public abstract void drawLine(int x1, int y1, int x2, int y2);
```

- An outlined rectangle:

```
public abstract void drawRect(int x, int y, int width, int height);
```

- A filled rectangle:

```
public abstract void fillRect(int x, int y, int width, int height);
```

- A rectangle with fill of the background color:

```
public abstract void clearRect(int x, int y, int width, int height);
```

- An Oval outline:

```
public abstract void drawOval(int x, int y, int width, int height);
```

- A filled oval

```
public abstract void fillOval(int x, int y, int width, int height);
```

- An outline of a polygon, where the x and y are arrays of coordinates:

```
public abstract void drawPolygon(int[] x, int[] y, int numEdges);
```

- A filled polygon, where the x and y are arrays of coordinates:

```
public abstract void fillPolygon(int[] x, int[] y, int numEdges);
```

Working with images

- Creating an Image object:

```
try {
Image taz = javax.imageio.ImageIO.read(new File("taz.jpg"));
Image stewie = javax.imageio.ImageIO.read(new URL("http//link.to/
pic/));
}
catch(IOException exc) {exc.printStackTrace()};
```

- Drawing the image:

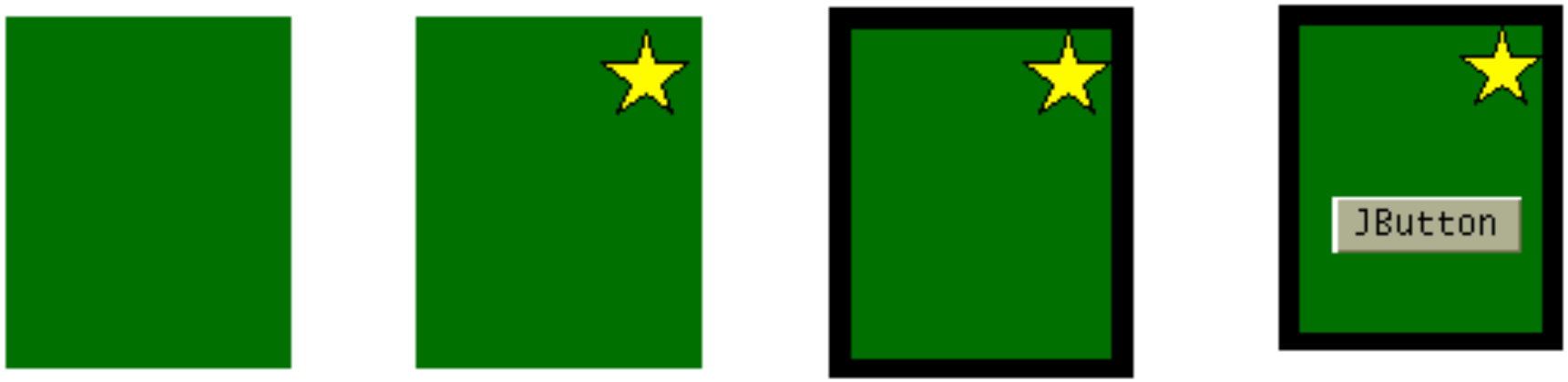
```
g.drawImage(taz, 0, 0, this);
g.drawImage(taz, 0, 200, 100, 100, this);
g.drawImage(taz, 200, 0, 200, 400, Color.yellow, this);
```

- General format:

```
boolean drawImage(Image img, int x, int y, int width, int height,
Color bgcolor, ImageObserver observer)
```

How painting works

1. background
2. custom painting
3. border
4. children



- Double-buffering: creates an in-memory drawing and then transfers it on the screen (avoids the "flickering" effect)

javax.swing.Timer class

- Generates events (`ActionEvent`) at some specified time intervals
- Useful for animations
  - the content of the frame is redrawn at fixed time intervals
- Listeners (`ActionListener`) needed for the events
- Specify the delay between the events

javax.swing.Timer class

- Constructor:

```
javax.swing.Timer t = new javax.swing.Timer(int ms,
ActionListener doIt);
```

- Usage (repaint a panel every second):

```
javax.swing.Timer t = new javax.swing.Timer(1000, new
ActionListener() {
            public void actionPerformed(ActionEvent e) {
                p.repaint();
            }});
```

- Start the timer: `t.start();`
- Stop the timer: `t.stop();`
- Restart the timer: `t.restart();`
- Additional methods:

```
t.setRepeats(boolean flag);
t.setInitialDelay(int initialDelay);
t.isRunning();
```