CSc 133 Lecture Notes

14 - Applications of Affine Transforms

Computer Science Department

California State University, Sacramento

Overview

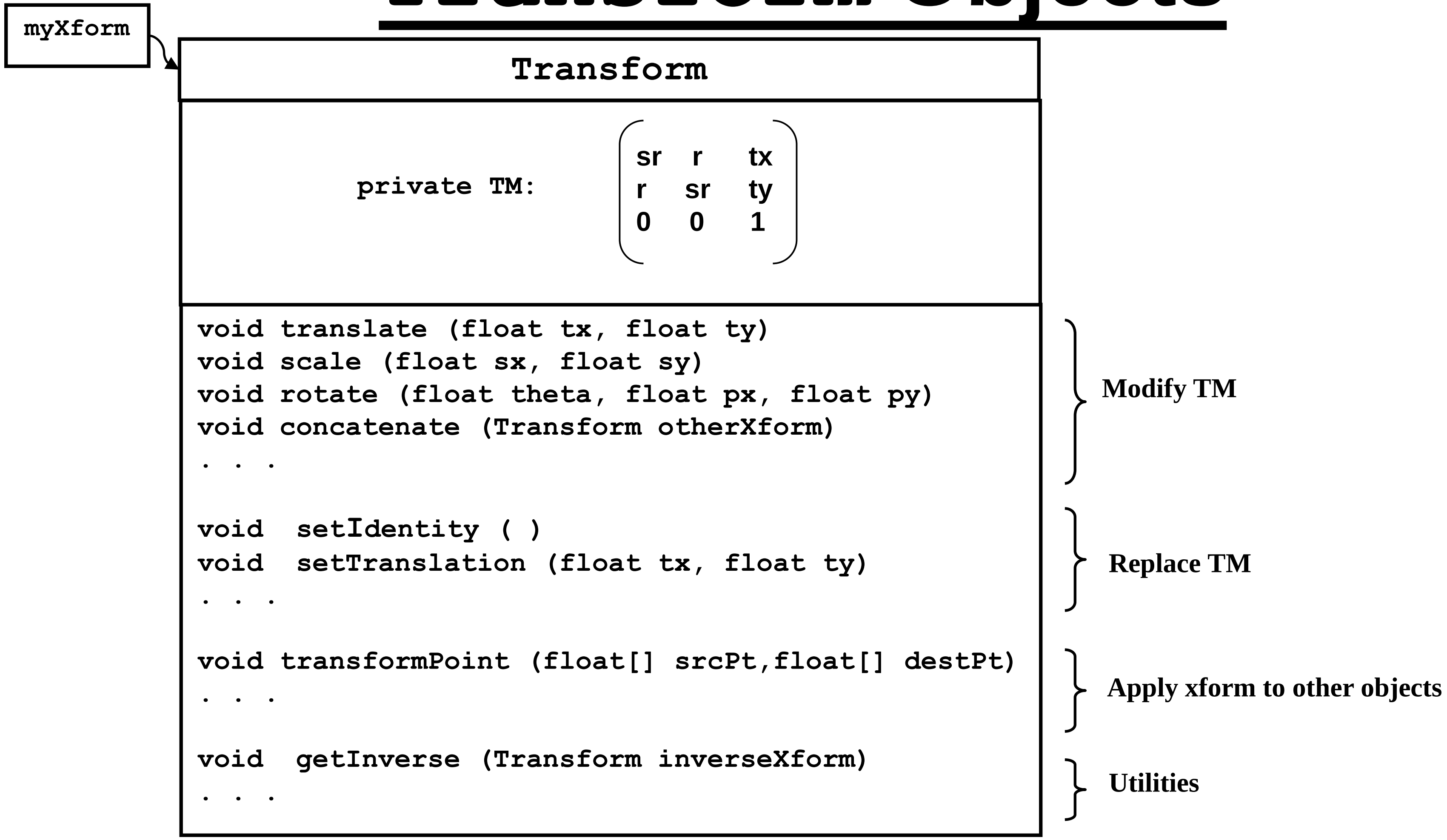
- `Transform` Class
- Local Coordinate Systems
- Display-Mapping Transforms
- `Graphics` Class revisited
- Transformable Objects
- Composite Transforms
- Hierarchical Object Transforms
- Dynamic Transforms

