CS416

Introduction to Computer Science II

Spring 2018

2 AWT Graphics

- Previously, in 415
  - We used the *wheelsunh* package for graphics
  - *wheelsunh* hid a lot of the grubby detail of the “real” graphics interface: Swing and AWT

Preview

- Time to ditch the training “wheels” and ride on our own using
  - AWT - the original Java 2D graphics support library
  - Swing - enhanced set of classes built on AWT

Abstract Window Toolkit

- AWT was part of the first major Java release (1.0)
- A set of object-oriented classes that makes GUI programming easier and <u>platform independent</u>
- Supports
  - UI components - windows, buttons, sliders, menus, scrollbars, and more
  - 2D graphics - shapes
  - event handling - mouse and keyboard interaction
- Its low-level and complex

Swing

- Swing appeared in Java 1.2
- Built on top of AWT
- More flexible, more powerful, easier to use
  - but still pretty complex
- Replaces some, but not all of the AWT functionality
  - GUI applications typically use both

Java2D Toolkit

- The Java2D toolkit appeared in Java 1.2
- Expanded graphical functionality of AWT
  - 2D Shapes, affine transformations
  - image manipulation, text, and more
- defined in java.awt.geom

wheelsunh FirstApp

```
public class FirstApp extends wheelsunh.users.Frame
{
   //------------ instance variables ----------
   private wheels.users.Ellipse circle;

   //------------ constructor -------------------
   public FirstApp()
   {
       circle = new wheels.users.Ellipse();
   }

   //------------- main ---------------------
   public static void main(String[] args)
   {
       // create an instance of FirstApp
       FirstApp app = new FirstApp();
   }
}
```

Creates a window with a border and buttons; an area to draw into (a panel); and a Quit button.

Creates an ellipse and draws it in the drawing panel.

From *wheelsunh* to Swing

- *wheelsunh* does a lot for us and it hides a lot
  - creates a window, hides the drawing panel
  - re-draws the image after window is obscured
    - cover your drawing with another window then uncover it
    - how did the ellipse get re-displayed?
  - handles many event details for us
  - and more
- Underneath *wheelsunh* is the “real” AWT/Swing code

Basic Swing Windows

- *JFrame* class creates and manages windows
  - provides the outside frame of the window, the application decides what goes inside it
  - extends AWT *Frame*
    - adds more flexible *look-and-feel* properties
- *JPanel* class instance goes into a *JFrame*
  - <u>Can't</u> draw on a frame; but <u>can</u> draw on a panel (like a physical window frame that contains a window pane upon which you can draw)

Swing Application Template

```
public class SwingApp extends JFrame
{
  //------ Constructor ---------------
  public SwingApp( String title )
  {
    super( title );
    this.setSize( 700, 500 );
    this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    // add a JPanel here
    this.setVisible( true );
  }

  //-------- main ----------------------
  public static void main(String[] args)
  {
     SwingApp app = new SwingApp( "A Swing Application" );
  }
}
```

Wheels did all this for us

If we encapsulate the application-specific code in a class that extends JPanel, this line is the only one that is not "boilerplate".

A Drawing Panel

```
public class DrawPanel extends JPanel
{
  //------ instance variables for contents of panel -------
  // declare instance variables for graphical objects

  //------ Constructor ------------
  public DrawPanel()
  {
    super();
    this.setBackground( Color.GRAY );
    // add creation of graphical objects, such as an ellipse
  }
  //--------- paintComponent( Graphics ) ----------------
  public void paintComponent( Graphics aBrush )
  {
    super.paintComponent( aBrush )
    // add code here to draw each object on the panel
  }
}
```

Application-specific code to define graphical objects and display them.

paintComponent is called from the Java environment whenever the panel's contents needs to be re-painted.

A *Graphics* object describes the graphical environment in which the objects are to be displayed and it knows how to display AWT graphical objects

Swing Application

```
public class SwingApp extends JFrame
{
  //------ Constructor ----------------
  public SwingApp( String title )
  {
    super( title );
    this.setSize( 700, 500 );
    this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    this.add( new DrawPanel() );
    this.setVisible( true );
  }

  //-------- main ----------------------
  public static void main(String[] args)
  {
     SwingApp app = new SwingApp( “A Swing Application” );
  }
}
```

Create the application’s drawing panel and add it to the frame.

