Tony Valderrama, SIPB IAP 2010

# PROGRAMMING IN

# Today

- Review of Object-Oriented Programming (OOP)
- Applets!
  - Threads
  - tvald.applet.AnimatedApplet (on website)
- Swing/AWT
  - Layout
  - Event model


Announcements

Course website: http://sipb.mit.edu/iap/java/

Email: sipb-iap-java@mit.edu

I need suggestions for advanced topics to cover on Thursday! (see website)

# Wake up and smell the coffee!

**Software**

Java Development Kit (JDK) - http://java.sun.com/javase/downloads/index.jsp

Eclipse Platform - http://www.eclipse.org/

**Reference**

The Java Tutorial - http://java.sun.com/docs/books/tutorial/index.html

Java Language API - http://java.sun.com/javase/reference/api.jsp

Java SE Documentation - http://java.sun.com/javase/downloads/index.jsp

Java SE Source Code - http://java.sun.com/javase/downloads/index.jsp

# OBJECT-ORIENTED


# PROGRAMMING

# What is a Class?

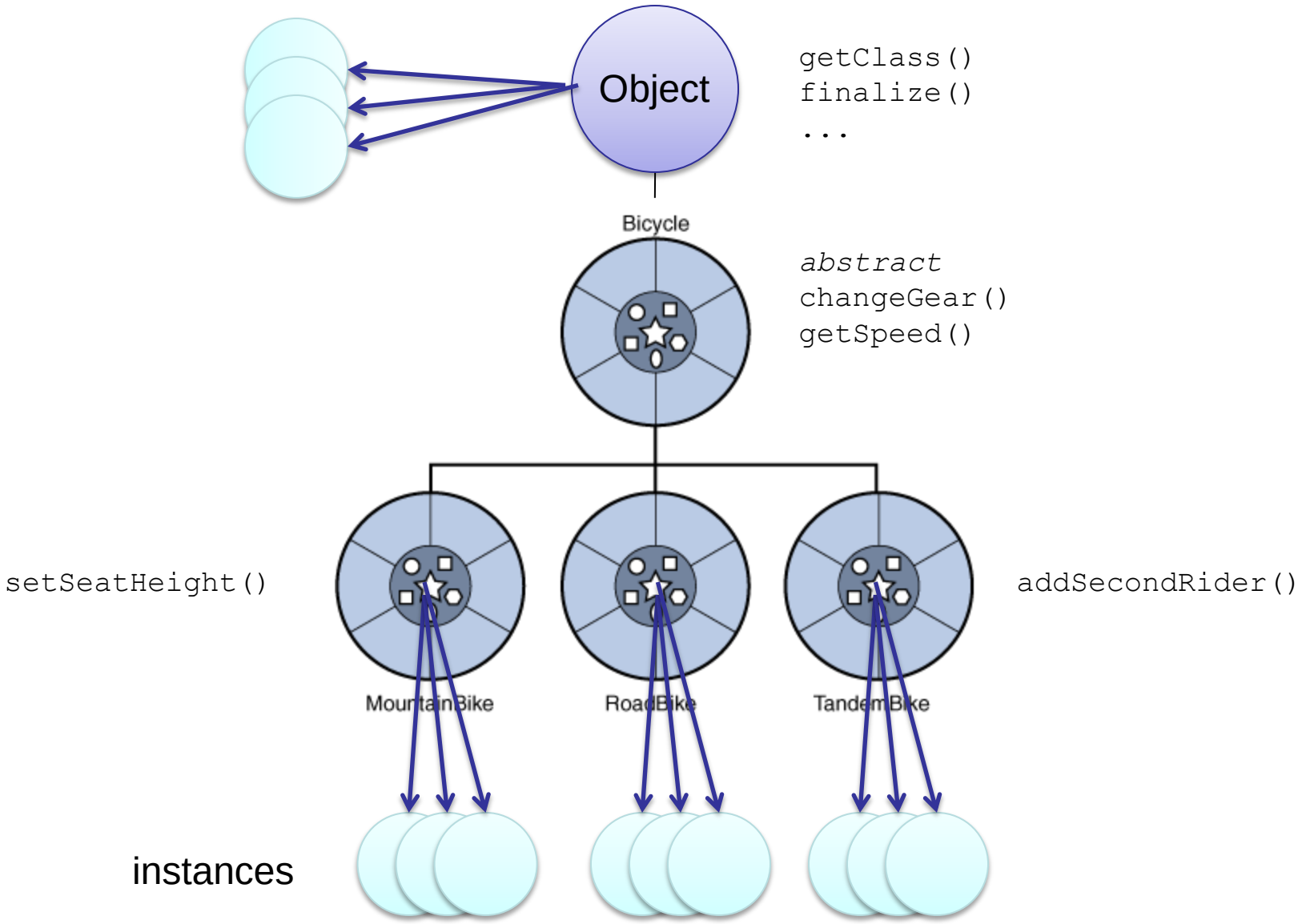
A class is a template, blueprint, or prototype from which objects are created.

```
<modifiers> class name {

// members: fields, methods, constructors

}
```

# Inheritance



# Constructors

```
class Bicycle {
    int gear,speed;
    public Bicycle(int startSpeed, int startGear) {
        gear = startGear;
        speed = startSpeed;
    }
    public Bicycle() { this(1, 0); }

    public static void main(String[] args) {
        Bicycle myBike = new Bicycle(0, 8);
        Bicycle otherBike = new Bicycle();
    }
}
```

```
modifiers classname (parameter list) throws exceptionlist {
    // method body
}
```

# Life cycle of an object

- instantiation
- do stuff
    - instantiate and manipulate other objects
    - serialize and send over network
    - interact with system libraries
- garbage collection
    - public void finalize()

# Object-Oriented Programming

Java is Object-Oriented from the ground up.

