Drawing and Painting in Java
IAT351

Week 5 Lecture 1

1.10.2012

Lyn Bartram

lyn@sfu.ca

Today

- Administrivia
- Building custom objects (assignment 2)
- Graphics
  - When / Why ?
  - Painting
  - What can be done
  - Java 2D

Slides adapted from Tom Lodge

Administrivia

- Assignment 2 out
- Submission issues:
  - Lots of IAT351-Assignment 1
  - “Ass1”
  - NO NAMES?

Assignment 2

- When you have finished this assignment you will have gained:
  - Experience with the Swing drawing pipeline.
  - experience with Swing layout managers and custom component architecture
  - Experience writing a variety of input listeners.

Making custom objects

- Assignment 2: develop your own components
- Key software development principle:
  - Reduce (don't do new work unless you have to)
  - Re-use (extend from existing tools and code)
  - Recycle (adapt to new uses by adding and refining new code)

- DON'T PUT everything in one class with static methods
- DO CHOOSE the appropriate class to extend

Making custom objects: basic steps

1. Choose the object you want to build from at the appropriate level of abstraction
   - Example: `JAbstractButton` or `JMenuItem`?
2. Extend from parent
   - `myTagger extends JAbstractButton`
3. Initialise
   - `super()` – Use parent's existing code
4. Over-ride
   - Replace existing routines or stubs with custom behaviour

Making custom objects

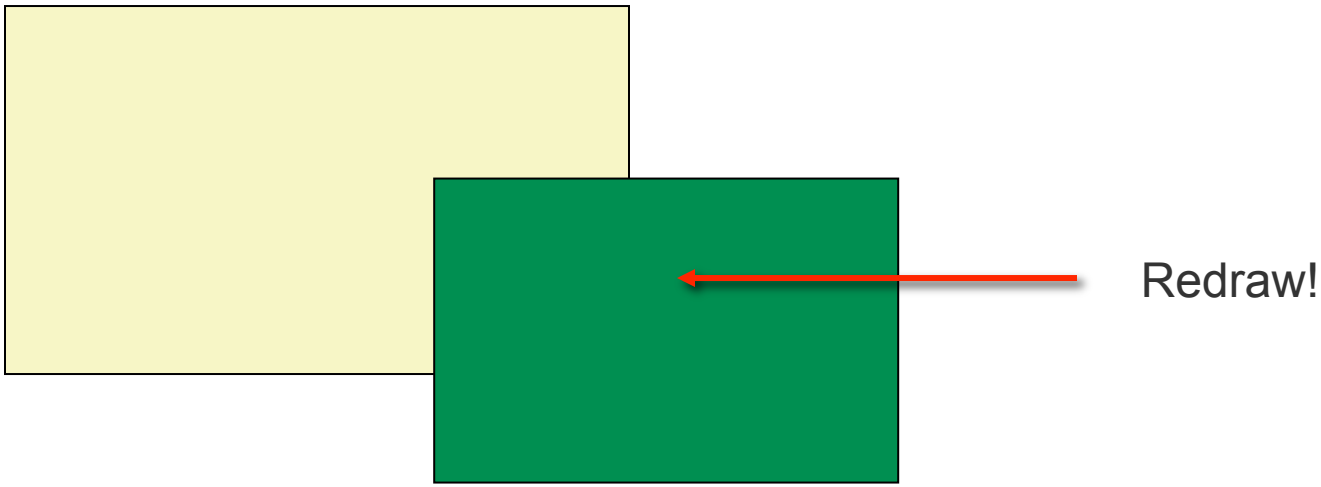
- In the user interface, what typically needs to be customised ?
  - How object looks
  - How object behaves
  - What kinds of interactions it accepts

Making custom objects

- In the user interface, what typically needs to be customised ?
  - How object looks (GRAPHICS)
  - Each object in Swing responsible for knowing how to render itself
  - We're going to learn how to extend it