Transform Class

- `com.codename1.ui.Transform`
- **Contains**
  - **A 3×3 "Transformation Matrix" (TM)**
    - **Uses *column-major* form**
    - ***Only the active 2x3 elements can be accessed***
  - **Methods to *manipulate* TM**
  - **Methods to *apply* the transform (xform) to other objects**
- **To initialize use the following static function:**

```
Transform myXform = Transform.makeIdentity();
```

Transform Objects



myXform

Transform

private TM:

$$\begin{pmatrix} sr & r & tx \\ r & sr & ty \\ 0 & 0 & 1 \end{pmatrix}$$

```
void translate (float tx, float ty)
void scale (float sx, float sy)
void rotate (float theta, float px, float py)
void concatenate (Transform otherXform)
. . .
```

Modify TM

```
void  setIdentity ( )
void  setTranslation (float tx, float ty)
. . .
```

Replace TM

```
void transformPoint (float[] srcPt,float[] destPt)
. . .
```

Apply xform to other objects

```
void  getInverse (Transform inverseXform)
. . .
```

Utilities

- Methods for modifying TM (e.g., `translate()`, `scale()` and `rotate()`) are always applied relative to the **screen origin** (i.e., coordinates passed to these methods are relative to screen origin).
- Also these methods multiply the new transform to the current TM ***on the right,*** which means the transform concatenated last to the xform will be applied first to a point.

Using `Transform` Object

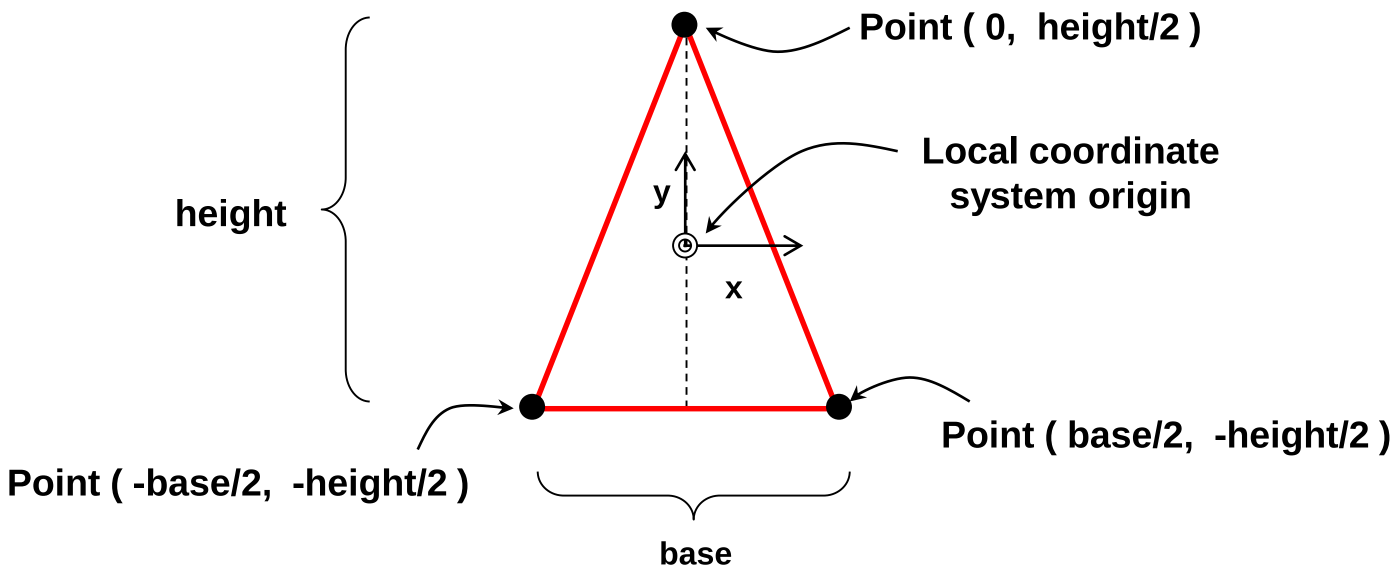
```
...
float[] p1 = new float[]{x,y};
float[] p2 = new float[]{0,0};
Transform myXform = Transform.makeIdentity();
myXform.rotate(Math.toRadians(45), 0, 0);
myXform.transformPoint (p1,p2);
```