*EVERYTHING* is an Object. Even primitives can be wrapped in Objects.

Objects can have a lifetime greater than the object that created them.

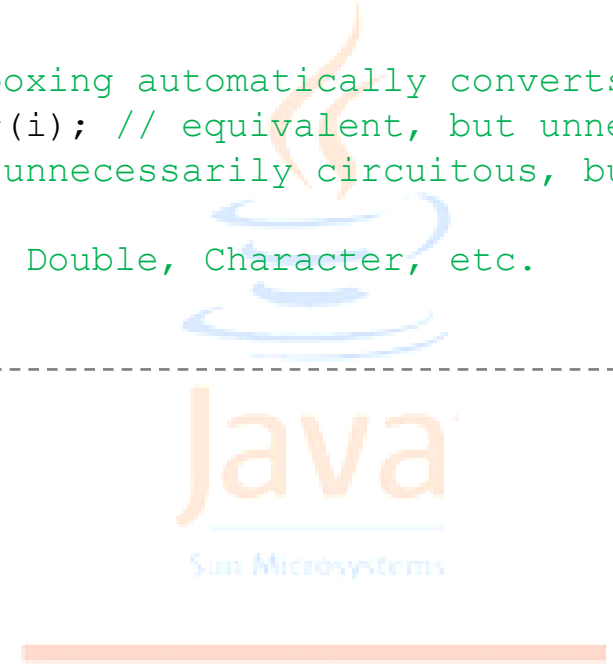
An Object-Oriented language should support:

- *Encapsulation* - information hiding and modularity (abstraction)
- *Polymorphism* - behavior is dependent on the nature of the object receiving a message
- *Inheritance* - new classes are defined based on existing classes to obtain code re-use and organization
- *Dynamic binding* - objects could come from anywhere, possibly across the network. Send messages to objects without knowing their specific type at the time you write your code.

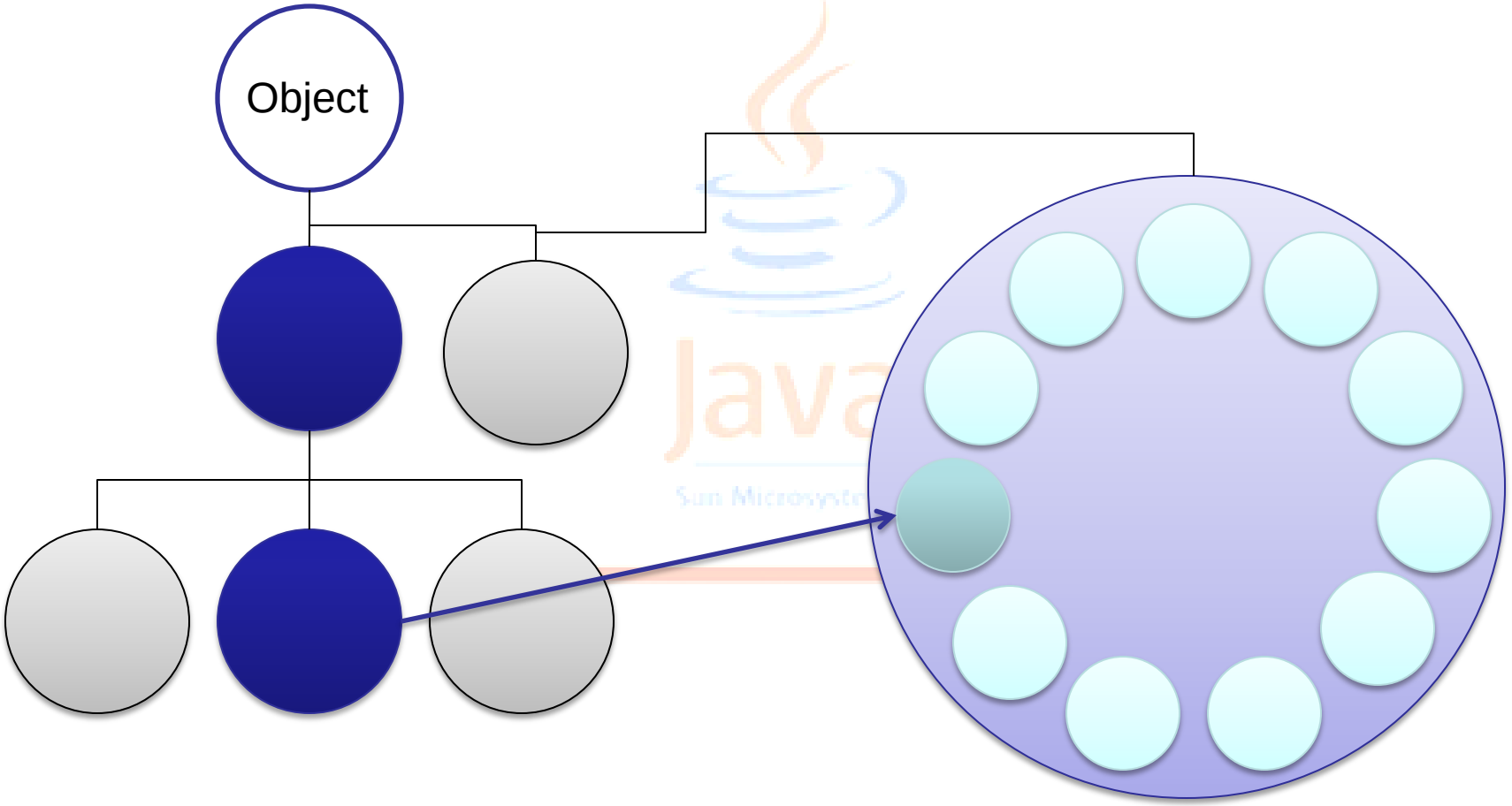
# Autoboxing

```
int i = 2;
Integer intObject;
intObject = i; // autoboxing automatically converts primitives
intObject = new Integer(i); // equivalent, but unnecessary
i = new Integer(4); // unnecessarily circuitous, but it works

// also: Boolean, Byte, Double, Character, etc.
```