DrawPanel: Shape creation

```
public class DrawPanel extends JPanel
{
  //------ instance variables for contents of panel -------
   private AEllipse ball;

  //------ Constructor ------------
  public DrawPanel()
  {
    super();
    ball = new AEllipse( java.awt.Color.RED );
    ball.setLocation( 75, 75 );
    ball.setSize( 60, 60 );
   }
  //--------- paintComponent( Graphics ) --------------
  public void paintComponent( Graphics aBrush )
  {
      super.paintComponent( aBrush );
      // add code here to draw each object on the panel
   }
}
```

The book uses a *SmartEllipse*. To make conversion from *wheelsunh* easier, we created a similar class, but named it *AEllipse*.

This looks just like *wheelsunh*, but if we end there, we'll get a blank screen!

DrawPanel: Shape display

```
public class DrawPanel extends JPanel
{
  //------ instance variables for contents of panel -------
   private Ellipse ball;

  //------ Constructor ------------
  public DrawPanel()
  {
    super();
    ball = new Ellipse( java.awt.Color.RED );
    ball.setLocation( 75, 75 );
    ball.setSize( 60, 60 );
  }
  //--------- paintComponent( Graphics ) ---------------
   public void paintComponent( Graphics aBrush )
   {
     super.paintComponent( aBrush );
     Graphics2D brush2D = (Graphics2D) aBrush;
     ball.fill( brush2D );
     ball.draw( brush2D );
   }
}
```

Java2D graphical objects need to be drawn with a *Graphics2D* context. *Graphics2D* extends *Graphics*, and the actual argument is always a *Graphics2D*, so the coercion (cast) works.

Need to *fill* the interior and *draw* the border. These methods need a *Graphics2D* object.

SmartEllipse SmartRectangle

- *SmartEllipse* (from the book) -- extends an AWT class to look a lot like the *wheelsunh Ellipse* class.
- *SmartRectangle* (from the book) -- extends an AWT class to look a lot like the *wheelsunh Rectangle* class.

New *wheelsunh*-like classes

- We’ve built new *Ellipse*, *Rectangle*, and *Line* classes that extend AWT classes.
- *AEllipse* is very similar to *SmartEllipse*
- *ARectangle* is very similar to *SmartRectangle*
- These classes "wrap" AWT classes in more convenient packages:
  - Better *encapsulation*: the color of a graphical object is part of the object
  - A different *interface*: one similar to *wheels*

The new *Ellipse* class

```
public class AEllipse extends java.awt.geom.Ellipse2D.Double
{
   ----- instance variable declarations ------
   public AEllipse( Color c ){ setColor( c ); }
   public void setFrameColor( Color c ) { _bCol = c; }

   ----- more wheels-like accessors and mutators ----------

   public int getXLocation(){ . . . }
   public int getYLocation(){ . . . }
   public void setLocation( int x, int y )
   {
      . . .;
   }
   public void setSize (int aWidth, int aHeight)
   {
      . . .;
   }
   public void fill( java.awt.Graphics2D newBrush ){ . . . }
   public void draw( java.awt.Graphics2D newBrush ){ . . . }
}
```

Most methods implement *wheels* interface methods

These last two are not *wheels*, but AWT methods

AWT *Ellipse2D* class

- *SmartEllipse* and the new *AEllipse* both extend *java.awt.Ellipse2D.Double*
- *Ellipse2D* is an abstract class, but it has two inner *static* classes that are concrete and public and they extend *Ellipse2D*. (Sounds pretty weird, huh?)

Ellipse2D.Double

```
public abstract class Ellipse2D
{
  . . .
  public boolean contains( double x, double y ) { ... }
  public static class Double extends Ellipse2D
  {
    . . .
    public void setFrame( double x, . . . ){ ... }
  }
}
// Application code:
  Ellipse2D.Double e = new Ellipse2D.Double( . . . );
  e.setFrame( 100, 100, 50, 60 );
  if ( e.contains( x, y ))
     . . .
```

The inner class extends its containing class!

Double must be *static* so it can be accessed without an Ellipse2D object

Since *Double* is a *static* inner class, it is referenced by the *Ellipse2D* class name, rather than an instance of the Ellipse2D class -- which can't exist since it is an abstract class.

