CS193J: Programming in Java
Summer Quarter 2003

Lecture 7
Repaint, Mouse, Advanced Drawing, Object Serialization

Manu Kumar
sneaker@stanford.edu
• 2 Handouts for today!
  – #17: Advanced Drawing
  – #18: Object Serialization
Recap

• Last Time
  – HW#1 Feedback
  – HW#2 Live Demo
    • Link between lecture materials and homework
  – Inner Classes
  – Anonymous Inner Classes
  – Listener model
    • Button Listener Example
  – Repaint
    • Left off before Repaint example
Lecture-Homework mapping revisited

• HW #2 will use
  – OOP concepts
    • Inheritance, overriding, polymorphism
    • Abstract classes
  – Drawing in Java
    • Layouts
    • paintComponent()
  – Event handling
    • Anonymous Inner classes
  – Repaint (continues Today)
  – Mouse Tracking (Today)
  – Advanced Drawing (Today)
  – Object Serialization (Today/Thursday)
Today

• Continue with Repaint
  – Repaint example code walkthrough
  – Erasing

• Mouse Tracking
  – DotPanel example code walkthrough

• Advanced Drawing
  – Region based drawing, Blinking, Smart Repaint

• Object Serialization
  – Cloning
    • Not Dolly, but Java Objects 😊
  – Serializing
Review

• Control-Listener Theory
  – Source
    • Buttons, controls etc.
  – Listener
    • An Object that wants to know when the control is operated
  – Notification Message
    • A message sent from the source to the listener as a notification that the event has occurred

• Essentially: registering callbacks
Source-Listener Interaction

Source Component

addListener

Listener

addListener

Notification of event

Listener
Anonymous Inner class
  – Most common method!
  – Create an Anonymous Inner Class that implements the interface
    • Can be created on the fly inside the method!

```java
button = new JButton("Beep");
panel.add(button);
button.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Toolkit.getDefaultToolkit().beep();
        }
    }
);```
public ListenerFrame() {
    super("ListenerFrame");

    JComponent content = (JComponent) getContentPane();
    content.setLayout(new FlowLayout());

    JButton button = new JButton("Beep!");
    content.add(button);

    // ----
    // Creating an action listener in 2 steps...

    // 1. Create an inner class subclass of ActionListener
    ActionListener listener =
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Toolkit.getDefaultToolkit().beep();
            }
        };

Button Listener Example

// 2. Add the listener to the button
button.addActionListener(listener);

// ----
// Creating a listener in 1 step...

// Create a little panel to hold a button
// and a label
JPanel panel = new JPanel();
content.add(panel);
JButton button2 = new JButton("Yay!");
label = new JLabel("Woo Hoo");
panel.add(button2);
panel.add(label);
Button Listener Example

// This listener adds a "!" to the label.
button2.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String text = label.getText();
            label.setText(text + "!");
            // note: we have access to "label" of
            // outer class
            // we do not have access to local vars
            // unless they are declared final.
        }
    }
);

pack();
setVisible(true);
• Repaint is **asynchronous**
  – It does not do the drawing immediately
    • It “requests” the system to call `paintComponent()`
  – Behind the scenes
    • The System maintains an event queue
    • `repaint()` simply adds a request on the event queue
    • The system draw thread will dequeue the draw request and ultimately call `paintComponent()`

• Do not call `paintComponent()`!
  – Call `repaint()` and the system will schedule a call to `paintComponent()`
• Setters
  – Change the object state
• Whenever object state is changed
  – Call repaint() to keep the pixels in sync
Face Repaint Example

- Default state:
  - Smiley face
  - ivar: boolean angry = false
- paintComponent()
  - Looks at value of angry ivar to change color accordingly
  - Draws the smiley