Example: managing windows

- Windows suffer "damage" when they are obscured and then re-exposed
  - resized



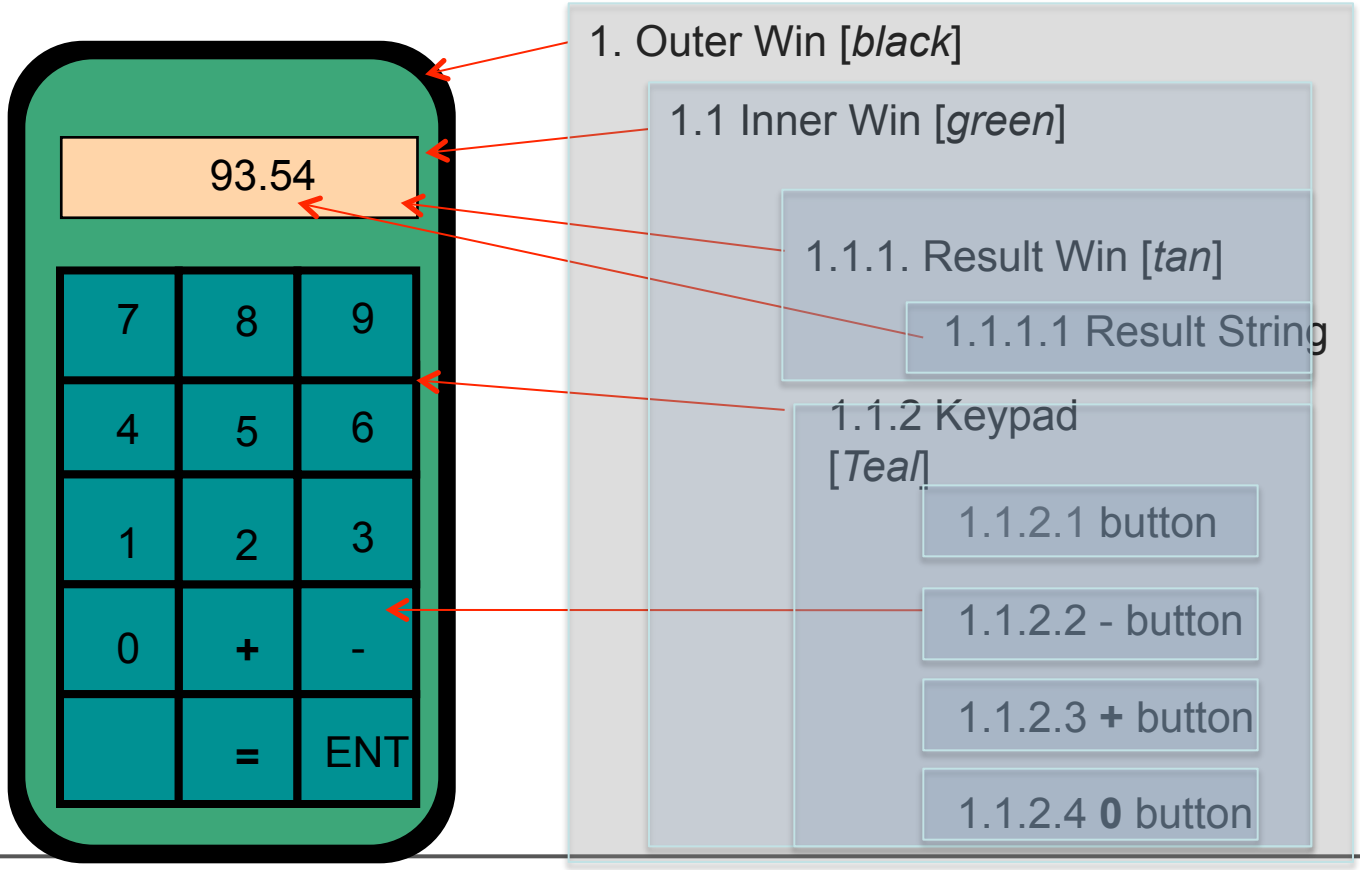
Issue: damage/redraw

- How much is exposed?
- System may or may not maintain and/or restore obscured portions of windows
- Have to be prepared to redraw anyway since larger windows create a new content area

