# Week 4 – Part 2

## Introduction to 2D Graphics & Java 2D

## 2D graphics

### Vector graphics

- Use of geometric primitives: points, lines, curves, etc.
- Primitives are created by using mathematical equations
- Can be zoomed infinitively, moved or transformed without losing in quality

### Raster (bitmap) graphics

- Images represented as pixels
- Resolution dependent: scaling affects image quality
- Stored in image files

## Types of image editors

Vector-based

- Adobe Illustrator
- CorelDraw
- Inkscape

Raster-based

- Photoshop
- Painter
- GIMP

## Java 2D API

Provides 2D graphics, text & image capabilities

- Wide range of geometric primitives
- Mechanisms for hit detection of shapes, text, images
- Color & transparency
- Transformations
- Printing
- Control of the quality of rendering

# Java 2D – Base classes

Graphics class : abstract base class for all graphics contexts, allowing applications to draw onto components

```
public class RectWidget extends JPanel {
    private int posx, posy, w, h;
    private Color color;

    public RectWidget(int x, int y, int w, int h, Color color){
        this.posx = x; this.posy = y;
        this.w = w; this.h = h;
        this.color = color;
    }

    public void paint(Graphics g) {
        g.setColor(color);
        g.drawRect(x, y, w, h);
    }
}
```

## Java 2D – Base classes

JComponent’s relevant methods

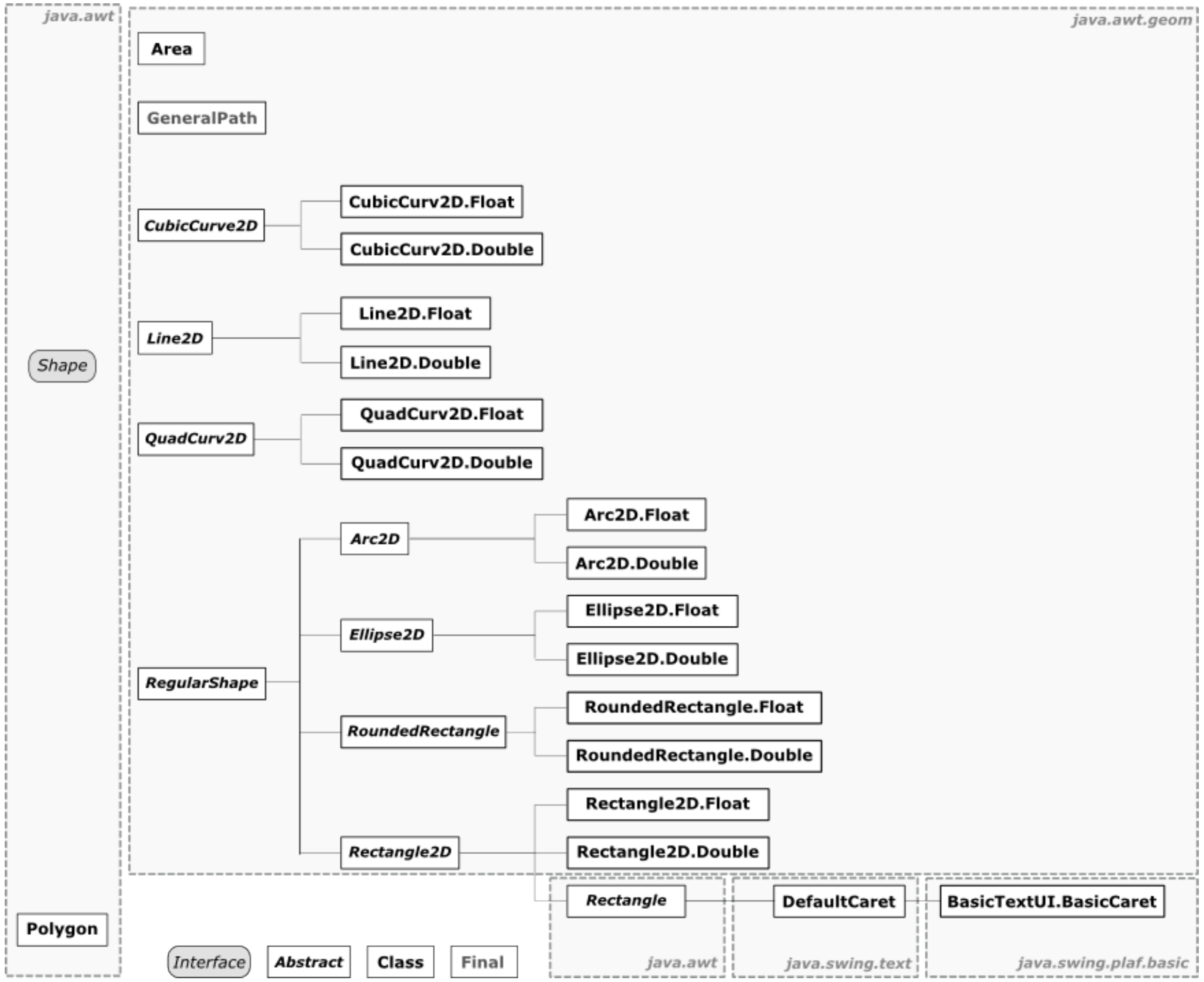
```
public paint(Graphics g)
protected paintComponent(Graphics g)
protected paintBorder(Graphics g)
protected paintChildren(Graphics g)


public print(Graphics g)
protected printComponent(Graphics g)
protected printBorder(Graphics g)
protected printChildren(Graphics g)
```