$$\begin{bmatrix} x2 \\ y2 \\ 1 \end{bmatrix} = \begin{bmatrix} & & \\ & Rotate(45^0) & \\ & & \end{bmatrix} \times \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

"Local" Coordinate Systems

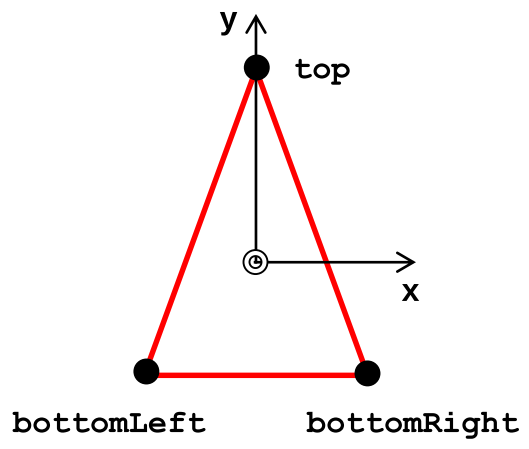
Define objects *relative to their own origin*

- Example: triangle
  - Base & Height
  - Local origin at "center"
  - Points defined *relative to local origin*


Point (0, height/2)
Local coordinate system origin
height
y
x
Point (-base/2, -height/2)
Point (base/2, -height/2)
base

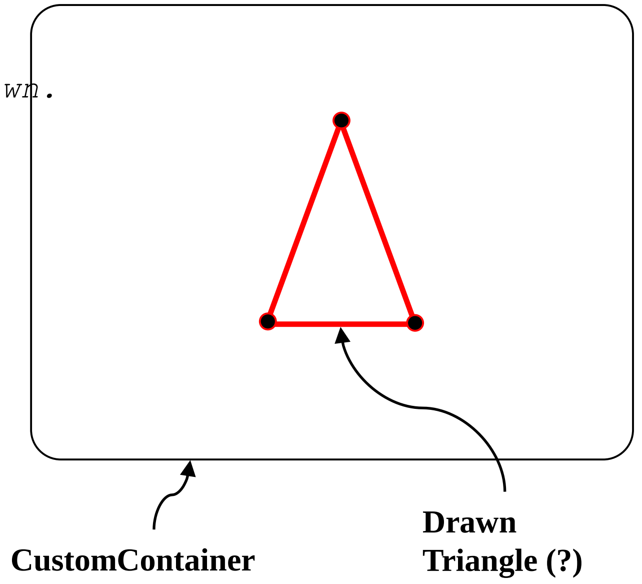
Triangle Class

```
/** This class defines an isosceles triangle with a specified base and height. The triangle
points are defined in "local space", and the local space axis orientation is X to the right
and Y upward. Local origin coincides with the container origin to draw the triangle on the
container. That is why, we pass "triangle point + pCmpRelPrnt" as a drawing coordinate to
the drawLine() method.*/
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int color ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    color = ColorUtil.BLACK;
}

  public void draw (Graphics g, Point pCmpRelPrnt) {
    g.setColor(color);
    g.drawLine (pCmpRelPrnt.getX()+ top.getX(), pCmpRelPrnt.getY()+top.getY(),
                pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY(),
                pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY(),
                pCmpRelPrnt.getX()+top.getX(),
                pCmpRelPrnt.getY()+top.getY());
  }
}
```



