Java 2D

- Review of Abstract Windows Toolkit (*awt*)
- Java 2D graphical objects
- Java 2D rendering

Abstract Windows Toolkit (*awt*) Review of *import* files

- *awt* has always provided a *Graphics* class - almost obsolete
- It now also offers a *Graphics2D* class - much more versatile
- Don't use the old *Graphics* class except where you have to

- Abstract Windows Toolkit
  *import java.awt.**
- Event handler
  *import java.awt.event.**
- Print handler
  *import java.awt.print.**
- Java 2D geometry package
  *import java.awt.geom.**
- Java 2D image package
  *import java.awt.image.**
- Java 2D font package
  *import java.awt.font.**

Java 2D Graphical Objects

- Shapes
  - Lines, Closed Shapes, Paths, Areas
- Images
  - Buffers, Codecs
- Text
  - Fonts, Layouts, Transforms

Java 2D Graphical Objects - Shapes

- Shapes are defined by creating classes that implement the `Shape` interface
- A Shape is a list of components
- A component has 1 of 5 types and 0-3 points
  - *moveto* specifies a non-drawing movement (needs 1 point)
  - *lineto* specifies drawing a straight line (needs 1 point)
  - *quadto* specifies drawing a quadratic spline (needs 2 points)
  - *cubicto* specifies drawing a cubic spline (needs 3 points)
  - *close* specifies drawing a straight line back to the point given with the last *moveto*, thus closing the shape (needs 0 points)

Java 2D Graphical Objects - Shape Example

- `Line2D` is an abstract class which implements the `Shape` interface
- The `Shape` list has just 2 components in this case
  - *moveto* (10,30)
  - *lineto* (180,190)
- This will specify the line but won't draw it
- Creating shapes doesn't display them
  - Some kind of `draw()` method is needed for that

Java 2D Graphical Objects - Closed Shapes

- `Rectangle2D.*(x,y,w,h)`
  - Your standard rectangle [`(x,y)` = upper left corner]
- `RoundRectangle2D.*(x,y,w,h,aw,ah)`
  - Rectangle with round corners
  - Arc used to round off [`aw,ah` are arc width, height]
- `Ellipse2D.*(x,y,w,h)`
  - An ellipse [defined by its bounding rectangle]
  - `w=h` gives a circle
- `GeneralPath()`

`*` = `Float` or `Double`

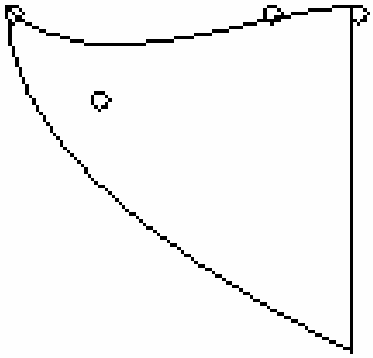
Java 2D Graphical Objects - Defining Shapes with Paths

```
public void paint (Graphics g)
{
   Graphics2D g2d = (Graphics2D)g;
   GeneralPath pathShape = new GeneralPath();
   pathShape.moveTo(150,100);
   pathShape.lineTo(150,150);
   pathShape.quadTo(50,100,50,50);
   pathShape.curveTo(75,75,125,50,150,50);
   pathShape.close();
   g2d.draw(pathShape);
}
```

Java 2D Graphical Objects - Iterating through Paths

```
PathIterator segmentList =
  pathShape.getPathIterator(null);
double[] coords = new double[6];
while (!segmentList.isDone())
{
   // Extract type of segment and coordinates
   int segmentType =
     segmentList.currentSegment(coords);
   // Do stuff with this segment ...
   segmentList.next();
}
```

Path Example



Java 2D Graphical Objects - Areas

- Java 2D provides an `Area` class which can be used to create shapes upon which boolean operations can be performed

	Boolean	**Set**
add()	OR	UNION
subtract()	AND … NOT	DIFFERENCE
intersect()	AND	INTERSECTION
exclusiveOr()	XOR	UNION minus INTERSECTION

- `Area` objects are created from `Shape` objects
  - The `Shape` object MUST BE CLOSED

Java 2D Graphical Objects - Area Example

- Consider 4 areas derived from ellipses

```
Area[] ellipse = new Area[4];
ellipse[0]= new Area(new Ellipse2D.Double(0,20,30,20);
ellipse[1]= new Area(new Ellipse2D.Double(30,20,30,20);
ellipse[2]= new Area(new Ellipse2D.Double(20,0,20,30);
ellipse[3]= new Area(new Ellipse2D.Double(20,30,20,30);
```

- Union ellipse[0] with ellipse[1] and ellipse[2] with ellipse[3] leaving results in ellipse[0] and ellipse[2] respectively

```
ellipse[0].add(ellipse[1]); ellipse[2].add(ellipse[3]);
```

- What will the following leave in ellipse[0]?

```
ellipse[0].exclusiveOr(ellipse[2]);
```

Java 2D Graphical Objects - Images

- Shapes are defined mathematically
  - Each segment type in a path uses a formula
  - They are an example of *vector graphics*
- Images are defined by pixel arrays
  - They are an example of *raster graphics*
  - Images can be created and manipulated in buffers

```
BufferedImage thisImage =
 new BufferedImage(xDimension, yDimension,
                   imageType);
```