## Java 2D – Base classes

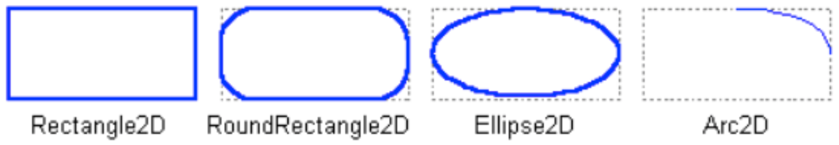
Graphics2D class : extends Graphics class to provide more sophisticated control over geometry, transformations, etc.

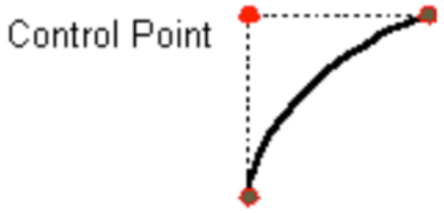
```
private double x, y, w, h;
...
public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.draw(new Rectangle2D.Double(x , y, w, h));
}
```

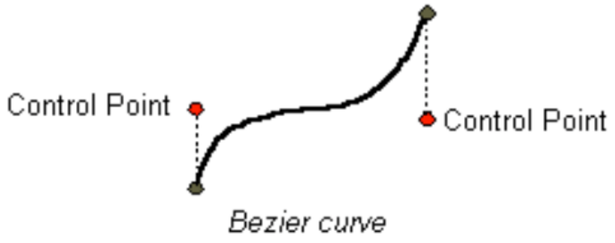
# Geometric primitives



# Shapes

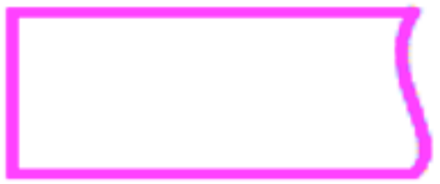






Quadratic Bézier curve

Cubic Bézier curve



Arbitrary shapes (*GeneralPath*)

## Bézier curves

Parametric curves widely used in Computer Graphics

Used to model smooth curves that can be scaled indefinetely

First studied by mathematician Paul de Casteljau (1959) and widely publicized by Pierre Bézier (1962)

# Bézier curves

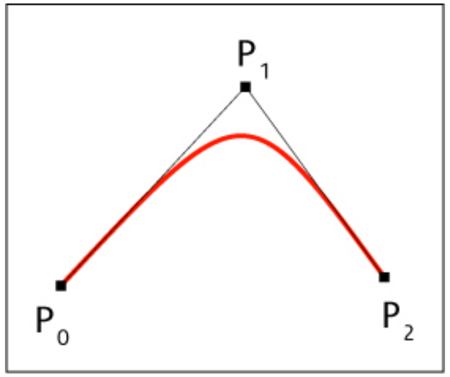
Defined by a set of control points: $P_{0}, \ldots, P_n$

