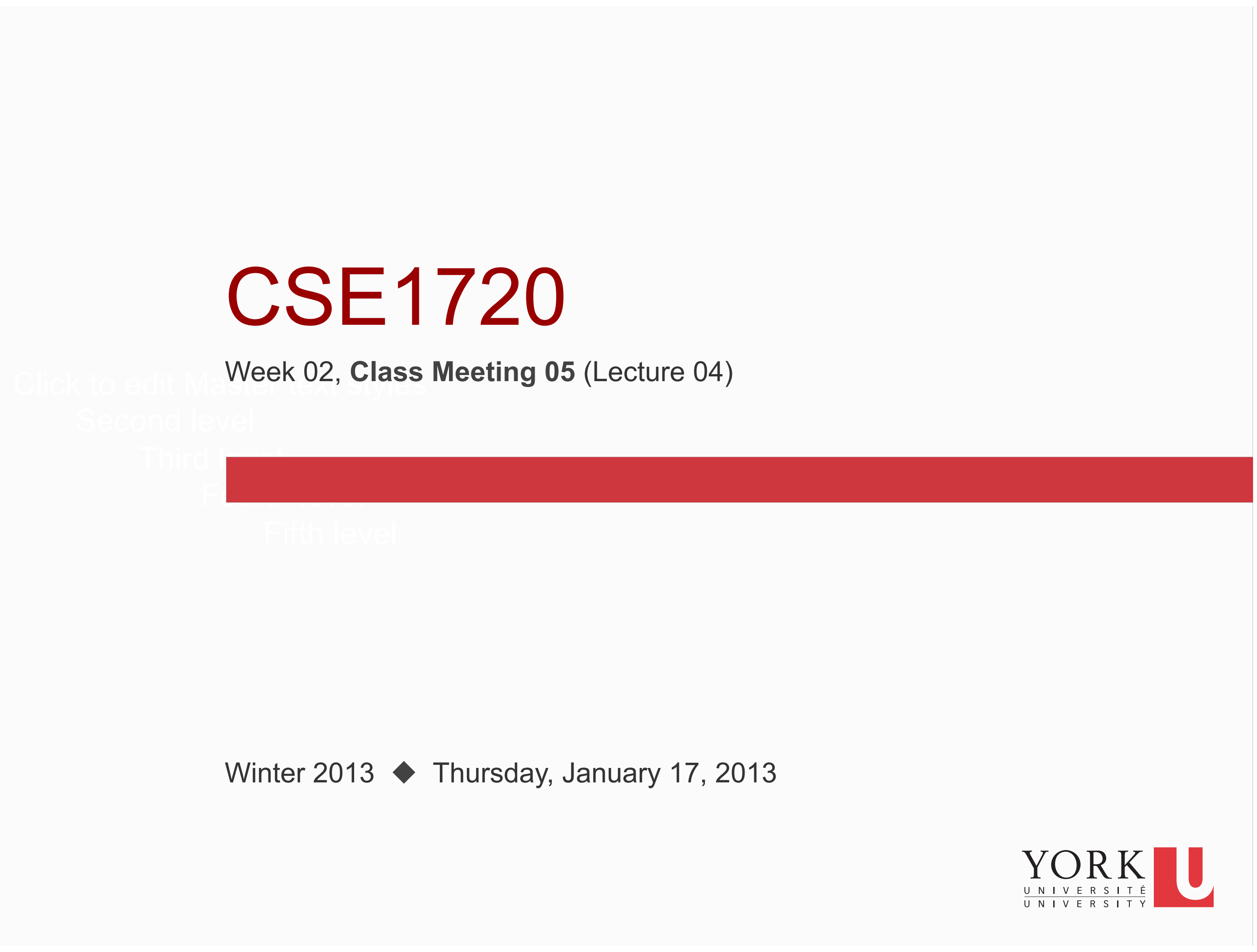