Java 2D Graphical Objects - JPEG Image Example

- Class `Jpeg` creates a JPEG file using the JPEG codec

```
import com.sun.image.codec.jpeg.*;
```

- An image buffer is defined

```
BufferedImage jpegImage =
new BufferedImage(200,200,BufferedImage.TYPE_INT_RGB);
```

- A *Graphics2D* object is created for the image

```
Graphics2D g2d = jpegImage.createGraphics();
```

- The JPEG codec does the work

```
JPEGImageEncoder encoder =
JPEGCodec.createJPEGEncoder(outFile);
encoder.encode(jpegImage);
```

Java 2D Graphical Objects - Text

- Fonts are defined with the `Font` class

```
Font font = new Font("SanSerif", Font.BOLD, 34);
```

- Fonts vary between devices
  - `FontRenderContext` describes the device

```
FontRenderContext fontRenderContext =
  g2d.getFontRenderContext();
```

- The `TextLayout` class specifies the text

```
TextLayout text = new TextLayout (string, font,
                                  fontRenderContext);
```

- `TextLayout` has its own `draw` method

```
text.draw(g2d, x, y);
```

Java 2D Graphical Objects - Text Transform Example

- To apply *Graphics2D* transformations we turn the text into a `Shape`

```
Shape textShape = text.getOutline(null);
```

- The `null` parameter refers to an affine transform
  - By default the text will be located at (0,0)
  - To locate it at (x,y) we can use a non-null transform

```
Shape textShape =
text.getOutline(AffineTransform.getTranslateInstance(x,y))
```

- Since the text is now a `Shape` we can draw it thus

```
g2d.draw(textShape);
```

Java 2D Graphical Objects - Affine Transformations

- Invariants of affine transformations
  1. Curvature - straight lines cannot become curved
  2. Parallelism - parallel lines remain parallel
- *Graphics2D* objects and affine transformations

```
// Create a shear transform
AffineTransform affTransform =
  AffineTransform.getShearInstance(0.5, 0);
//Post-multiply by a translation (so shear done last)
affTransform.translate(20, 20);
// Apply the transform to object -g2d-
g2d.setTransform(affTransform);
// Draw the shape -aShape- (not defined here)
g2d.draw(aShape);
```

Java 2D Rendering

- Colour
  - Fixed colours, Transparency, Composition
- Filling
  - Solid fills, Gradient fills, Textured fills
- Line Styles
  - Strokes, Caps, Mitres, Dashes
- Clipping

Java 2D Rendering - Colours

- Colour is set with the `setPaint()` method

```
g2d.setPaint(Color.black);
```

- The complete palette of fixed colours is:
  - `black, blue, cyan, darkgray, gray, green, lightgray, magenta, orange, pink, red, white, yellow`

Java 2D Rendering - Transparency

- We can define more colours by specifying their Red, Green and Blue (RGB) components

```
Color deepPurple = new Color(0.5f,0.0f,0.5f);
```

- We can also add a fourth parameter called the *alpha* value to indicate the degree of opacity

```
Color seethruPurple = new Color(0.5f,0.0f,0.5f,0.5f);
```

- This is the RGBA colour model
  - If $\alpha = 0.0$ the colour is completely transparent
  - If $\alpha = 1.0$ the colour is completely opaque (default)

Java 2D Rendering - Composition

- *Composition* determines what to display when different colours are drawn on top of each other
- Composition algorithms use the concepts of a *destination* and a *source*
  - Destination (*d*) is what has already been drawn
  - Source (*s*) is what is about to be added to it
- Colour to be displayed and its alpha are given by

$$c_d := f_s \alpha_s c_s + f_d \alpha_d c_d \qquad \text{[for each of R, G \& B]}$$

$$\alpha_d := f_s \alpha_s + f_d \alpha_d \qquad \text{[for each of R, G \& B]}$$

- Further draws continue to update c_d and a_d

Java 2D Rendering
- Source-over Composition Rule

- Various composition algorithms exist but the most common is the *source-over rule*
  - The object currently being drawn (the source) is placed over any objects already drawn (the destination)
  - In the source over rule (`SrcOver` in Java)

    $f_s = 1$ *and* $f_d = 1-\alpha_s$ [for each of R, G & B]

    So,

    $c_d := \alpha_s c_s + (1- \alpha_s) \alpha_d c_d$ [for each of R, G & B]

    $\alpha_d := \alpha_s + (1- \alpha_s) \alpha_d$ [for each of R, G & B]

```
g2d.setComposite(AlphaComposite.SrcOver);
```

Java 2D Rendering
- Other composition rules

Rule	**Java static**	f_s	f_d
Source-over	`SrcOver`	*1*	$1- \alpha_s$
Source	`Src`	*1*	*0*
Source-in	`SrcIn`	α_d	*0*
Source-out	`SrcOut`	$1- \alpha_d$	*0*
Destination-over	`DstOver`	$1- \alpha_d$	*1*
Destination-in	`DstIn`	0	α_s
Destination-out	`DstOut`	0	$1- \alpha_s$
Clear	`Clear`	0	0