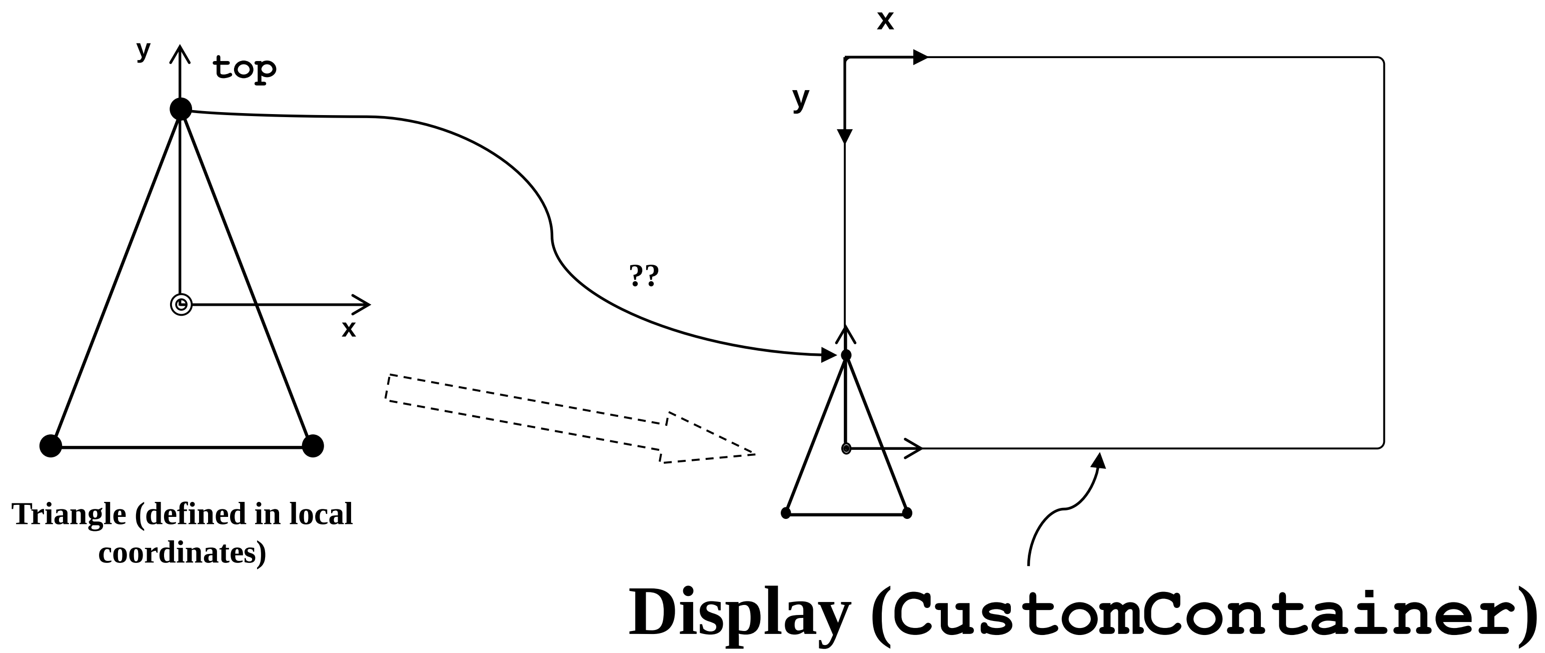
Drawing A Triangle

```
/** This class defines a container that has a triangle.
 *  Repainting the container causes the triangle to be drawn.
 */

public class CustomContainer extends Container{ {

  private Triangle myTriangle ;

  public CustomContainer () {
    myTriangle = new Triangle (200, 200) ;
  }

  public void paint (Graphics g) {
    super.paint (g);
    myTriangle.draw(g, new Point(this.getX(), this.getY())) ;
  }
}
```