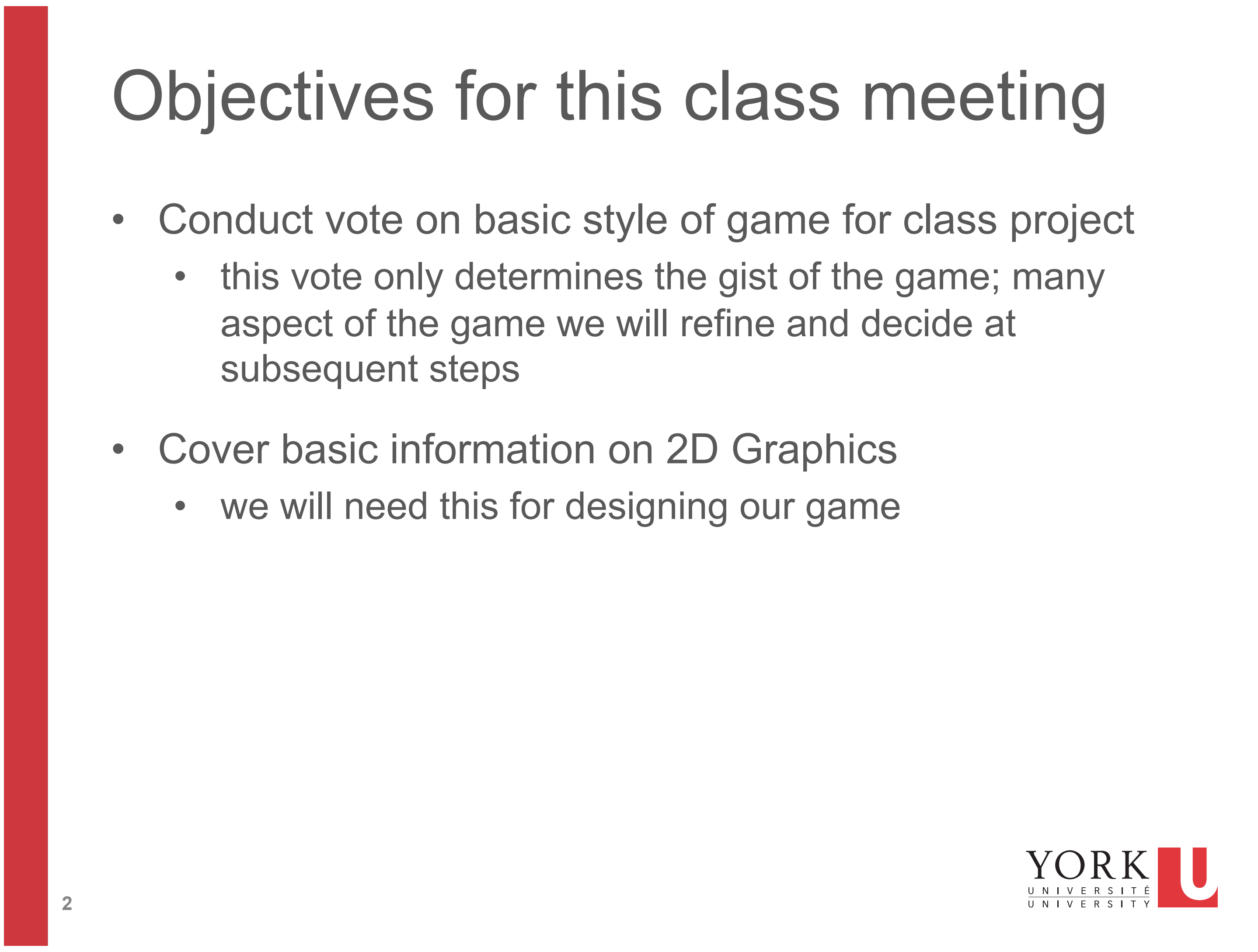