**Linear Bézier curve**: straight line between $P_0$ and $P_1$

$B(t) = (1 - t)P_0 + tP_1,\ t \in [0,1]$

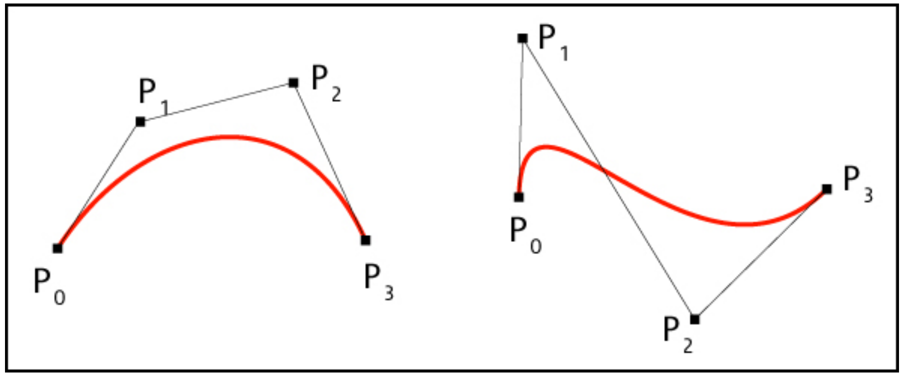
**Quadratic Bézier curve**:

$B(t) = (1 - t)^2P_0+2(1 - t)tP_1+t^2P_2,\ t \in [0,1]$



**Cubic Bézier curve**:

$B(t) = (1 - t)^3P_0+3(1 - t)^2tP_1+3(1-t)t^2P_2+t^3P_3,\ t \in [0,1]$



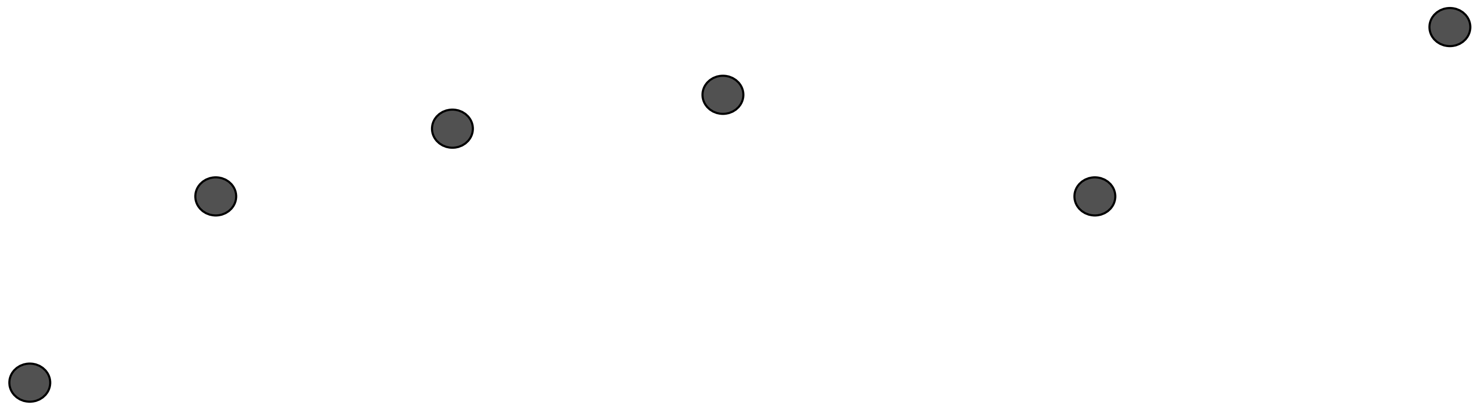
# Examples in Java

```
// create new QuadCurve2D.Float
QuadCurve2D q = new QuadCurve2D.Float();
q.setCurve(p0.getX(), p0.getY(), p1.getX(), p1.getY(), p2.getX(), p2.getY());
g2.draw(q);
```

```
// create new CubicCurve2D.Double
CubicCurve2D c = new CubicCurve2D.Double();
c.setCurve(p0.getX(), p0.getY(), p1.getX(), p1.getY(), p2.getX(), p2.getY(),
                                                         p3.getX(), p3.getY());
g2.draw(c);
```