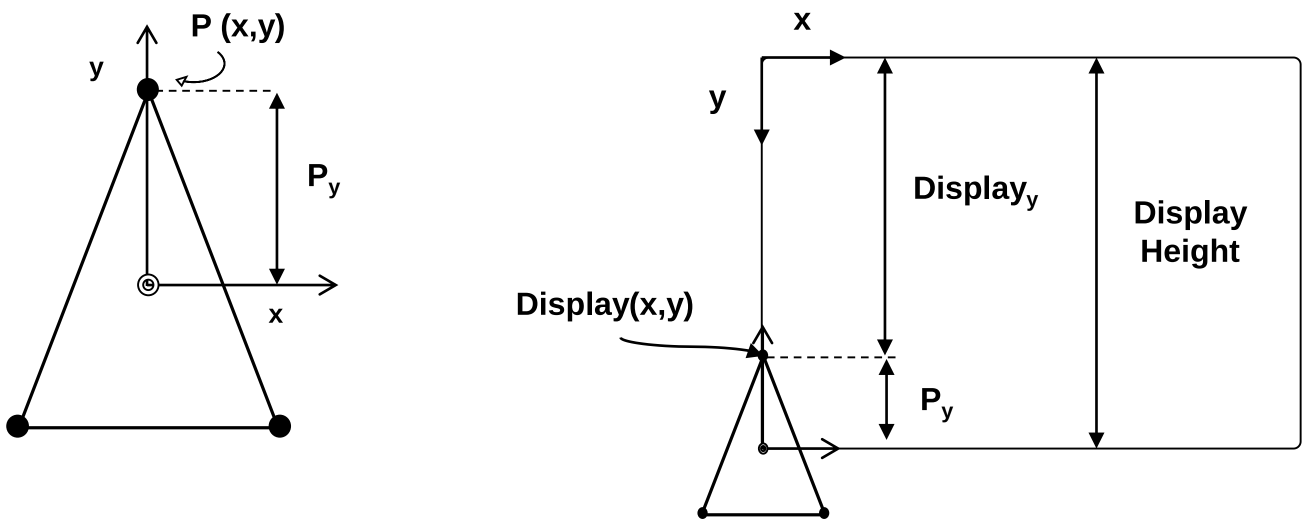
Mapping To Display Location

- Suppose desired location was “centered at lower-left display corner”
- How do we compute location of “top”?


y
top
x
??
x
y
Triangle (defined in local coordinates)
Display (CustomContainer)

Mapping To Display Location

(cont.)



- $Display_x = P_x$
- $Display_y = DisplayHeight - P_y$

$$= \underbrace{(-1*(P_y))}_{\texttt{Scale}_y\texttt{(-1)}} + \underbrace{DisplayHeight}_{\texttt{Translate}_y\texttt{(DisplayHeight)}}$$