# Inheritance vs. Composition



# What is an Interface?

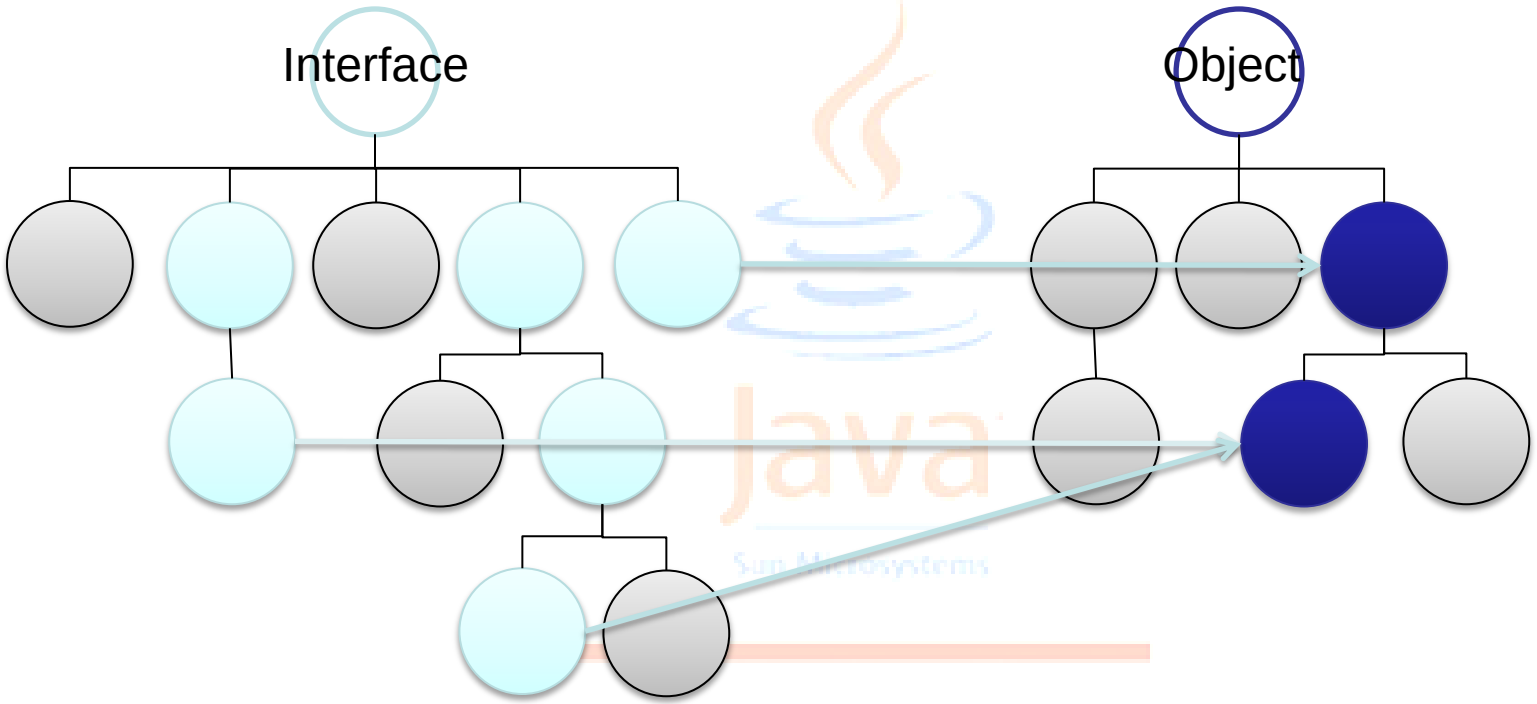
An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.

```
interface Bicycle {
    // constant declarations, if any
    public static final MAX_GEAR = 10;

    void changeGear(int newValue); //interfaces are completely abstract
    void speedUp(int increment);
    public abstract void applyBrakes(int decrement); // implied
}


class ACMEBicycle implements Bicycle {
   // remainder of this class implemented as before
}
```