Java 2D Rendering - Solid Fills

- Solid Fills

```
//Set the fill colour
g2d.setPaint(Color.green);
// Fill the shape -aShape-
g2d.fill(aShape);
// Make sure border drawn on top of filled
  area
g2d.setPaint(Color.black);
g2d.draw(aShape);
```

Java 2D Rendering - Gradient Fills

- Filling with gradient patterns

```
Paint pattern = new GradientPaint(x1,y1, Color.blue,
  x2,y2, Color.yellow);
g2d.setPaint(pattern);
g2d.fill(aShape);

(x1,y1)  start point for gradient - pure blue
(x2,y2)  end point of gradient - pure yellow
(x,y)    in general some mix of blue/yellow
```

Java 2D Rendering - Textured Fills

- Filling with textured patterns
  - A `BufferedImage` is used to define the texture

```
private BufferedImage textureConstructor()
{
   BufferedImage texcha = new BufferedImage(…);
   .
   .
   .
   return texcha;
}
```

  - We apply our method with `TexturePaint()`

```
Paint pattern = new
  TexturePaint(textureConstructor());
```

Java 2D Rendering - Line Styles I

- Line styles are supported by the `Stroke` interface and its implementing class `BasicStroke`

```
BasicStroke stroke;
stroke = new BasicStroke(width, capStyle, joinStyle);
g2d.setStroke(stroke);
```

- The *cap* styles are the shapes of the ends of lines
  - `BasicStroke.CAP_SQUARE`
    - Extends the line by an extra square
  - `BasicStroke.CAP_BUTT`
    - Squares off the end of the line but doesn't extend it
  - `BasicStroke.CAP_ROUND`
    - Extends with a semi-circle to round end off

Java 2D Rendering - Line Styles II

- The *join* styles determine how lines are connected to each other
  - `BasicStroke.JOIN_BEVEL`
    - Joins the outside edges of lines with a straight line giving blunt corner
  - `BasicStroke.JOIN_MITER`
    - Extends the outside edges of lines until they meet giving sharp corner
    - Note spelling: MITER (US), not MITRE (UK)
    - Optionally a mitre limit parameter can be provided which limits how far a corner can be extended in the construction of the mitred joint
  - `BasicStroke.JOIN_ROUND`
    - Caps corners with circular segments to give rounded corner

Java 2D Rendering - Line Styles III

- Dashed lines are created by specifying templates in arrays

```
float[] dashPattern = {10,5,5,5};
```

  - This sets up pattern which draws a line 10 pixels long, leaves a gap 5 pixels long, draws a line 5 pixels long and then leaves another gap 5 pixels long (then repeats)
  - A *dash phase* (float!) specifies where in the array the pattern should start (0.0f means start with first entry in array)

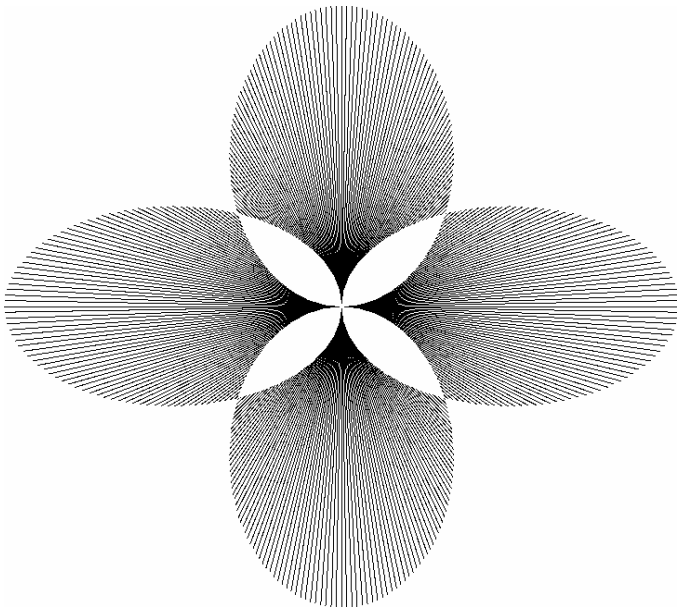
```
BasicStroke stroke;
stroke = new BasicStroke(width, capStyle, joinStyle,
  miterLimit, dashPattern, dashPhase);
```

Java 2D Rendering - Clipping

- Clipping is achieved in Java 2D by creating a shape whose boundary determines what should and should not be drawn
  - Anything outside the shape's boundary is clipped away and not displayed
  - Clipping with an ellipse for example

```
Shape clippingShape =
      new Ellipse2D.Double(30,70,200,100);
g2d.setClip(clippingShape);
```

Petzold's Clover Leaf



Java 2D Exercise

- Develop a program to
  - Display your name in a colourful and interesting way
    - I.e. apply some transformations to it
  - Draw a fancy border around it
    - I.e. construct a path or three around it
  - E-mail the output to me (nkt@macs.hw.ac.uk)
    - I.e. save it as a JPEG file
- Before the next lecture please