```
// smiley -- draws in red if angry
public void paintComponent(Graphics g) {
    if (angry) g.setColor(Color.red);
    else g.setColor(Color.blue);
    // draw smiley
}
```
• Setter Repaint Pattern in the example
  – setAngry() should call repaint
    public void setAngry(boolean angry) {
      this.angry = angry;
      repaint();
    }

• Could be intelligent and call repaint only when needed
  public void setAngry(boolean angry) {
    if (this.angry != angry) {
      this.angry = angry;
      repaint();
    }
  }
Repaint Example
• Code walk through….

  – Widget.java
  – Boxer.java
  – Repaint.java
    • Layout
    • Event handling with listeners
• We do not actively erase in java
  – To erase something, simply don’t draw it in `paintComponent`

• `paintComponent` starts out with a erased canvas
  – Draws components back to front
    • *What you draw later is drawn on top*

• Again
  – To erase something, just don’t draw it
Fish Example

- Fish with a hat
- Fish without a hat
The Fish class…

```java
void paintComponent() {
    // draw fish body
    if (hasHat) // draw the hat
    }
void setHat(boolean hat) {
    hasHat = hat;
    repaint();
}

• Scenario: fish.hasHat is true. Send fish.setHat(false) -- the hat disappears
```
Boxer example

- Boxer draws the image when image ivar is not null
  - To erase the image – set the image ivar to null and repaint
Smart Repaint

• Painting the screen can be time consuming
  – One approach is to paint only those region which need to be painted
  – System already does this for most events (expose, resize, scroll etc)

• But
  – The programmer can also be intelligent and tell the system which regions need painting
  – Done with repaint(Rectangle r)
    • Repaint just old+new rectangles when a component moves
    • We will see more of this soon…
MouseListener and MouseMotionListener

- To get notification about mouse event over a component
- The component itself is the source of the notification

• Add the listener to the component
Listener vs. Adapter Style

• Problem
  – Listener has a bunch of abstract methods
    • 5 in MouseListener
  – We typically care only about implementing one or two

• Solution
  – “Adapter” classes have empty {} definitions of all methods
  – Only need to implement the ones we care about
    • The adapter catches the others

• Gotcha
  – If you write your method prototype wrong you won’t override the empty {} implementation in the adapter!
    • Example MousePressed() instead of mousePressed()
public interface MouseListener extends EventListener {
    /**
     * Invoked when the mouse has been clicked on a component.
     * (press+release)
     */
    public void mouseClicked(MouseEvent e);
    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e);
    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e);
    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e);
    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e);
}
public abstract class MouseAdapter implements MouseListener {
    /**
     * Invoked when the mouse has been clicked on a component.
     */
    public void mouseClicked(MouseEvent e) {}
    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e) {}
    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e) {}
    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e) {}
    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e) {}
}
• How does a component handle a mouse press?

```java
component.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        // called when mouse button first pressed on component
    }
});
```
• How does a component detect a mouse movement?