# Interfaces and Classes



# APPLETS

# Applets

A special kind of Java application that can be loaded into a web browser.

demo.html

```
<html>
<body>
<applet code="HelloWorld.class" width="200" height="200"></applet>
<body>
</html>
```

HelloWorld.java

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawRect(0, 0,
                   getSize().width - 1,
                   getSize().height - 1);
        g.drawString("Hello world!", 5, 15);
    }
}
```

# Graphics

```
java.awt.Graphics;

...

public void paint(Graphic g) {
        g.drawLine(x1,y1,x2,y2);
        g.drawRect( x, y, w, h);
        g.fillRect( x, y, w, h);
        g.drawOval( x, y, w, h);
        g.fillOval( x, y, w, h);
        g.drawString(x, y);

        g.setColor(new Color(r, g, b));
}
```

# Life cycle of an Applet*

**init**:
Initialize your applet. Called when the applet first loads.

**start**:
Called whenever the user visits the page containing the applet.

**paint**:
This method updates the graphical interface.

**stop**:
Called whenever the user moves away from the page containing applets.

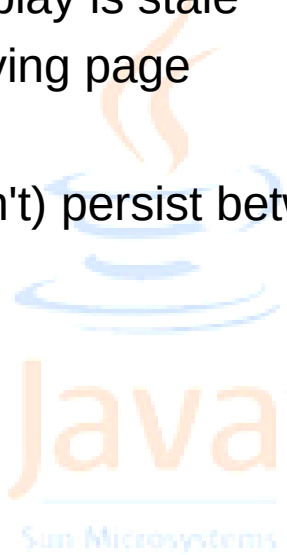
**destroy**:
Clean up (like finalize). Called when the applet is unloaded.

* Not really

# The *real* life cycle of an Applet

**init, start:** Called upon navigation to page
**paint:** Called when display is stale
**stop, destroy:** Called upon leaving page

Static fields *may* (but probably won't) persist between visits.




(Yes, browsers suck at compatibility and at following specifications…)

# Files

```
// retrieve an Image
java.awt.Image image = getImage(getCodeBase(), "imgDir/a.gif");

g.drawImage(image, x, y);


// retrieve a file
File f = new File("config.txt");


// all file paths are resolved from the Applet source directory
```

# Threads

Process: a self-contained execution environment, usually with its own memory space.

Thread: a *lightweight process*; a thread exists within a single process, sharing processor resources

http://www.doc.ic.ac.uk/~jnm/concurrency/classes/ThreadDemo/ThreadDemo.html

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

# Threads

Process: a self-contained execution environment, usually with its own memory space.

Thread: a *lightweight process*; a thread exists within a single process, sharing processor resources

http://www.doc.ic.ac.uk/~jnm/concurrency/classes/ThreadDemo/ThreadDemo.html

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

# Dealing with traffic

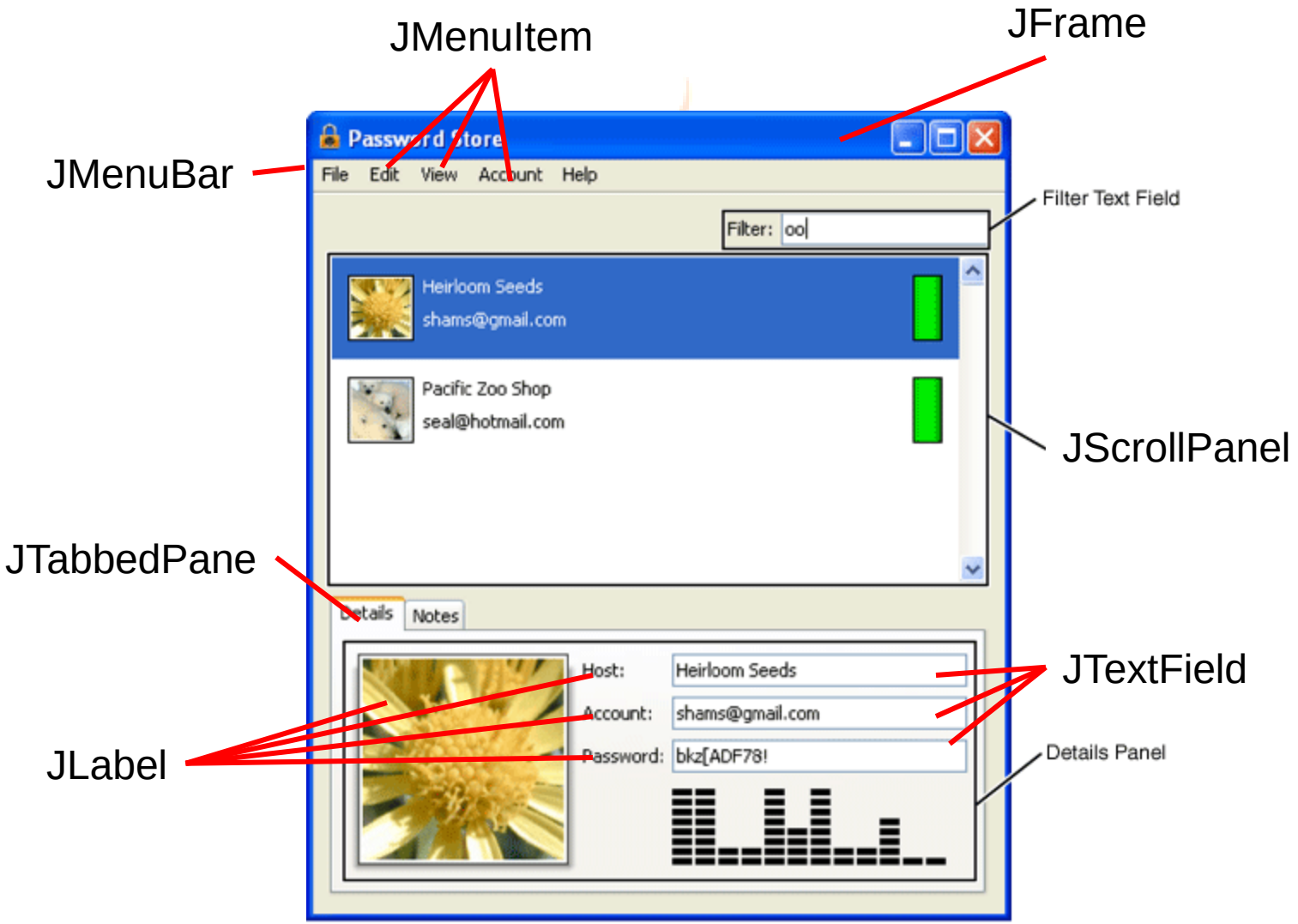
```
public class Thread1 extends Thread {
    public void run() {
        Thread.sleep(2000);
        Thread otherThread = new ThreadThatDoesSomething();
        otherThread.join(); // wait for the thread to finish
        // do stuff
    }