Displaying objects

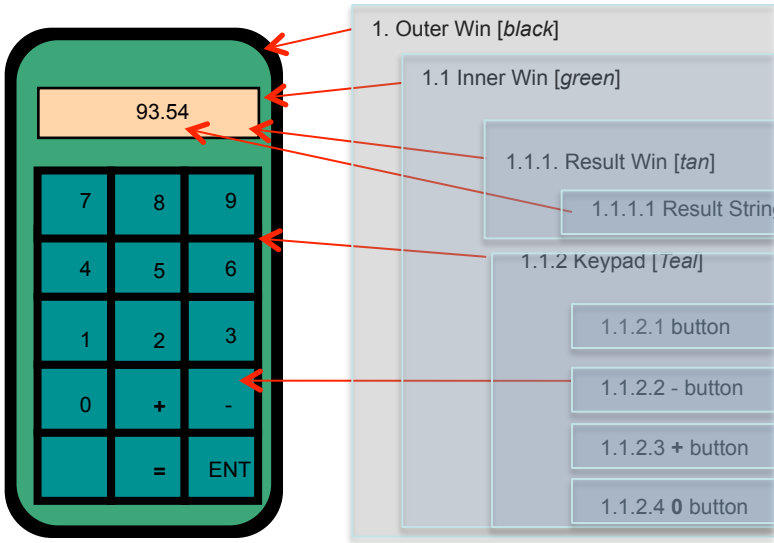
- output is organised around the **interactor tree** structure
  - Each object knows how to draw and do tasks specific to what it is
  - Each object knows what children it has and what their capabilities are
  - Generic tasks specialised to specific subclasses

Interactor tree



Damage

- each object reports its own damage
  - Tells parent, which tells parent, up the tree
- WM collects all the damaged regions at the top
  - determines which actually need to be redrawn
  - Normally one enclosing rectangle

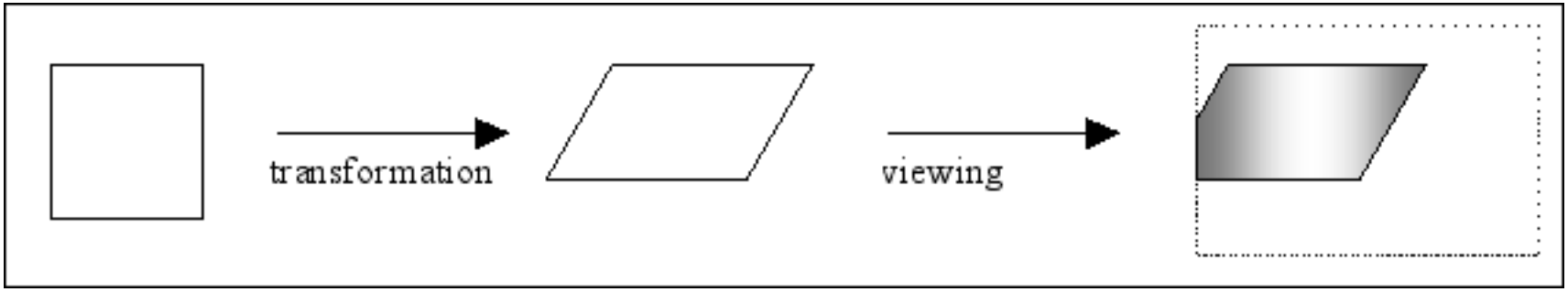


Redraw

- Interactors draw on a certain screen area
- When screen image changes, need to schedule a redraw
- 2 kinds of updates in Swing
  - Layout changes (a component may be added or deleted)
  - State changes, drawing updates

Basic Graphics Operations

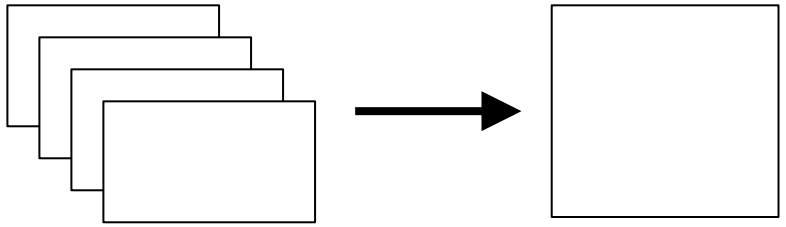
1. Construct the 2D objects.
2. Apply transformations to the objects.
3. Apply color and other rendering properties.
4. Render the scene on a graphics device.



