Lecture 1

We discuss what to do when you want 'custom graphics', i.e. you want to draw something 'special' on the screen. We explain how Java 'paints' the screen, how painting is triggered, how to paint your own component, and how you can make explicit requests for painting.

We illustrate the ideas with a number of pie chart programs.

Painting a `JComponent`

- Swing draws ('paints') `JComponent`s onto their top-level containers
- Each `JComponent` has a `paint` method
- `paint` invokes 3 methods:
  - `paintComponent`, which draws the component itself (e.g. the button itself) (with the help of its UI delegate)
  - `paintBorder`, to add borders (if any)
  - `paintChildren`, which calls `paint` on any components that this component contains

The `paint` method has a parameter

- Every component in your GUI has an object of class `Graphics` associated with it
- This object is known as the *graphics context*. It hold info that facilitates painting: the current output device (e.g. the screen or an offscreen buffer), current drawing colour, current font, the clip rectangle (the area that need repainting)
- Whenever the system invokes a component's `paint` method, it supplies the component's graphics context as a parameter

When is `paint` invoked?

- *System-triggered painting:* the system software requests painting
  - when a component is first made visible
  - when a component is resized by the user
  - when a component has been 'damaged' and needs repair (e.g. after deiconification; e.g. after it stops being obscured by something else)
- *Application-triggered painting:* your program does something that causes painting
  - when a component's model is updated (e.g. after `setText` on a button)
  - when you explicitly invoke `repaint` (Never invoke `paint`)

How to do custom painting

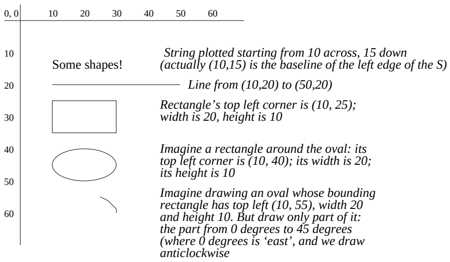
- Define a subclass of some `JComponent`
  - most likely it will be a subclass of `JPanel`
  - but, e.g., for a button with unusual appearance, define a subclass of `JButton`
- In your class definition, override `paintComponent` ('Never' override `paint`, `paintBorder` or `paintChildren`)
- 'Always' have the following as the first line in your definition of `paintComponent` (to get the background properly painted):

```
super.paintComponent(g);
```

- Then send messages to `g`, the graphics context, to get it to draw things for you

Methods of the `Graphics` object

```
g.drawString("Some shapes!", 10, 15);
g.drawLine(10, 20, 50, 20);
g.drawRect(10, 25, 20, 10);
g.drawOval(10, 40, 20, 10);
g.drawArc(10, 55, 20, 10, 0, 45);
```



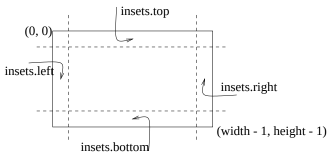
Lots more methods (incl. `fillRect`, `fillOval`, `fillArc`), and even more in Java 1.2

Pie chart

```
public PieChart extends JPanel
{
  public void paintComponent(Graphics g)
  { super.paintComponent(g);
    // Insert 5 statements from next slide
    // HERE
    chartWidth = (int) (availWidth *
      WIDTH_SCALER);
    chartHeight = (int) (availHeight *
      HEIGHT_SCALER);
    x = insets.left +
      ((availWidth - chartWidth) / 2);
    y = insets.top +
      ((availHeight - chartHeight) / 2);
    for (int i = 0; i < noOfSegments; i++)
    { g.setColor(colours[i]);
      g.fillArc(x, y, chartWidth,
        chartHeight, startAngles[i],
        segmentAngles[i]);
    }
 }
```


The painting area

If you draw near the edges, what you draw may be obscured by the borders:




```
int panelWidth = getSize().width;
int panelHeight = getSize().height;
Insets insets = getInsets(); // borders
int availWidth = panelWidth -
   insets.left - insets.right;
int availHeight = panelHeight -
   insets.top - insets.bottom;
```

E.g. to put a rectangle in the top left, don't draw it at (0, 0); draw it at `(0 + insets.left, 0 + insets.right)`

Explicit requests to `repaint`

- Suppose you want to force the system to redraw your component. Use `repaint`
- Why should you never invoke `paint`; why must you use `repaint`?
- 'All' requests to `paint` (system-triggered & application-triggered) are treated like events: the request goes onto the event queue; when the request eventually reaches the front of the queue, `paint` will be run
- Similarly, `repaint` puts a request onto the event queue
- If instead you called `paint`, `paint` would start executing now and might interfere with some execution of `paint` that had just come out of the queue

SelectablePieChart

```
public class SelectablePieChart
  extends PieChart
{ public void setRaisedSegment(
    int theRaisedSegment)
  { raisedSegment = theRaisedSegment;
    repaint();
  }
  public void paintComponent(Graphics g)
  { super.paintComponent(g);
    if (raisedSegment >= 0)
    { Color shadedColour =
        colours[raisedSegment];
      for (int i = 1; i < RAISE_AMOUNT; i++)
      { shadedColour = darken(shadedColour);
        g.setColor(shadedColour);
        g.fillArc(x, y - i, chartWidth,
          chartHeight,
          startAngles[raisedSegment],
          segmentAngles[raisedSegment]);
      }
    }
  }
```


PieSelector

The list's listener:

```
list.addListSelectionListener(
   new ListSelectionListener()
   {  public void valueChanged(
         ListSelectionEvent lse)
      {  if (lse.getValueIsAdjusting())
         {  // do nothing
         }
         if (list.isSelectionEmpty())
         {  pieChart.setRaisedSegment(-1);
         }
         else
         {  pieChart.setRaisedSegment(
               list.getSelectedIndex());
         }
      }
   });
```


Smarter painting

- There's a version of `repaint` that has 4 arguments specifying the rectangle that you want repainted. Calling this rather than `repaint()` can improve efficiency
- Your version of `paintComponent` mustn't involve heavy computation for the same reason that all event-handlers mustn't involve heavy computation. (People who put an animation loop inside `paintComponent` are wrong to do so)
- Swing has bugs still. If a component isn't changing size or position in the way you expect (and if you're *sure* it's not your fault!!), invoke `revalidate()`; if this still doesn't work, then also invoke `repaint()`; if this still doesn't work, then it probably was your fault after all