Since *Double* extends *Ellipse2D*, this is a valid method call.

Ellipse2D.Double

- *Ellipse2D.Double* and its pal *Ellipse2D.Float*
  - extend *Ellipse* and thus *RectangularShape*
  - hence, location and size are set using the *setFrame* method of *RectangularShape:*
    - *void setFrame(double x, double y, double width, double height);*
- We want to translate *wheelsunh*-like methods to the *setFrame* interface in the new *Ellipse* class

Location/size methods

```
public class AEllipse extends java.awt.geom.Ellipse2D.Double
{
  . . . .
  public int getXLocation(){ return (int) this.getX(); }
  public int getYLocation(){ return (int) this.getY(); }

  public void setLocation( int x, int y )
  {
    this.setFrame( x, y, this.getWidth(), this.getHeight() );
  }
  public void setSize( int w, int h )
  {
    this.setFrame( this.getX(), this.getY(), w, h );
  }
  . . .
}
```

AEllipse paint methods

```
public class AEllipse extends java.awt.geom.Ellipse2D.Double
{
   . . .
   public void fill( java.awt.Graphics2D brush2D )
   {
      Color savedColor = brush2D.getColor();// save brush color
      brush2D.setColor( _fillColor );      // set brush to fill color
      brush2D.fill( this );                // fill this ellipse
      brush2D.setColor( savedColor );      // restore original color
   }
   public void draw( java.awt.Graphics2D brush2D )
   {
      Color savedColor = brush2D.getColor(); // save color
      brush2D.setColor( _borderColor );      // set to border color
      java.awt.Stroke savedStroke = brush2D.getStroke(); // line info
      brush2D.setStroke( new java.awt.BasicStroke( _lineWidth ));
      brush2D.draw( this );
      brush2D.setStroke( savedStroke );
      brush2D.setColor( savedColor );
   }
}
```

fill is called to fill the interior of RectangularShape objects

draw is called to draw the border of RectangularShapes

The *Graphics2D* object Color and Stroke fields; saving/restoring them guarantees that these methods have no *side effects*. This is good software engineering practice.

Composite Objects

- What if we want a composite object (like a Robot)?
- *Robot* needs a data member for each component.
- Its constructor creates the components
- It needs *draw* and *fill* methods that in turn will call the *draw* and *fill* methods of each of its components.
- Need the *DrawPanel* to create the *Robot* objects and call their *draw* and *fill* methods in its *paintComponent*
- Our *A*-classes have a *display* “convenience” method that calls *fill* then *draw*; we do that for composites, too

Composite *A-wheels* Object Template

```
public class AGroup implements AShape
{
  private Vector<AShape> shapes;
  . . . .
  public AGroup( ... )
  {
    shapes = new Vector<AShape>();
    AEllipse ae = new AEllipse( ... );
    ... // set attributes
    shapes.add( ae );
    // create more AEllipse, ARectangle, ALine
    // add to the Vector
  }
  public void display( Graphics2D g2 )
  {
    for ( int i = 0; i < shapes.length; i++ )
      shapes.get( i ).display( g2 );
  }
}
```

Need *AShape* interface for the *display* method

Vector or *ArrayList* of all A-objects that need to be "painted".

Add each graphical object to the *shapes* collection

The composite's *display* needs to be called by some object's *paintComponent* -- usually a *JPanel*

Here, we "paint" them, by calling their "display" method

Other *Graphics* methods

- *Graphics/Graphics2D* objects can draw images, polylines, polygons, rounded rectangles, arcs, text, etc.
- Polygon
  - A closed region bounded by connected lines. A Polygon object can be created with it.
      - array of x values, array of y values, and a count.
- Graphics.fillPolygon and Graphics.drawPolygon
    - take arrays <u>or</u> Polygon object. Array example:
    - g2.drawPolygon(_drawX, _drawY, drawX.length);
    - where _drawX and drawY are int[]

Review

- We want more freedom and control
  - We’re using Swing to manage windows
  - We’re building on AWT for drawing
- We now have to deal with some new details
  - We need to know about *JFrame* and *JPanel*
  - We need our objects to know when to re-paint themselves
- We have new *AEllipse*, *ARectangle* and *ALine* classes to simplify transition

Next, in 416

- More *JComponent* features
- J-objects
    - wheels-like classes such that each object is a *JComponent*