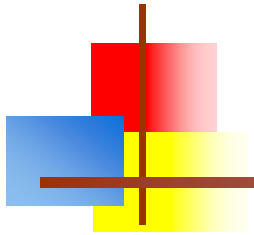
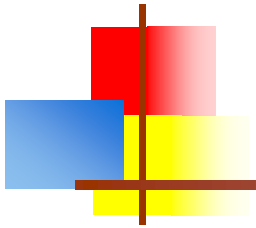


Animation

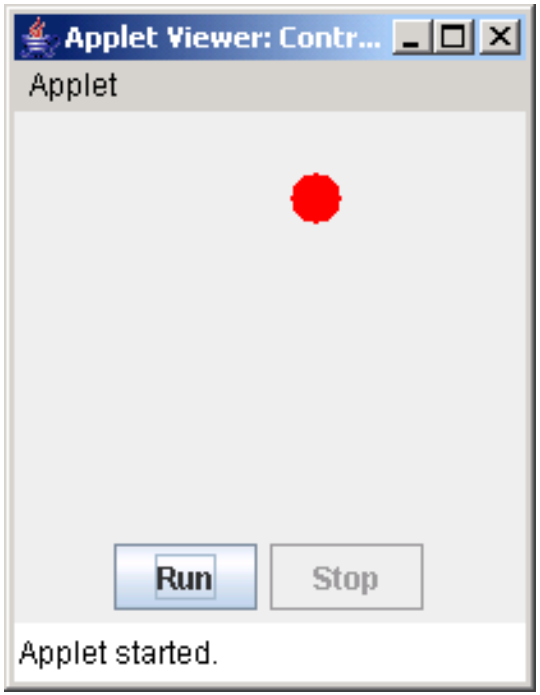


Moving pictures

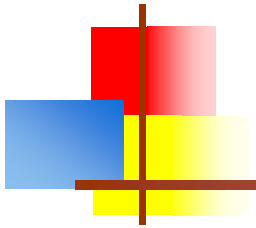
- Animation—making objects that appear to move on the screen—is done by displaying a series of still pictures, one after the other, in rapid succession
  - Generally you should try for at least 20 pictures/second
  - 20 pictures/second is repainting every 50 milliseconds

The bouncing ball

- "Bouncing ball" is the "Hello World" of animation

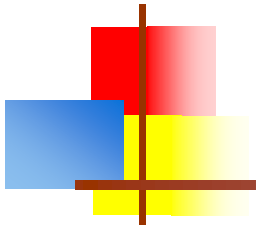


- We will develop this program using:
  - Model-View-Controller
  - Observer-Observable
  - Threads
  - Timers



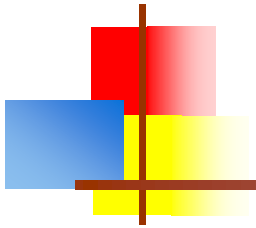
Review of MVC

- MVC stands for Model-View-Controller
  - The Model is the actual internal representation
    - It should be independent of the other classes
    - It's handy if this class can extend Observable
  - The View (or a View) is a way of looking at or displaying the model
    - It's handy if this class implements Observer
  - The Controller provides for user input and modification
- These three components are usually implemented as separate classes (or sets of classes)



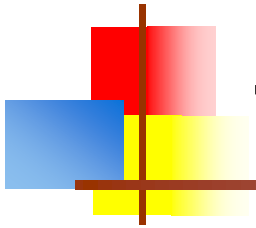
Review of Observer and Observable

- java.util.Observable is a *class*
  - When it does something that should be observed, it says:
    - setChanged();
    - notifyObservers(); /* or */ notifyObservers(***arg***);
- java.util.Observer is an *interface*
  - It has to register itself with (subscribe to) an Observable:
    - ***myObservable***.addObserver(***myObserver***);
  - It has to implement:
    - public void update(Observable ***obs***, Object ***arg***)
    - This method is automatically called when observers are notified
    - ***obs*** is the object being observed
    - If the Observable did notifyObservers(), ***arg*** is null



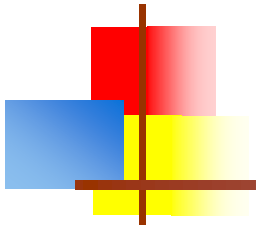
Review of Threads

- You can extend the Thread class:
  - class Animation extends Thread {…}
  - Limiting, since you can only extend one class
  - You must override public void run( )
  - To make it "go":
    - Animation anim = new Animation( );
      anim.start( );
- Or you can implement the Runnable interface:
  - class Animation implements Runnable {…}
  - You must implement public void run( )
  - To make it "go":
    - Animation anim = new Animation( );
      Thread myThread = new Thread(anim);
      myThread.start( );



Timers

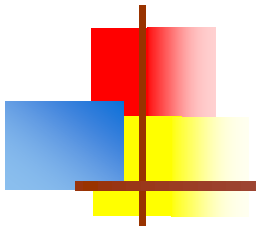
- A java.util.Timer is used to schedule code for future execution
- A Timer may:
  - Schedule a one-time execution, or
  - Schedule repeated executions at regular intervals
- Timer constructors:
  - Timer()
  - Timer(boolean ***isDaemon***)
  - Timer(String ***name***)
  - Timer(String ***name***, boolean ***isDaemon***)
- A Timer can keep an application from terminating, unless it is specified as a daemon thread
  - A daemon thread dies if there are no no-daemon threads running
  - Create daemon Timer threads with new Timer(true) or new Timer(***name***, true)