CSE1720

Week 02, **Class Meeting 05** (Lecture 04)

Winter 2013 ◆ Thursday, January 17, 2013



Objectives for this class meeting

- Conduct vote on basic style of game for class project
  - this vote only determines the gist of the game; many aspect of the game we will refine and decide at subsequent steps
- Cover basic information on 2D Graphics
  - we will need this for designing our game

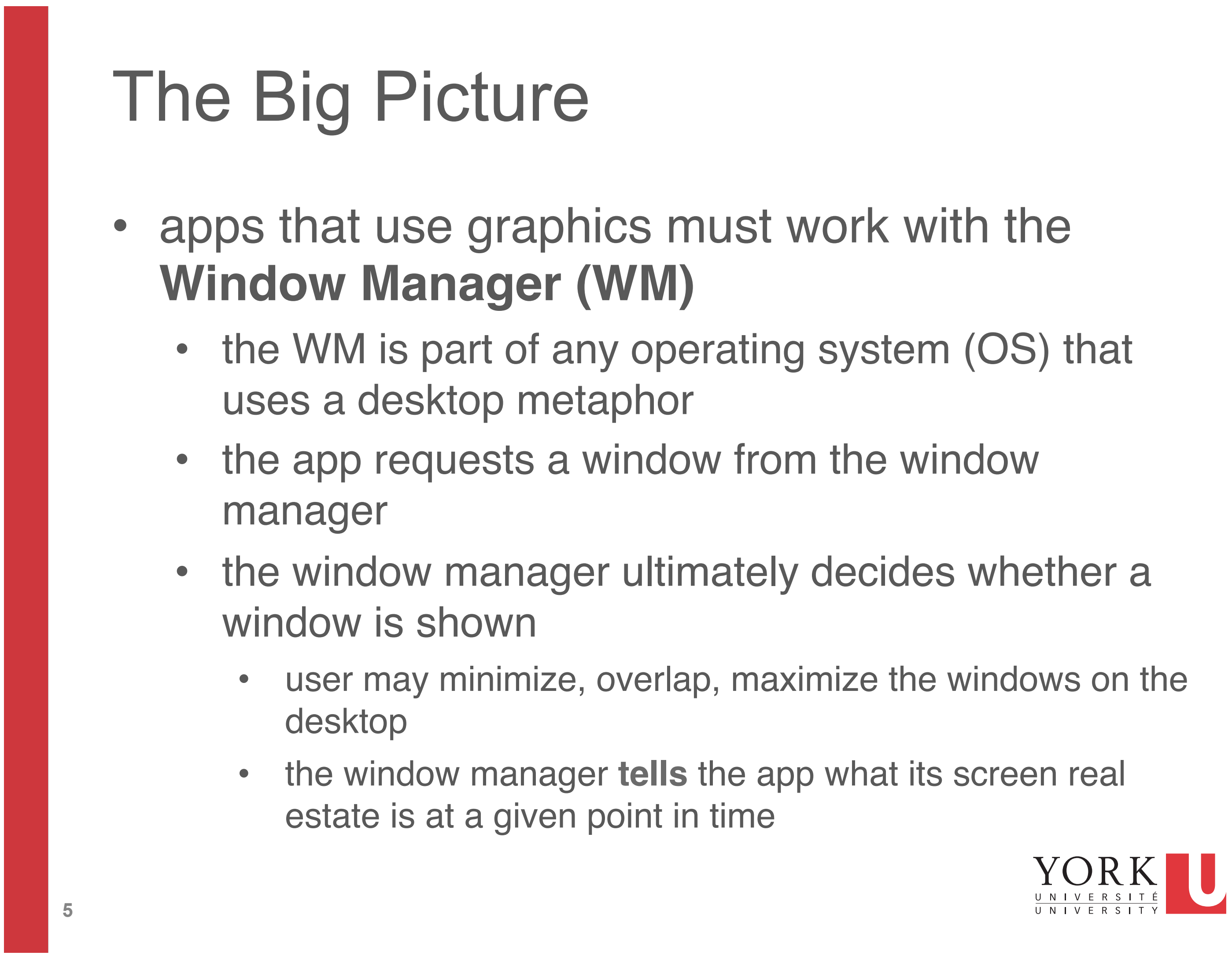
Voting on the game

- you will each receive a ballot by email
- submit the ballot as follows

```
submit 1720 Vote ballot.txt
```