// synchronization:
// neither of these can be called at the same time.
// a second call will cause the calling Thread to
// block until it execution has completes
    public synchronized foo() { }
    public synchronized bar() { }

    // lock an object
    public void foo() {
        // locks object
        synchronized(object) {
                statement(s)...
        }
        statement(s)...
}
```

# SWING/AWT

# Modeling Widgets as Objects

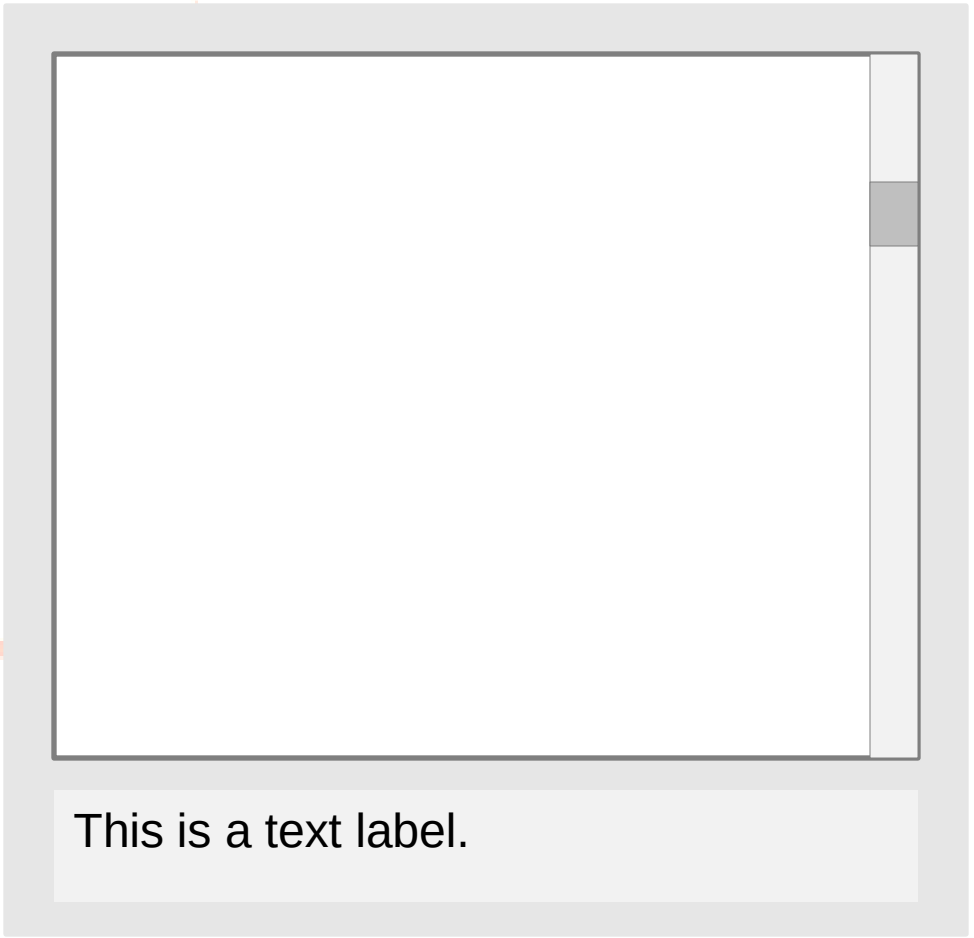


# AWT/Swing

GUI Widgets are called *Components*, and extend the class JComponent.