Drawing

- Draw and redraw and undraw operations
  - Rubber banding a line over complex background
- Once you have drawn a shape on the display, how do you undraw?
  - Redraw everything under, then redraw previous state (expensive)
  - “Save-under”: restore previous state - issues?
  - Use XOR bit manipulation of colours (X^B^B == A)
  - Simulate bit planes with CLUT tricks
    - Used for special cases like cursors

Managing drawing



- Typically don't want to "just draw it"
- optimise redraw
- Add to **display list**/redraw list
  - Series of commands that defines the image
  - Executed when called
  - "retained mode" system
- `repaint()` adds the command to the list

Object-oriented abstractions for drawing

- Most modern systems provide uniform access to all graphical output capabilities and devices
- Abstraction provides set of drawing primitives
- Hide low- level details
  - Graphics/bitmap operations
  - Device dependence
- Might be drawing on
  - Window, direct-to-screen, in-memory bitmap, printer
- Key point is that you write code that does not have to know which one

Object-oriented abstractions for drawing

- Generally don't want to depend on details of device but sometimes need to know some aspects (limitations)
  - Size
  - Can it be resized, and how much?
  - Colour depth
  - Pixel resolution (for fine details)
- As a UI designer, you need to be sensitive to properties of each one and context of use, e.g.
  - Visibility (a mobile backlit screen)
  - Position (peripheral vs central)

Why do I need to code my own graphics?

- May need graphics if
  - Wish to drastically change appearance of an existing component
  - Draw shapes to screen
  - Draw text to screen
  - Animation
  - “Invent” / customise your own component

Component Painting (1)

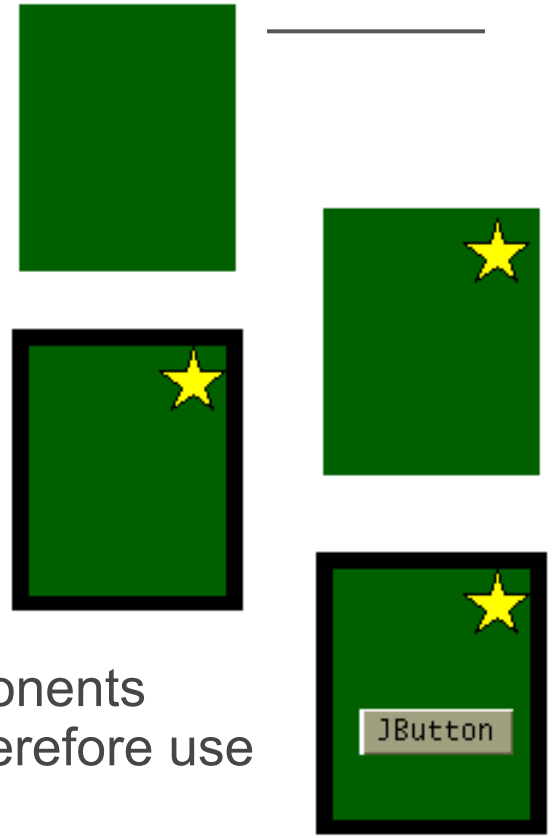
- **`repaint()`**
  - (sometimes automatically) called when necessary. E.g.Window unhidden
- **`revalidate()`**
  - (automatically) called before repaint when needed. E.g. when component sizes or position have changed
- painting on the event-dispatching thread
- **`repaint()`** and **`revalidate()`** are thread safe

Painting (2)