Using a Timer for animation

- public void schedule(TimerTask ***task***, long ***delay***, long ***period***)
  - Schedules the specified task for repeated fixed-delay execution, beginning after the specified delay (which may be zero)
  - Subsequent executions take place at approximately regular intervals separated by the specified period
  - Times are specified in milliseconds (1/1000s of a second)

- Notice that schedule requires a TimerTask as an argument
  - TimerTask is an abstract class you must extend and provide a public void run() method
  - TimerTask provides an (implemented) public boolean cancel() method
    - Returns false if there were no scheduled executions to cancel



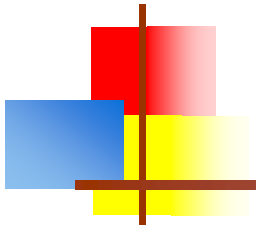
The Model class, I

```
import java.util.Observable;

class Model extends Observable {
    public final int BALL_SIZE = 20;
    private int xPosition = 0;
    private int yPosition = 0;
    private int xLimit, yLimit;
    private int xDelta = 6;
    private int yDelta = 4;

    // methods (on next slide)

}
```



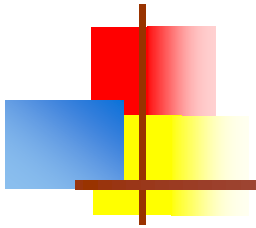
The Model class, II

```
public void setLimits(int xLimit, int yLimit) {
    this.xLimit = xLimit - BALL_SIZE;
    this.yLimit = yLimit - BALL_SIZE;
}

public int getX() {
    return xPosition;
}

public int getY() {
    return yPosition;
}

public void makeOneStep() {
    // code for making one step (on next slide)
}
```



The Model class, III

```
public void makeOneStep() {
    // Do the work
    xPosition += xDelta;
    if (xPosition < 0 || xPosition >= xLimit) {
        xDelta = -xDelta;
        xPosition += xDelta;
    }
    yPosition += yDelta;
    if (yPosition < 0 || yPosition >= yLimit) {
        yDelta = -yDelta;
        yPosition += yDelta;
    }
    // Notify observers
    setChanged();
    notifyObservers();
}
```

The View class

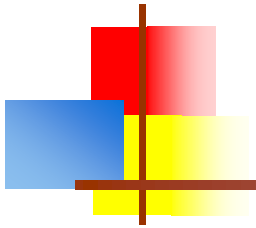
```
import java.awt.*;
import java.util.*;
import javax.swing.JPanel;

class View extends JPanel implements Observer {
    Model model;

    View(Model model) {
        this.model = model;
    }

    @Override
    public void paint(Graphics g) {
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(Color.RED);
        g.fillOval(model.getX(), model.getY(),
                model.BALL_SIZE, model.BALL_SIZE);
    }

    public void update(Observable obs, Object arg) {
        repaint();
    }
}
```



The Controller class, I

```
import java.awt.*;
import java.awt.event.*;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.*;

public class Controller extends JApplet {
    JPanel buttonPanel = new JPanel();
    JButton runButton = new JButton("Run");
    JButton stopButton = new JButton("Stop");
    Timer timer;

    Model model = new Model();
   View view = new View(model);  // View must know about Model
```

The Controller class, II

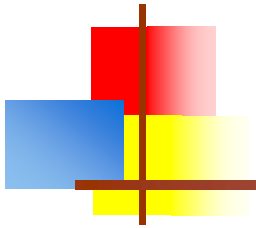
```
@Override
public void init() {
    layOutComponents();
    attachListenersToComponents();
    // Connect model and view
    model.addObserver(view);
}
private void layOutComponents() {
    setLayout(new BorderLayout());
      this.add(BorderLayout.SOUTH, buttonPanel);
         buttonPanel.add(runButton);
         buttonPanel.add(stopButton);
         stopButton.setEnabled(false);
      this.add(BorderLayout.CENTER, view);
}
```

The Controller class, III

```
private void attachListenersToComponents() {
    runButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            runButton.setEnabled(false);
            stopButton.setEnabled(true);
           timer = new Timer(true);
           timer.schedule(new Strobe(), 0, 40); // 25 times a second
        }
    });
    stopButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
           runButton.setEnabled(true);
           stopButton.setEnabled(false);
           timer.cancel();
        }
    });
}
```

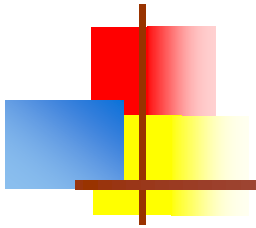
The Controller class, IV

```
@Override
private class Strobe extends TimerTask {
    public void run() {
        model.setLimits(view.getWidth(), view.getHeight());
        model.makeOneStep();
    }
}
```



Summary

- In this program I used:
  - Model-View-Controller
    - This is a good design pattern for many uses; it separates the "business logic" (the Model) from the classes that are basically I/O
  - Observer-Observable
    - This is a good design pattern for helping to isolate the Model from the View
  - Threads
    - If you want to have a *controllable* animation, Threads are essential
    - The animation runs in one Thread, the controls in another
  - Timers
    - Timers are a convenient way to schedule regularly repeating tasks
    - With a slightly different design, you could use `Thread.sleep(`***ms***`)` instead



The End