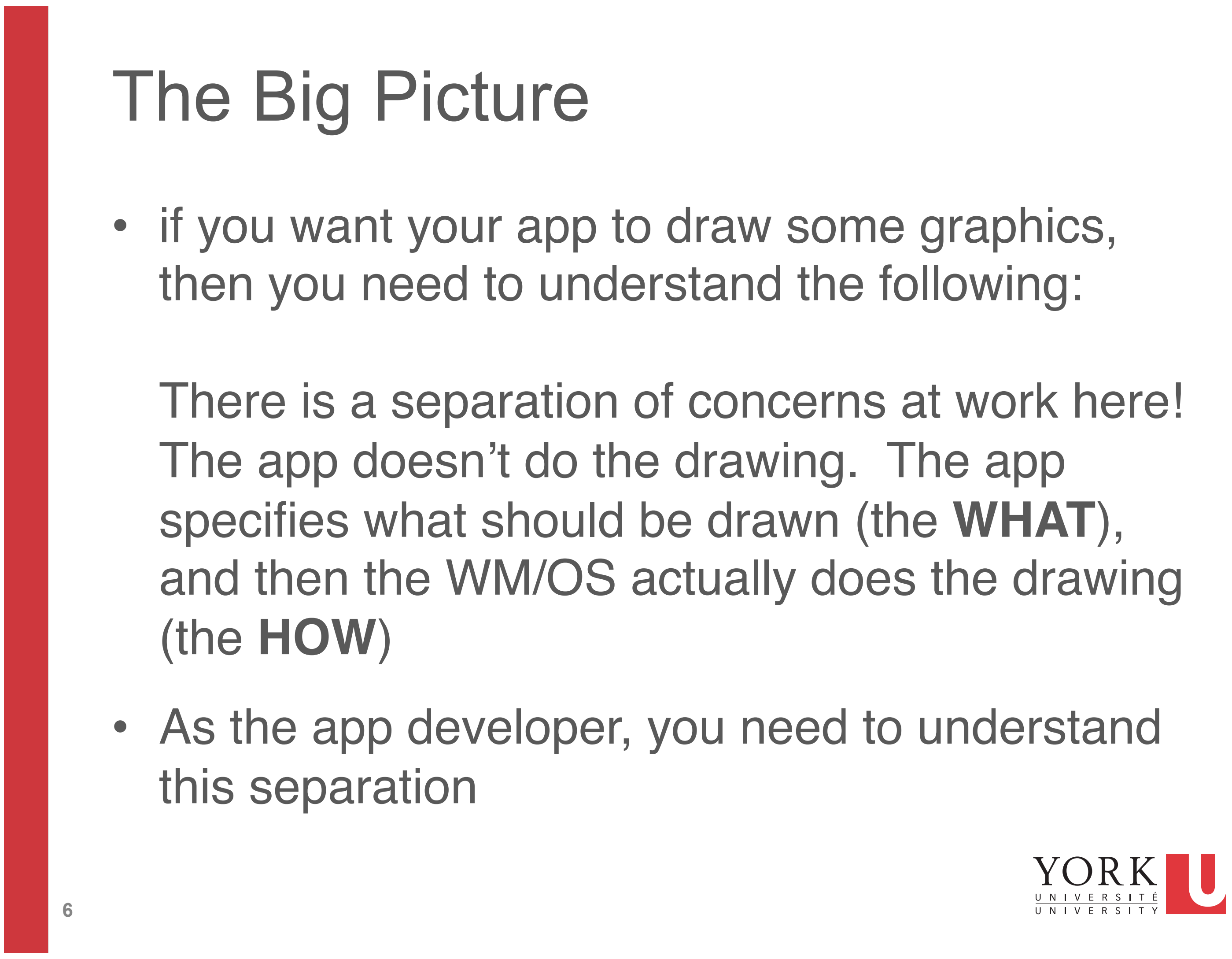
Basic Graphics

- Suggested background reading:
  The Java Tutorials, **Trail: 2D Graphics**
- **http://docs.oracle.com/javase/tutorial/2d/index.html**
- These lecture slides provide a basic overview of that material, enough to get you started with the lab exercises



The Big Picture

- apps that use graphics must work with the **Window Manager (WM)**
  - the WM is part of any operating system (OS) that uses a desktop metaphor
  - the app requests a window from the window manager
  - the window manager ultimately decides whether a window is shown
    - user may minimize, overlap, maximize the windows on the desktop
    - the window manager **tells** the app what its screen real estate is at a given point in time



The Big Picture

- if you want your app to draw some graphics, then you need to understand the following:

  There is a separation of concerns at work here! The app doesn’t do the drawing. The app specifies what should be drawn (the **WHAT**), and then the WM/OS actually does the drawing (the **HOW**)
- As the app developer, you need to understand this separation

`Graphics2D` class services

- the `Graphics2D` object encapsulates the “HOW” part of the drawing
- the complexity of the “HOW” is hidden from the clients
  - how to translate drawing coordinates to screen coordinates
  - which pixels need to be modified and how
  - all of the low level stuff that concerns graphics rendering

`Graphics2D` class services

- an app that has a window will be able to access the `Graphics2D` object that is associated with the window
  - console based apps cannot get access to a Graphics2D object – there is no window!!
- The client (the app) uses the `Graphics2D` object to specify the “WHAT” --- a description of which graphic primitives are desired. The `Graphics2D` object, as part of its services, deals with getting those primitive rendered as computer graphics.
- 2D Graphic primitives include: basic geometric shapes, lines, arcs, text

Graphics2D class services

- The client uses the methods of `Graphics2D` to specify the graphic primitives to be drawn
  - a useful method is called `draw`
  - it takes one argument, a `Shape`
  - all of the graphic primitives can be provided, since Shape is the parent class
- **The *manner* in which these primitives are rendered depends on the current values of several properties of the `Graphics2D` object.**
- By manner – we are talking about aspects such as the width and color of the lines.

Graphics2D class services

- The client specifies the graphic primitives to be drawn in *user space* (in “coordinate units”)
- `Graphics2D` class services translates the coordinates in *user space* to coordinates in *device space* (in pixels)
- Depending on the screen resolution, one point in user space may translate to several pixels in device space
- Your app can invoke the following to determine the screen resolution in dots per inch:
  `Toolkit.getDefaultToolkit().getScreenResolution()`

Coordinate spaces

- From: http://docs.oracle.com/javase/tutorial/2d/overview/coordinate.html
- The Java 2D API maintains two coordinate spaces:
  - *User space* – The space in which graphics primitives are specified
  - *Device space* – The coordinate system of an output device such as a screen, window, or a printer
- User space is:
  - a device-independent logical coordinate system.
  - the coordinate space that your program uses.
- All geometries passed into Java 2D rendering routines are specified in user-space coordinates.
- When it is time to render the graphics, a transformation is applied to convert from user space to device space. The origin of user space is the upper-left corner of the component's drawing area.

11

Examples

Construct a graphic primitive and draw it

```
Rectangle2D.Double shape1 =
            new Rectangle2D.Double(5, 15, 20, 50);
```

20 units wide and 50 units high, as given in "coordinate units". The upper left hand corner is anchored at (5,15)

If your screen resolution is 72, then there will be 72 "coordinate units" per inch. But this can vary.

The name of the class is weird – there is a dot in the middle of it. Nevermind this for the moment!

12

Examples

Now, the weirdly-named class is a sub-class of the class `Rectangle2D`

So we can do this:

```
Rectangle2D shape1 =
            new Rectangle2D.Double(5, 15, 20, 50);
```

…And also the class `Rectangle2D` is a subclass of `Shape`

So we can do this:

```
Shape shape1 =
            new Rectangle2D.Double(5, 15, 20, 50);
```

13

Examples

We haven't drawn anything yet!

We need to tell the `Graphics2D` object that we want `shape1` to be drawn.

So we do this to obtain a reference to the Graphics2D object:

```
Graphics2D graphicsObj = myPict.getGraphics();
```

(Assuming here that `myPict` is a `Picture` object)

And then we tell the graphics2D object that we want shape1 drawn:

```
graphicsObj.draw(shape1);
```

14

Examples

The rectangle is drawn with the current settings of the `Graphics2D` object.

To change the colour of the “pen” (so to speak)

```
graphicsObj.setColor(Color.BLUE);
graphicsObj.draw(shape1);
graphicsObj.setColor(Color.RED);
graphicsObj.draw(shape1);
```

This draws a red rectangle on top of the blue rectangle

Any shape that is drawn is drawn with the current settings until the settings change

Examples

Note that there is no way to “move” rectangle.