- Double buffering
  - Painting performed to off screen buffer, then flushed to screen when finished.
- Opaque components
    - Improves performance. Time not spent painting behind components.
- Shape of painting areas
  - always rectangular
  - non-opaque (transparent) components can appear any shape, although need to use glass panes to mask hit detection area

Painting order

- Visibility based on containment hierarchy
  - Background (if opaque)
  - Custom painting
  - Border
  - Child components

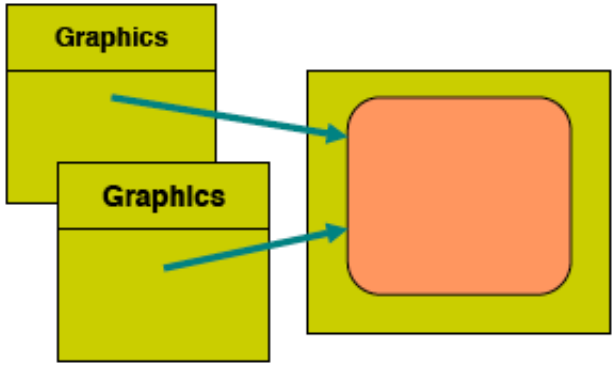


- repaints
  - if transparent component repaints, all components under it must also repaint. This is costly, therefore use opaque when possible

What graphics tools are available in Swing?

- Java AWT Graphics
  - Basic but usually enough
  - Shapes
  - Text
  - Images
- Java 2D graphics
  - Extends the AWT graphics to provide much more, eg gradients, textures etc

java.awt.Graphics[2D]



- indirect access to drawing surface (device)
  - Drawing state (graphics context)
    - Current clipping rectangle (what area to draw)
    - Colours of objects
    - Font
    - Double buffering..
  - Multiple Graphics instances may reference the same underlying drawing surface but hold different state information
- Think of it like a set of “brushes”

A Simple Java 2D Program

- Usually have at least two classes
  - A class to serve as the high level container
    - The main method needs to create a **`JFrame`** for a stand-alone application
  - A class that extends **`JPanel`** to serve as the drawing surface
    - The **`paintComponent`** method contains code to do the drawing

paintComponent()

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D)g;

   // code to draw desired picture

 }
```

Graphics class

- Original AWT classes had a graphics context which was a Graphics object
  - No separation of modeling and rendering
- Swing classes have a Graphics2D context
  - Parameter of the paintComponent method still has type Graphics
  - The actual argument to paintComponent is a Graphics2D object so it can be cast

The `Graphics2D` Class

- Java 2D rendering engine
- Extends the Graphics class
- Encapsulates all rendering functions
- Two ways to access the graphics context

```
void paintComponent(Graphics g){

}

Graphics getGraphics()
```

Methods of Graphics Class

- Data includes the following
  - Foreground color
  - Font
  - Rendering modes