```java
component.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        // called as mouse is dragged, after initial click
    }
});
```
Delta rule for mouse motion

• Cannot use absolute coordinates for mouse movement!
  – Setting the position to the actual mouse coordinated may result is weird movements

• Correct approach
  – Get the current coordinates
  – Compare to the last known coordinates
    • Compute the delta
  – Apply the delta to the position of the object

• Test-case
  – A click-release with no motion should not change any state in a correct implementation of relative mouse tracking
DotPanel Example
DotPanel Example Code

• Code walkthrough…
  – DotPanel.java
• JPanel
  – Simple component that draws itself
  – Subclass of JComponent
  – Use setBackground to get an automatic background color
  – Use setOpaque(true) in order to tell the system that we are drawing every pixel
    • Optimization since then the system doesn’t draw what is behind us
  – Call super.paintComponent() from paintComponent()
    • Graphics will be erased to background color
Clipping Region

- The 2D region within which the system will accept changes to what is shown on the screen
  - Any pixel changes outside the clipping region are ignored.

- System sets a “clipping region” on the Graphics object before sending paintComponent()
  - Affects all drawing operations
    - Pixels outside clipping region do not get affected
  - By default is set to the bounds of the component
    - Basic drawing case works fine – nothing special needed
    - Room to optimize for better performance
component.getGraphics() -- NO

- component.getGraphics()
  - Almost never right to use component.getGraphics()
  - There may be special cases, but in general, this goes against the system/paintComponent paradigm
Repaint Details

- Repaint call tells system what region to redraw
  - repaint() uses bounds
  - repaint(Rectangle) uses a sub-rectangle
- System maintains “update region”
  - A 2D representation of areas that need to be redrawn
  - Repaint call adds a region to the update region
- System paint thread
  - Checks regions to be updated
  - Computes intersection of region vs. components
  - Initiated draw recursion down the component nesting hierarchy
  - Composites pixels back to front
Region Based Drawing

• The drawing area is always expressed as a region not in components
  – Handles intersections and z-order correctly

• Z-order
  – Visual layering of components

• Mechanics
  – Draw all the components that intersect the pixel region
  – Draw the components from back to front
Moving components

• When a component moves
  – Update the old region
    • Redraw any exposed components or erase moved component
  – Update the new region
    • Redraw the component at it’s new location
Smart Repaint revisited

- Repaint just the rectangle of the component that needs to be redrawn
  - Not the entire component or window bounds
- Makes the drawing cycle faster
  - Smoother drawing, esp if clipping region is small
- repaint(x, y, width, height) does this
- Must repaint both old and new regions
  - Union of old and new clipping rectangles
Coalescing

• Intelligently combining multiple repaint() requests into a single draw operation
  – Benefit of asynchronous repaint() calls
• No 1-1 correspondence between repaint() and paintComponent() calls
  – Multiple repaints can be coalesced by the system and handled by a single paintComponent() call
• Time: Multiple repaint requests are “coalesced” into one draw operation
  – You can repaint() 3 times, but it just draws once
• Space: Repaint regions may overlap, but the areas of intersection is drawn once
  – System is maintaining the update region
Coalescing Example

• JSlider in Repaint example
  – As the slide moves it sends multiple setCount() messages to the Widget
    • If we move it quickly it would result in lots of calls
  – However, it doesn’t redraw every state
    • The previous states would all be overwritten by the last state anyway
  – Draws the last state by coalescing the repaint() calls and calling paintComponent less (possibly just once) times
Blinking Animation

• Animation Steps
  – Draw old state on the screen
  – Erase the old state and restore the background
  – Draw the new state on the screen

• Problem
  – Erasing the old state and restoring the background results in a blinking effect! 😞
  – If the redraw is fast, it looks like a “shimmer”
    • Still undesirable
Solution: Double Buffering

• Concept:
  – Do all the erasing and drawing in memory before copying the final changes to the screen

• Mechanics
  – Build a pixel buffer offscreen (called offscreen graphics)
  – Draw the old appearance
  – Erase offscreen buffer
  – Draw the new appearance to the offscreen buffer
  – Copy final bits (aka “blit”) to the onscreen graphics

• Result
  – Smooth animation since we minimize the changes on the onscreen graphics
Swing is double buffered!

- Swing double buffers automatically
  - All JComponent drawing goes through a offscreen buffer
  - Graphics object passed to paintComponent is pointer to an offscreen buffer
- Makes life easier for us as the programmer!
Smart Repaint Implementation

- Start with the region to draw, but make it smaller
- Find intersection of components
- Allocate an offscreen bitmap
  - Exactly the size of the small update region
- Setup the origin and the clip of Graphics g to point to the small offscreen buffer
  - Drawing outside the buffer is clipped, but components do not need to do anything special
- Copy the small buffer to the screen when done
  - Smaller the region, faster the copy
Smart Repaint Conclusion

• Using repaint(rect) to redraw just a region of the component can be a lot faster
  – Client components don’t need to know what is going on, they just respond to paintComponent()

• Calling repaint(x, y, width, height)
  – System is smart about using an offscreen buffer of the size needed
    • Great potential speedup

• Theme: with little work, JComponent can do some complex drawing
Example #1

• Circle and rectangle
  – Changing the circle to be filled with a pattern
    
    ![Circle and rectangle diagram]

  – State change ➔ Repaint ➔ Update Region
    • Change the state of the circle to pattern = true
    • Repaint just around the circle
    • Add the square to the update region
Example #1 continued

- Offscreen drawing
  - Draw thread notices update region
  - Creates offscreen buffer of same size
    - Notice how fewer pixels need to be re-used
  - Clipping is set around the buffer
    - Pixels outside clipping region have no effect
  - Draw thread sends paintComponent() to the components to draw themselves back to front
    - Only the parts that intersect the update region actually draw
Example #1 continued

- Copy bits
  - Once all the drawing is done draw thread copies the buffer back to the screen with a fast copy ("blit") operation
  - Deletes the offscreen buffer
Example #2: moving

• Move circle down
• Repaint
  – Old rectangle
  – New rectangle
Example #2: Moving continued

• Offscreen graphics
  – Same as before!
Example #2: Moving continued

- Copy bits to screen
  - Delete offscreen buffer
• Equals revisited
  – `a == b` tests for pointer equality only
    • i.e. pointer `a` and `b` point to the same location/object
    • This is called “shallow semantics”
  – `boolean Object.equals(Object other)`
    • Defined in the `Object` class
      – Default implementation does `a == b` test (shallow semantics)
    • May override to do “deep comparison”
      – Example: `String.equals()`
Calling equals()