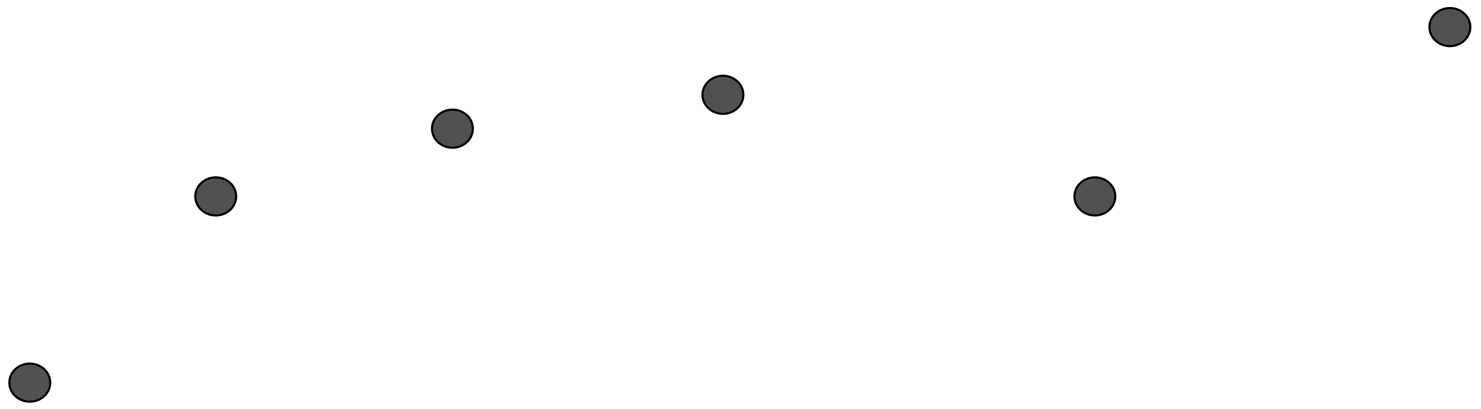
## Curves from points

Given a sequence of points, how do we create a smooth curve?



## Curves from points

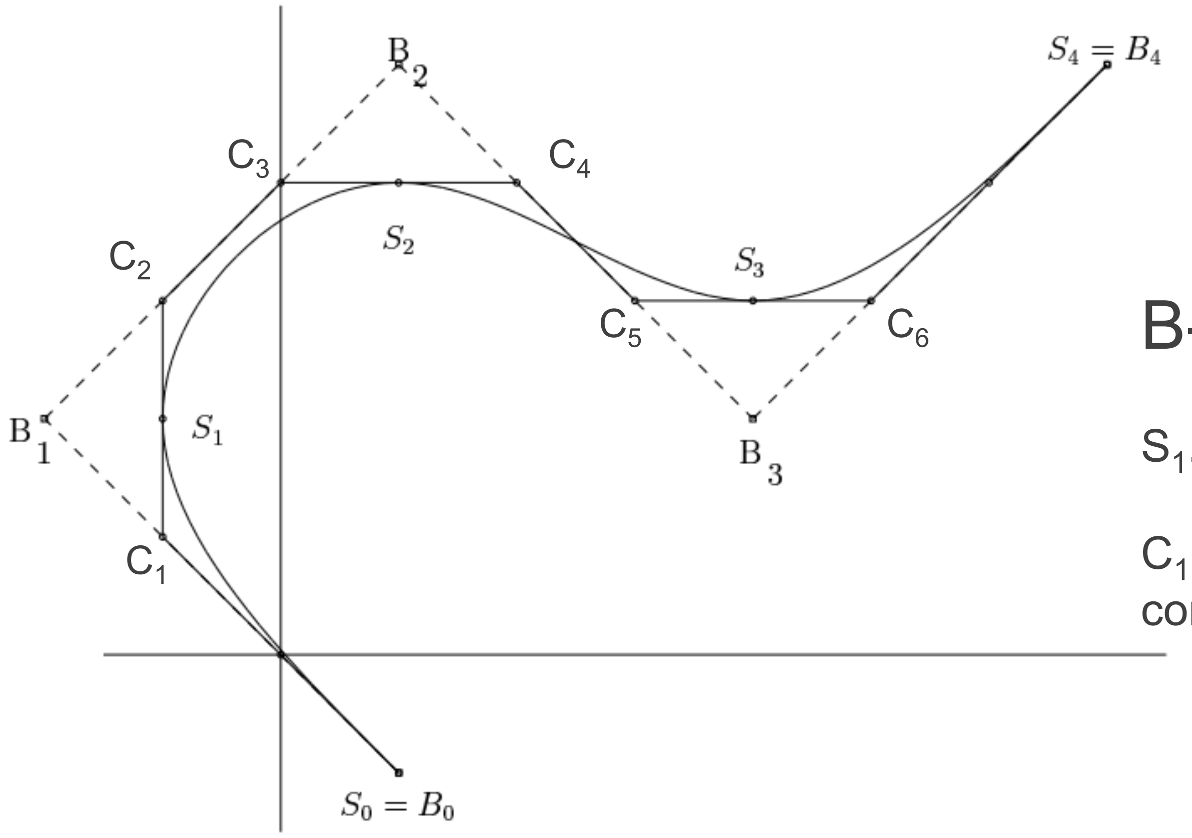
Given a sequence of points, how do we create a smooth curve?