- You can move the origin of the coordinate system up/down or left/right
  - this will make it appear as though the anchor of the rectange has moved
  - this is not recommended at this point, since we want a fixed origin
- Instead, just instantiate new shapes with different anchor points

Examples

Instead of drawing the outline of a shape, we can draw it as a filled shape

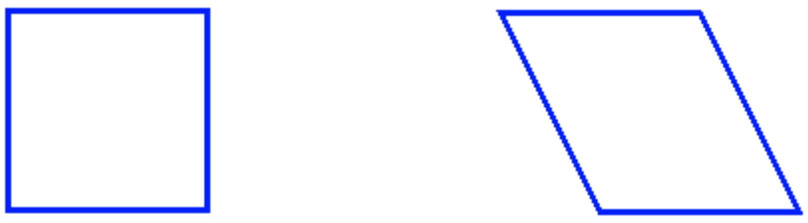
```
graphicsObj.setColor(Color.BLUE);

graphicsObj.fill(shape1);
```

About transformations

Once a shape is specified in user space, then any number of **transformations** can be applied to it

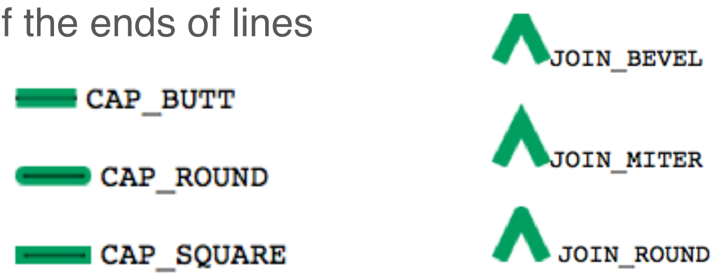
For instance, here is a shear transformation of a rectangle



There are also transformations to rotate and scale.

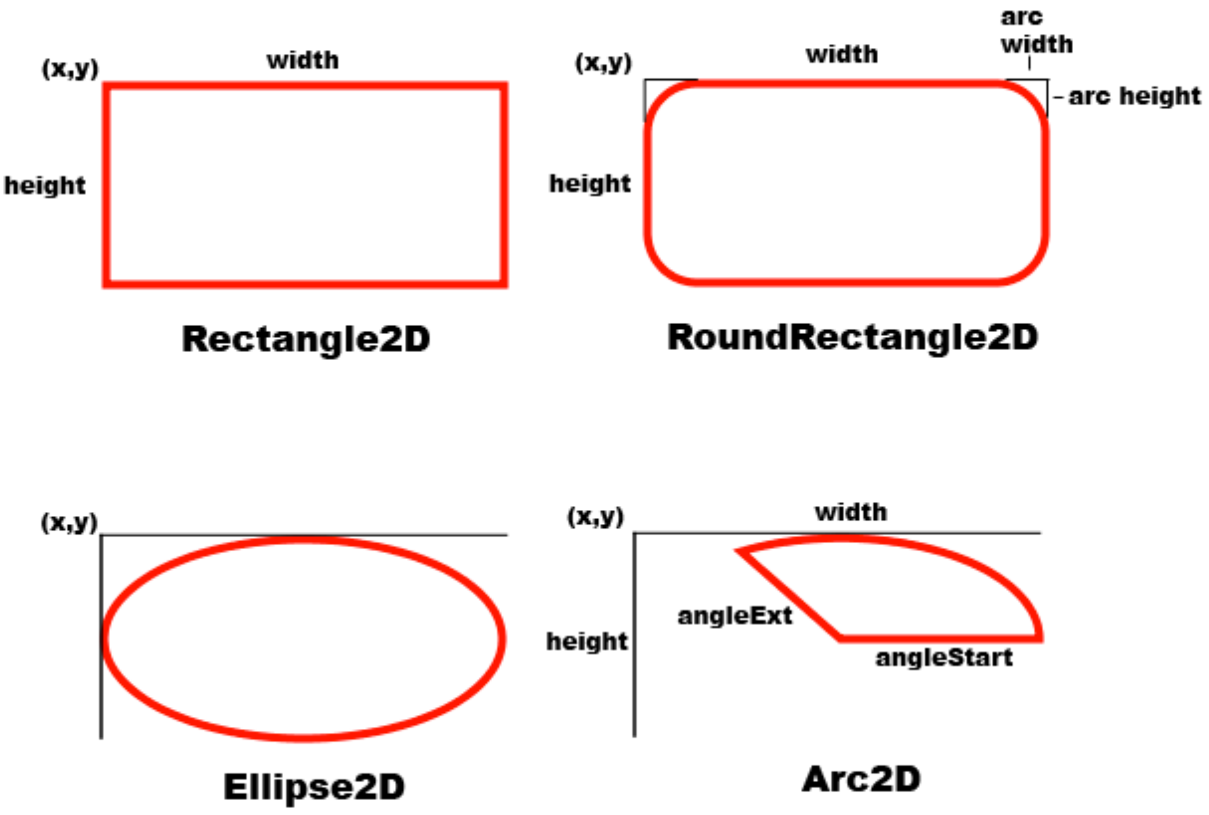
About "State"