```java
{
    String a = "hello";
    String b = "hello";

    (a == b) \(\rightarrow\) false
    (a.equals(b)) \(\rightarrow\) true
    (b.equals(a)) \(\rightarrow\) true
}
```
Equals strategy

• boolean equals(Object other)
  – Take Object, return boolean
    • Must have exact prototype for overriding to work
  – Return true on (this == other)
  – Use (other instanceof Foo) too test class of other
    • False if not same class
  – Otherwise do a field-by-field comparison of this and other
// in Student class...

boolean equals(Object obj) {
    if (obj == this) return(true);
    if (!(obj instanceof Student)) return(false);
    Student other = (Student)obj;
    return(other.units == units)
}
Cloning

• Used to create a copy of an object
  – Not just another pointer to the same object
  – Cloned object has it’s own memory space
• Lets say `Foo b = a.clone();`
  • `a == b` will return false
  • `a.equals(b)` will return true!
• Copied object has same state
  – But its own memory
• We use this in HW#2 for cut-copy-paste!
Cloneable interface

• Used as a marker to indicate that the class implements the clone() method
  – Not compiler enforced
  – Object.clone() is pre-built
    • Create a new instance of the right class
    • Assign all fields over with ‘=‘ semantics
• Object.clone() will do above default behavior
  – If class implements the cloneable interface
  – Otherwise, it will throw an exception
Implementing clone()

- Implement the Cloneable interface
  - Call the super classes clone method first to copy structure
    - copy = (Class) super.clone()
  - Copy fields where a simple ‘=‘ is not deep enough
    - Example, arrays, arraylists, objects
Alternative approaches

• Copy Constructor
  – MyClass(MyClass myObject)
    • Construct a new instance of MyClass based on the state of MyObject

• “Factory” method
  – Static method that makes new instances
    • static MyClass newInstance(MyClass myObject)
    • May use constructor internally

• Advantage
  – Simpler than Object.clone(), no new concepts

• Disadvantage
  – Client must know the class of the Object
// Eq.java

/*
   Demonstrates a simple class that defines equals and clone.
*/
public class Eq implements Cloneable {
    private int a;
    private int[] values;

    public Eq(int init) {
        a = init;
        values = new int[10];
    }
}
/*
   Does a "deep" compare of this vs. the other object.
*/
public boolean equals(Object other) {
    if (other == this) return(true);
    if (!(other instanceof Eq)) return(false);

    Eq e = (Eq) other;

    // now test if this vs. e
    if (a != e.a) return(false);

    if (values.length != e.values.length) return(false);
    for (int i=0; i<values.length; i++) {
        if (values[i] != e.values[i]) return(false);
    }
    return(true);
}
Eq Code example: clone()

/*
   Returns a deep copy of the object.
*/
public Object clone() {
    try {
        // first, this creates the new memory and does '=' on all fields
        Eq copy = (Eq)super.clone();

        // copy the array over -- arrays respond to clone() themselves
        copy.values = (int[]) values.clone();
        return(copy);
    }
    catch (CloneNotSupportedException e) {
        return(null);
    }
}
public static void main(String[] args) {
    Eq x = new Eq(1);
    Eq y = new Eq(2);
    Eq z = (Eq) x.clone();

    System.out.println("x == z" + (x==z));   // false
    System.out.println("x.equals(z)" + (x.equals(z))); // true
}
}
Serialization

• Motivation
  – A lot of code involves boring conversion from a file to memory
    • Write code in 106A to translate by hand
    • HW#1 read ASCII file and required parsing
  – This is a common problem!

• Java’s answer:
  – Serialization