Easy but ugly: connect the points with straight lines

## Curves from points

Better solutions: parametrize the curve as connected cubic Bézier curves



B-spline

$S_1 \ldots S_4$ are the points

$C_1 \ldots C_6$ are additional control points

# Arbitrary shapes

GeneralPath path = new GeneralPath();

Move the current point of the path to the given point

    path.moveTo(x, y);

Add a line segment to the current path

    path.lineTo(x, y);

Add a quadratic curve segment to the current path

    path.quadTo(ctrlx, ctrly, x2, y2);

Add a cubic curve segment to the current pathclosePath

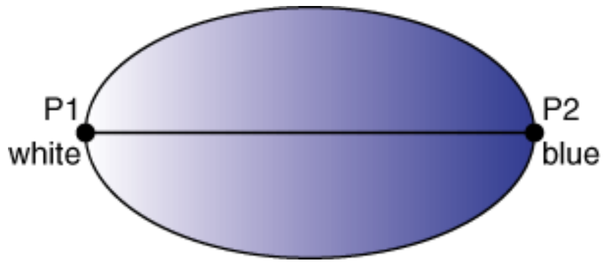
    path.curveTo(ctrlx1, ctrly1, ctrlx2, ctrly2, x3, y3);

Close the current path

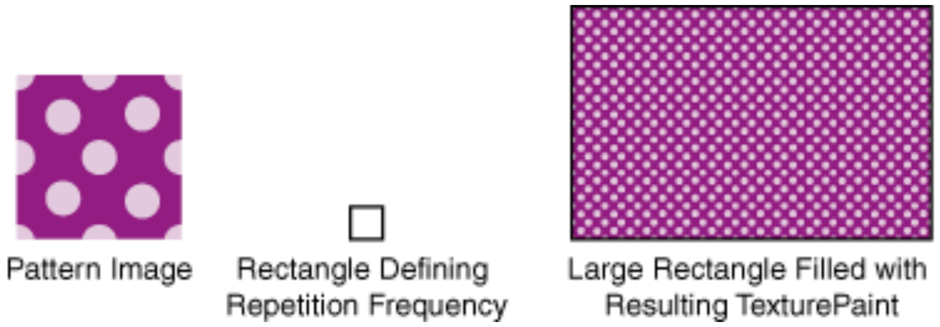
    path.closePath();

# Stroking and painting

stroke patterns



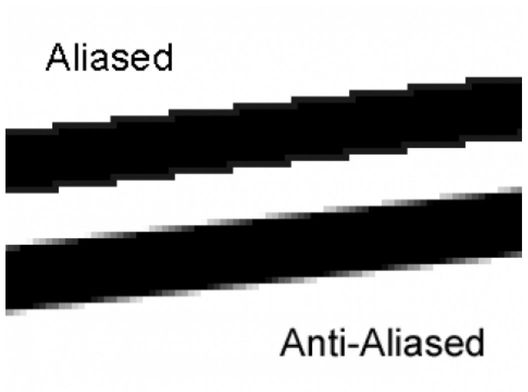
gradient filling colors


Pattern Image
Rectangle Defining Repetition Frequency
Large Rectangle Filled with Resulting TexturePaint

filling patterns

## Rendering hints & antializing

```
public void paint (Graphics g){
    Graphics2D g2 = (Graphics2D)g;
    RenderingHints rh = new
    RenderingHints(RenderingHints.KEY_TEXT_ANTIALIASING,
                        RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2.setRenderingHints(rh);
    ...
}
```



## Clipping

Restricts the drawing area to be rendered

## Clipping

Restricts the drawing area to be rendered

```
rect.setRect(x + marginx, y + marginy, w, h);
g2.clip(rect);
g2.drawImage(image, x, y, null);
```