Most components are containers, which means they can hold other components.

How many components are in this example?



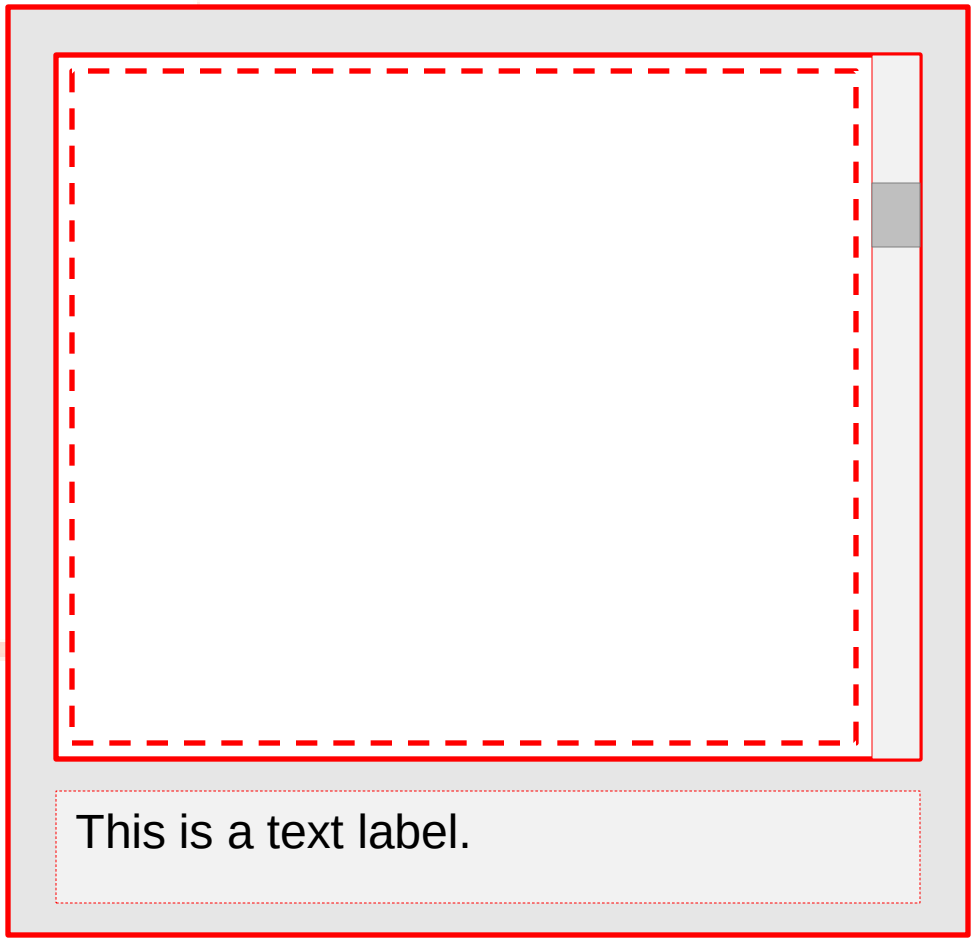
# AWT/Swing

GUI Widgets are called *Components*, and extend the class JComponent.

Most components are containers, which means they can hold other components.

How many components are in this example?

5

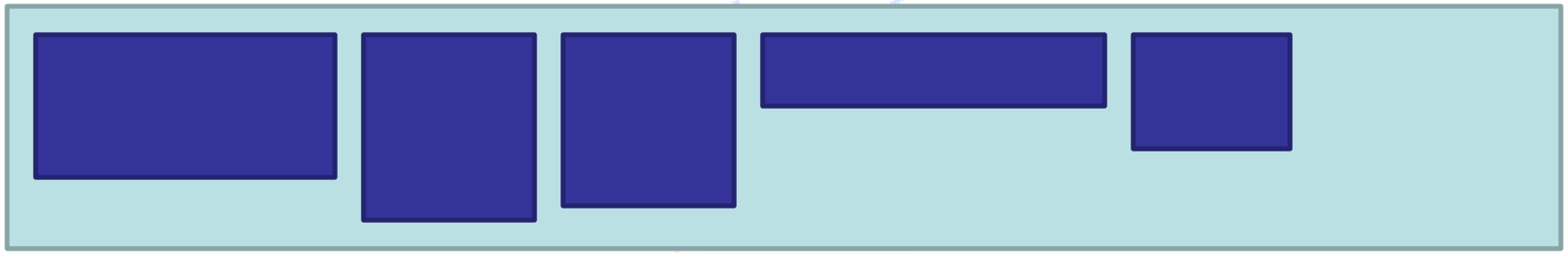



Gallery of Components:
http://java.sun.com/docs/books/tutorial/ui/features/components.html

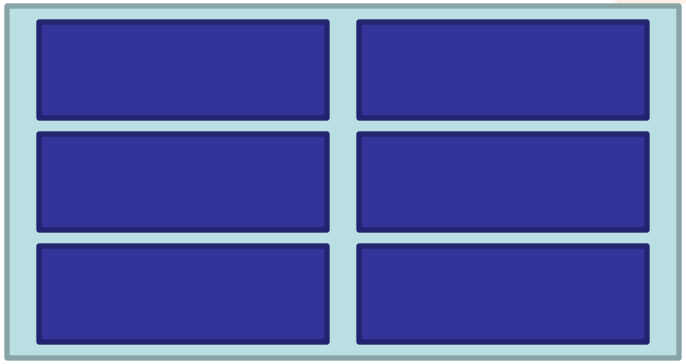