```
void setColor(Color c)
void setFont(Font f)
void setXORMode(Color c)
void setPaintMode()
```

Custom painting (1)

- Occurs between background and border
- Extend component and customise
  - JPanel recommended e.g. class `myPaintingPanel extends JPanel {`
  - can also use atomic components
    - e.g. class `myPaintingButton extends JButton {`
- Code goes in overridden `paintComponent()` method
  - public void `paintComponent(Graphics g) {`

Custom painting (2)

- Methods involved:
- `Public void paint(Graphics g)`
  - AWT
  - You will typically not use it
- `public void paintBorder(Graphics g)`
- `public void paintChildren(Graphics g)`
- `public void paintComponent(Graphics g)`
  - **This is where you write your code**

Custom painting

- Here's the kicker: you NEVER call these routines directly!
- Remember: an object "knows how to redraw itself"
  - You are adding to that knowledge : **`paintComponent()`**
  - You call it indirectly and let it render itself
  - `repaint() , revalidate()`

Custom painting: the call process

- Whenever you want to execute the code in **`paintComponent(…)`**, call **`repaint()`** on the object (`drawingPanel`) instead

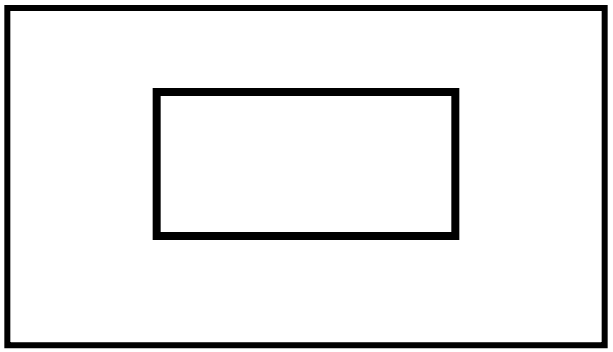
```
drawingPanel.repaint();
```

- Java calls **`drawingPanel`**'s **`paintComponent(…)`** for you and also creates the context **`(Graphics2D)`** for you.
- *Summary*: To make a JPanel do something useful, create a subclass of it and *augment* ("partially override") **`paintComponent()`** to call **`paint()`** on the shapes in your panel

Picasso calls repaint()!

Repaint creates a Graphics2D and calls `paintComponent(…)`

`repaint()`



DrawingPanel

(In `DrawingPanel`)

```
paintComponent(Graphics g){
  super.paintComponent(g);
  Graphics2D brush = (Graphics2D)g;
  Myrectangle.paintMe(brush);
}
```

(In `Myrectangle`)

- a user defined object

```
paintMe(Graphics2D brush) {
  brush.setColor(_borderColor);
  brush.draw(_shape);//_shape is a Rectangle2D
  brush.setColor(_fillColor);
  brush.fill(_shape);
}
```

Repainting custom paints : recap

- Calling repaint() method *requests a* repaint to occur
  - **`repaint()`**
    - paints whole component & any others that need it if transparent
  - **`repaint(int leftx,int lefty,int width,int height);`**
    - only repaints a specified area
- Component painted after all pending events dispatched.
- Painting should be kept short - Done in event thread - slows GUI

Custom painting

- When writing custom painting code
  - should not reproduce painting functionality of any other Swing component

- Before doing anything else in **`paintComponent()`**
  - invoke **`super.paintComponent()`** **or**
  - **`setOpaque(false)`**

Painting coordinates

- Component sizes in pixels
  - (0,0) at top left corner
  - (width-1,height-1) at bottom right corner
- Border sizes (e.g. a 1-pixel border)
  - (1,1) at top left
  - (width-2, height-2) at bottom right
- General painting area for b pixel border
  - [{b,b} : {w-(b+1), h-(b+1)}]

Getting painting coordinates

- Use component getWidth() and getHeight()
- Get border sizes with getInsets()
  - e.g. to get width and height of custom painting area use something like:
  - ```
    Insets insets = getInsets();
    int currentWidth = getWidth() -
          insets.left - insets.right;
    int currentHeight = getHeight() -
          insets.top - insets.bottom;
    ```
    ```

Graphics - shapes and text (1)

- Properties stored in Graphics object.
  - Represents ‘state’
    - eg Color, Font, etc
- Methods of Graphics can draw...
  - Shapes
    - lines, rectangles, 3D rectangles, round-edged rectangles, ovals, arcs, polygons
  - Text
    - Draw string to screen as a graphic

Graphics - shapes and text (2)

- Drawing a shape on screen
- Use paintComponent Graphics

```
void paintComponent(Graphics g) {
    super(g)
    g.setColor(Color.RED);
    g.drawRect(x, y, width, height);
}
```

- shapes: x and y specify upper left

Graphics - shapes and text (3)

- Drawing a string on screen
- Use **`paintComponent`** Graphics

```
void paintComponent(Graphics g) {
    super(g)
    g.setFont(new Font("Serif", Font.PLAIN,
                       12));
    g.drawSring("Java Swing", x, y);
}
```