    • Object know how to write themselves out to disk and to read themselves back from disk into memory!

• We use this in HW#2 to load and save!
Serialization / Archiving

- Objects have state in memory
- Serialization is the process of converting objects into a streamed state (Network, Disk)
  - No notion of an address space
  - No pointers
- Serialization is also called
  - Flattening, Streaming, Dehydrate (rehydrate = read), Archiving
How it works?

• To write out an object
  – ObjectOutputStream out;
  – out.writeObject(obj)

• To read that object back in
  – ObjectInputStream in;
  – obj = in.readObject();

• Must be of the same type
  – class and version
• Serializable Interface
  – By implementing this interface a class declares that it is willing to be read/written by automatic serialization machinery

• Automatic Writing
  – System knows how to recursively write out the state of an object
  – Recursively follows pointers and writes out those objects too!
  – Can handle most built in types
    • int, array, Point etc.

• “transient” keyword to mark a field that should not be serialized
  – Transient fields are returned as null on reading

• Override readObject() and writeObject() for customizations

• Versioning
  – Can detect version changes
Circularity: not an issue

- Serialization machinery will take circular references into account and do the right thing!
Dot example

• Build on DotPanel example!
• saveSerial(File f)
  – Given a file, write the data model to it with Java serialization.
  – Makes an Point[] array of points and writes it which avoids the bother of iteration.
    • We use an array instead of the ArrayList to avoid requiring a 1.2 VM to read the file, although maybe the ArrayList would have been fine
• loadSerial(File f)
  – Inverse of saveSerial.
  – Reads an Point[] array of Points, and adds them to our data model.
public void saveSerial(File file) {
    try {
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream(file));

        // Use the standard collection -> array util
        // (the Point[0] tells it what type of array to return)
        Point[] points = (Point[]) dots.toArray(new Point[0]);

        out.writeObject(points); // serialization!

        out.close(); // polite to close on the way out
        setDirty(false);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
private void loadSerial(File file) { 
    try {
        ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));

        // Read in the object -- the CT type should be exactly as it was written
        // -- Point[] in this case.
        // Transient fields would be null.
        Point[] points = (Point[]) in.readObject();
        for (int i=0; i<points.length; i++) {
            dots.add(points[i]);
        }

        in.close(); // polite to close on the way out
        setDirty(false);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
• **CS193J classes for serialization**
  
  - shield you from the exceptions, but otherwise behave like `ObjectOutputStream` and `ObjectInputStream`

```java
SimpleObjectWriter w;
SimpleObjectWriter w =
    SimpleObjectWriter.openFileForWriting(filename);
w.writeObject( <object>) -- write an array or object (Point[] in above example)
w.close()

SimpleObjectReader r;
SimpleObjectReader r =
    SimpleObjectReader.openFileForReading(filename);
obj = r.readObject() -- returns the object written -- cast to what it is
    (Point [] in above example)
r.close()
```
Summary

• Today
  – Repaint
    • Repaint Example
    • Erasing
  – Mouse Tracking
    • DotPanel Example
  – Advanced Drawing
    • Region based drawing, blinking, smart repaint
  – Object Serialization
    • Cloning and Serializing

• Assigned Work Reminder
  – HW 2: Java Draw
    • Due before midnight on Wednesday, July 23rd, 2003
    • Start early!!