## Tansformations

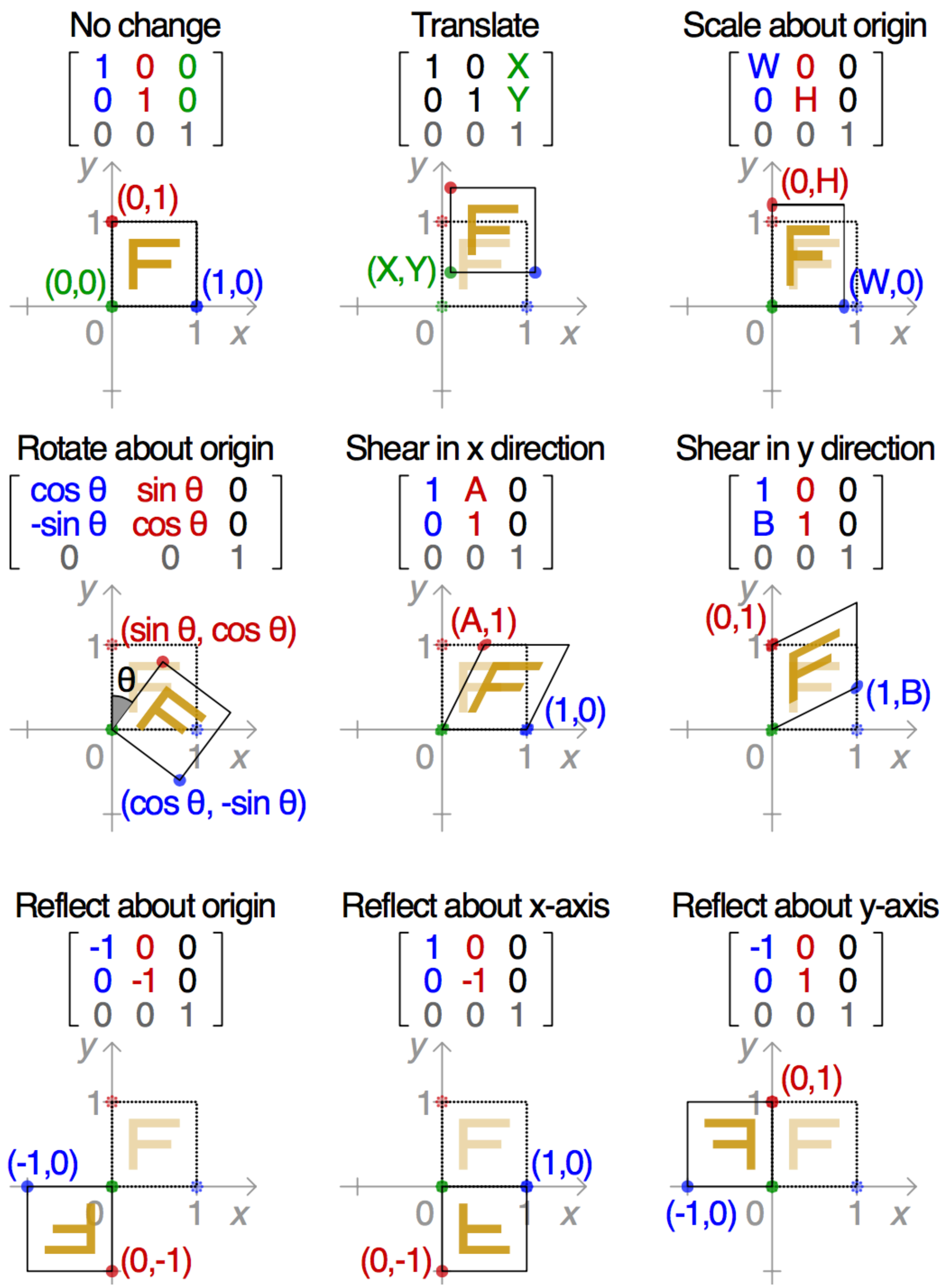
rotate, scale, translate, shear methods of Graphics2D

```
g2.translate(100, 200);
```

AffineTransform class

```
AffineTransform atransf = new AffineTransform();
atransf.rotate(Math.PI/2); // rotate 90°
g2.transform(atransf);
```

# Affine Transformations


No change
1 0 0
0 1 0
0 0 1
(0,1)
(0,0)
(1,0)
Translate
1 0 X
0 1 Y
0 0 1
(X,Y)
Scale about origin
W 0 0
0 H 0
0 0 1
(0,H)
(W,0)
Rotate about origin
cos θ sin θ 0
-sin θ cos θ 0
0 0 1
(sin θ, cos θ)
θ
(cos θ, -sin θ)
Shear in x direction
1 A 0
0 1 0
0 0 1
(A,1)
(1,0)
Shear in y direction
1 0 0
B 1 0
0 0 1
(0,1)
(1,B)
Reflect about origin
-1 0 0
0 -1 0
0 0 1
(-1,0)
(0,-1)
Reflect about x-axis
1 0 0
0 -1 0
0 0 1
(1,0)
(0,-1)
Reflect about y-axis
-1 0 0
0 1 0
0 0 1
(0,1)
(-1,0)
x
y
0
1