- There are **settings** for several aspects of drawing:
  - The stroke width, the way the strokes are joined together, the appearance of the ends of lines



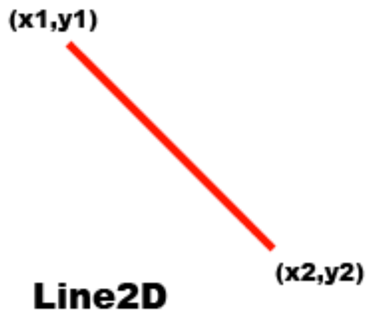
  - The current translation, rotation, scaling, and shearing values
  - The paint color
  - The fill pattern
- Since these aspects are controlled by attribute values, we say that the **state** of the `Graphics2D` object determines the drawing settings.

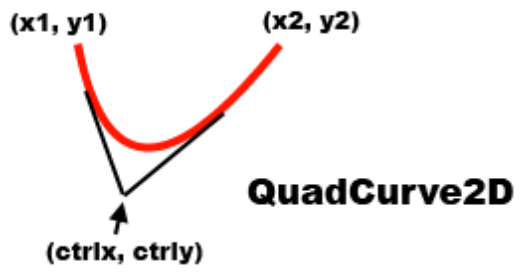
Shape Primitives

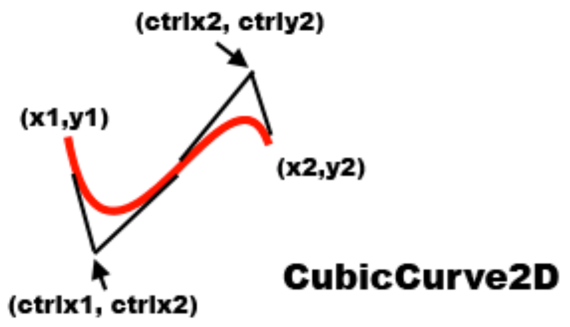


Shape Primitives

ref: http://java.sun.com/developer/technicalArticles/GUI/java2d/java2dpart1.html







About Stroke

- Stroke controls the width of the drawing pen
- The default width is 1 unit (typically 1 pixel wide, so it is teeny-tiny)
- Here's how to change it:

```
BasicStroke newStroke = new BasicStroke(4.0);
graphicsObj.setStroke(newStroke);
```

Since `Stroke` is the parent class of `BasicStroke`, you can also write:

```
Stroke newStroke = new BasicStroke(4.0);
graphicsObj.setStroke(newStroke);
```

About Colour

- Paint controls the colour of the drawing pen
- The default width is WHITE
- Here's how to change it (older version):

```
graphicsObj.setColor(Color.BLUE);
```

- Here's how to change it (newer, better version):

```
graphicsObj.setPaint(Color.BLUE);
```

the `setPaint` method takes a `Paint` argument, and a `Color` object can fit the bill!

Here is a fancier fill

```
Point p1 = new Point(0, 0);
Point p2 = new Point(50, 50);
GradientPaint paint1 =
       new GradientPaint(p1, Color.RED, p2, Color.MAGENTA,
true);
graphicsObj.setPaint(paint1);
```

Try it yourself!

To Do:

- Practise using all of these various methods and experiment on your own.
- Complete the lab exercises