# Laying Out Components

Every container has a layout manager which dynamically controls the *size* and *location* of its children. When you add a child component, you specify options that affect the child's layout.

FlowLayout.HORIZONTAL

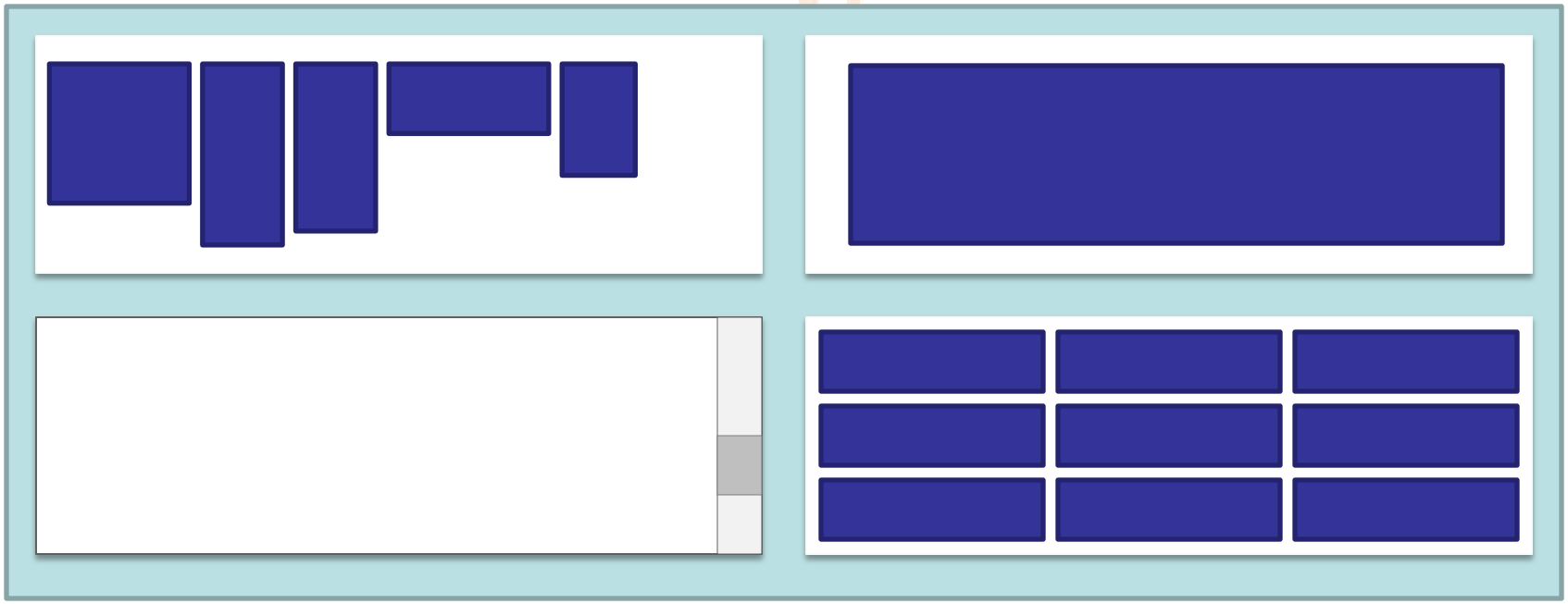


GridLayout



# Laying Out Components

Combine layouts and containers to create the GUI.



Gallery of layouts:
http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html

# Event s and Listeners

Graphical User Interfaces (GUIs) are **event-driven**. Most of the time they sit around waiting for user input or other things to happen.