- x and y specify *baseline left* of text

Graphics - Images

- Swing Icons by far and away the easiest method of displaying graphics.
- If more features needed then use the AWT/ Graphics functionality
- Supports GIF, PNG and JPEG *NOT BMP*
- Loading and displaying images

```
Image myImage
=Toolkit.getDefaultT
oolkit().getImage(fi
lename);
g.drawImage(myImage,
x, y, this);
g.drawImage(myImage,
x, y, width,
height,this);
```

Java 2D - Images

- Much more powerful than basic image display
- Images are best rendered in a Buffer off screen as a BufferedImage
- Then drawn to screen with call to Graphics2D.drawImage
- Graphics2D provides variety of image filtering options
  - sharpen/blur, rotate, transform, scale etc

Java 2D – (VERY) Basic use

- Simply cast the Graphics object to Graphics2D

```
void paintComponent(Graphics g) {
    super(g);
    Graphics g2 =
        (Graphics2D)g.create();
    g2.setColor(Color.BLACK);
    g2.drawRect(x, y, width, height);
     g2.dispose();
}
```

Graphics 2D

- Graphics2D extends Graphics
- New properties include
  - Stroke
  - Paint
- New capabilities
  - drawing and filling Shape objects
  - applying AffineTransforms

2D Objects

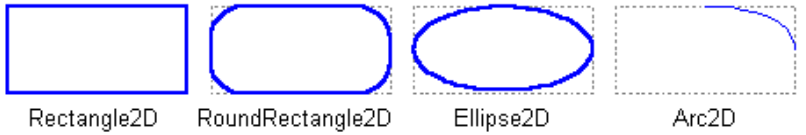
- A 2D object consists of a set of points in a plane
- Examples
  - point
  - lines, curves
  - shapes like rectangles, ellipses, polygons
  - text and images

Graphics primitives

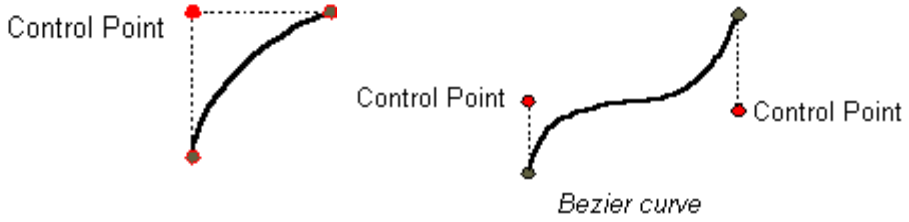
- Point
- Line
- Curve
- Rectangle
- Ellipse/circle
- arc

Java2D Shapes

- Rectangle Shapes



- QuadCurve2D and CubicCurve2D



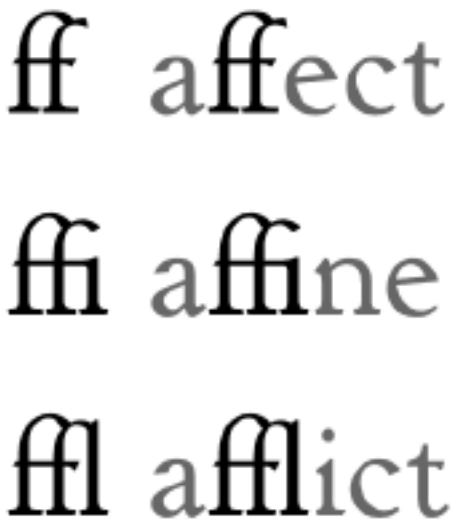
- GeneralPath
  - Arbitrary shapes made by connecting lines

Java 2D - Text

- Often easier to use JLabels since much of the work is done for you
- Use the drawString method
  - Eg g.drawString("Hello", x, y);
- To set font use setFont method
  - Create an instance of Font

Fonts and drawing strings

- Font provides description of the shape of a collection of chars
  - Called *glyphs*
  - Information about how to advance after drawing glyph
  - Aggregate information for whole collection
- More recent formats (OpenType™) can specify lots more
  - Alternates
  - ligatures



Drawing fonts

- Sometimes easier to use drawString() method but font family gives you full control over appearance