**Events** describe actions, like pressing a key, clicking the mouse, resizing the window, etc.

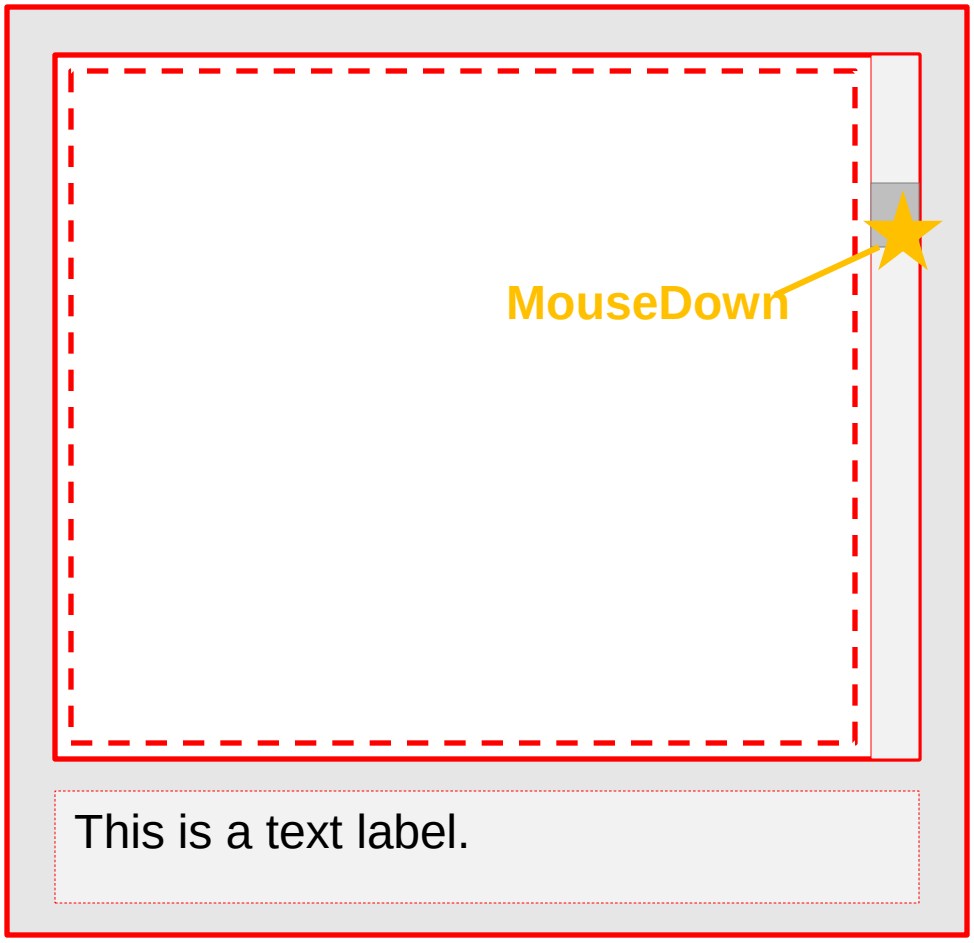
**Listeners** handle events.

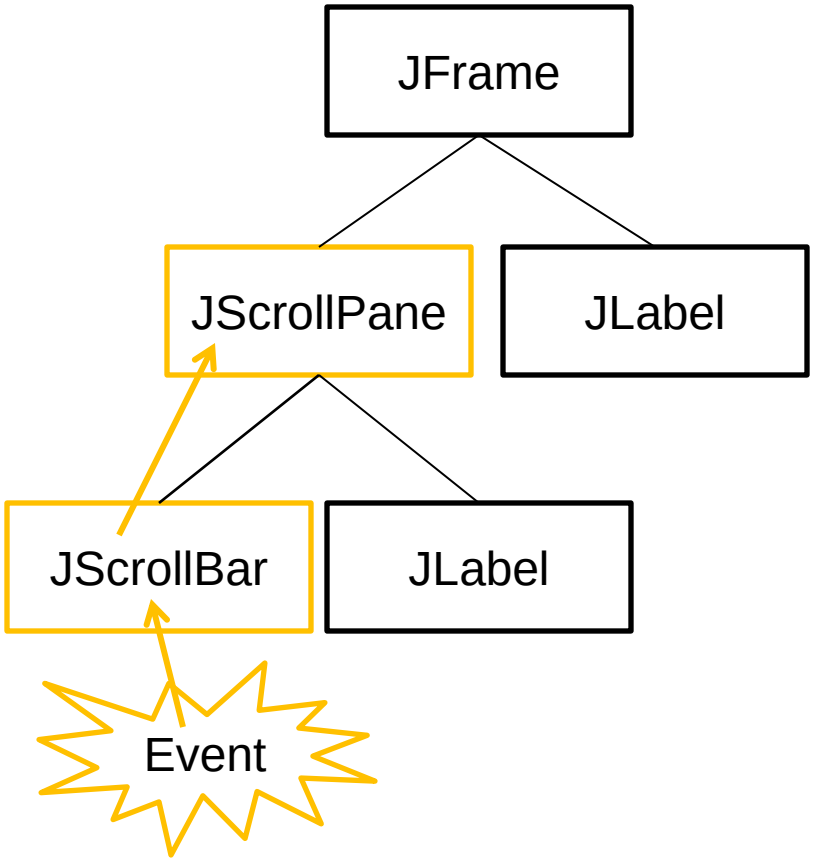
```
public class ClickHandler extends MouseListener {

    public void initialize() {
        JComponent button = new JButton("Click Me!");
        button.addMouseListener(this);
    }

    public void MouseDown(MouseEvent e) {
        // do something
    }
}
```

# Event s and Listeners




Events "bubble" up the layout tree until a component handles the event.

# Thanks for coming!