Fonts

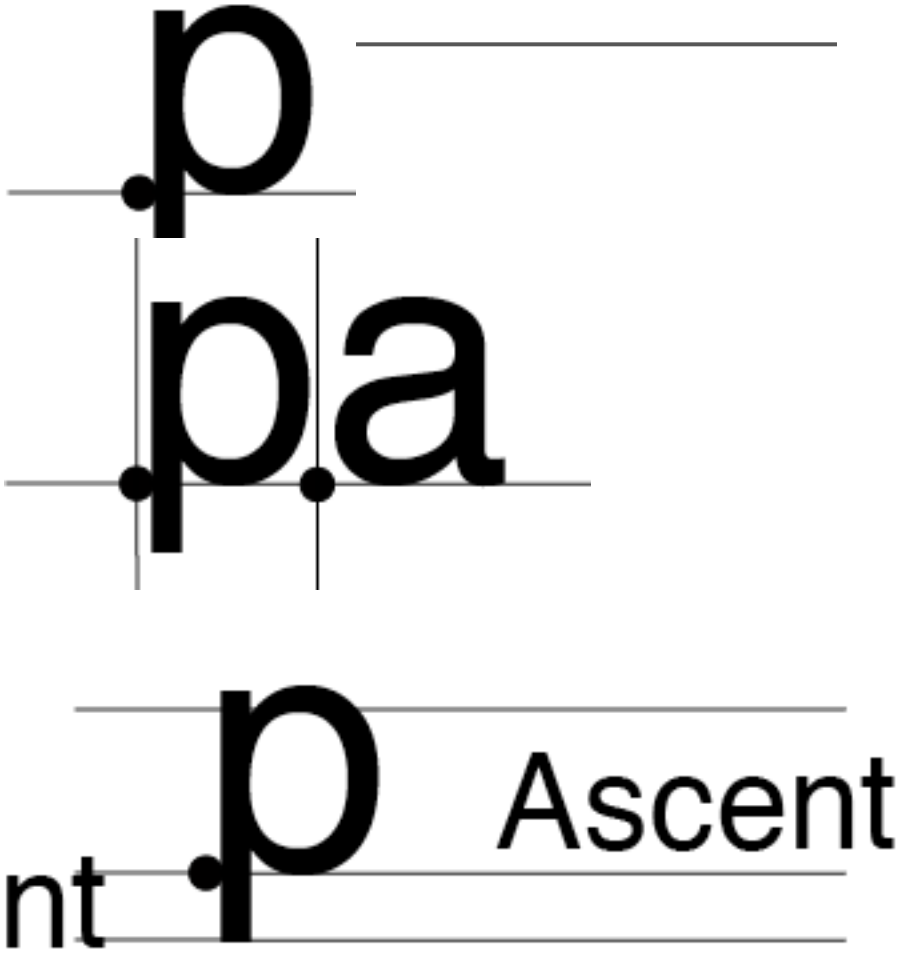
- Typically specified by
  - Family or typeface (courier, helvetica, geneva, ….)
  - Size (in points)
  - Style (**bold**, *italic*, plain, ….)
- See `java.awt.Font`

FontMetrics

- Objects that allow you to measure characters, strings and properties of whole fonts
- java.awt.FontMetrics
- Get it by using
  - FontMetrics fm = new FontMetrics (Graphics.getFontMetrics());

FontMetrics

- Reference point and baseline
- Advance width
  - Where reference point of next glyph goes along baseline
- Ascent and descent



- http://java.sun.com/docs/books/tutorial/2d/geometry/examples/ShapesDemo2D.java
- Demo code for drawing each of these shapes
- Can fill, change line (stroke) appearance and texture objects

Images

- Java.awt.image.BufferedImage

Loading, Displaying and Scaling Images

- Demonstrate some Java multimedia capabilities
  - **`java.awt.Image`**
    - **`abstract`** class (cannot be instantiated)
    - Can represent several image file formats
      - e.g., GIF, JPEG and PNG
  - **`javax.swing.ImageIcon`**
    - Concrete class