Applying the Display Mapping

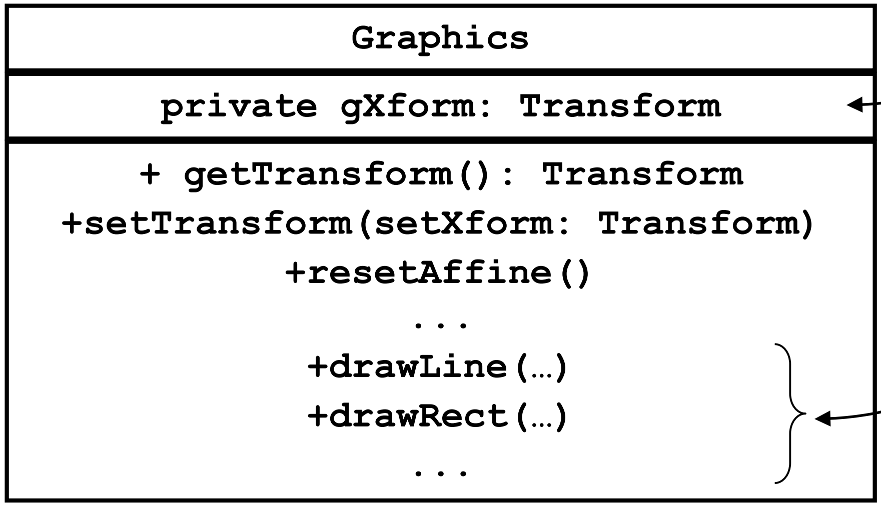
```
/** This class draws an Isosceles Triangle applying "display mapping"
 *  transformations to the triangle's points.
 */
public class Triangle {

  private float[] top, bottomLeft, bottomRight ;
  ...

  public void draw (Graphics g, Point pCmpRelPrnt, int height) {
    // create an displayXform to map triangle points to "display space"
    Transform displayXform = Transform.makeIdentity();
    displayXform.translate (0, height);
    displayXform.scale (1, -1);
    // apply the display mapping transforms to the triangle points
    displayXform.transformPoint(top,top);
    displayXform.transformPoint(bottomLeft,bottomLeft);
    displayXform.transformPoint(bottomRight,bottomRight);

    // draw the (transformed) triangle
    g.setColor(color);
    g.drawLine(pCmpRelPrnt.getX()+(int)top[0], pCmpRelPrnt.getY()+(int)top[1],
      pCmpRelPrnt.getX()+(int)bottomLeft[0],
      pCmpRelPrnt.getY()+(int)bottomLeft[1]); // left side
    g.drawLine(pCmpRelPrnt.getX()+(int)bottomLeft[0],
          pCmpRelPrnt.getY()+(int)bottomLeft[1], pCmpRelPrnt.getX()+
          (int)bottomRight[0], pCmpRelPrnt.getY()+ (int)bottomRight[1]); // bottom
    g.drawLine(pCmpRelPrnt.getX()+(int)bottomRight[0],
          pCmpRelPrnt.getY()+(int)bottomRight[1], pCmpRelPrnt.getX()+(int)top[0],
          pCmpRelPrnt.getY()+(int)top[1]); // right side
  }
}
```

Problems…

- Triangle flips between top and bottom of the display.
- Because the transformations <u>permanently alter</u> the triangle points.
- We could solve this by using *temporary variables* for the transformed points.
- There is a better solution which does not require us to transform the triangle points (this solution will allow us to directly use the points that are defined relative to the local origin).

The `Graphics` Class

- Every **`Graphics`** contains a **`Transform`** object
  - This **transform is applied to all drawing coordinates** during drawing


Graphics
private gXform: Transform
+ getTransform(): Transform
+setTransform(setXform: Transform)
+resetAffine()
...
+drawLine(…)
+drawRect(…)
...

gXform has the current xform of the `Graphics` object

All drawing methods apply current xform to drawing coordinates

Using **`Graphics`**’s Xform

- We can concatenate scale and translate associated with the display mapping to the current xform of the **`Graphics`** object. Then tell the triangle to draw itself using that **`Graphics`** object.

- This causes the specified scale and translate to be applied to the drawing coordinates when the triangle is drawn.

- To draw the triangle on display (**`CustomContainer`**), the local origin coincides with the display origin.

- Remember that this origin is positioned at **`(getX(), getY())`** relative to component’s parent container origin (origin of the content pane of the form) and point **`pCmpRelPrnt`** contains this position.

- That is why, a drawing coordinate is positioned at “triangle point + **`pCmpRelPrnt`**” relative to parent origin and we pass this value to the **`drawLine()`** method which expects coordinates relative to parent origin.

Using `Graphics's` Xform (contd.)

- However, since transformations are applied relative to the screen origin (i.e., coordinates passed to transformation methods are relative to screen origin), we first need to move the drawing coordinates so that local origin coincides with the screen origin.
- Remember that local origin (positioned at **`(getX(), getY())`** relative to component's parent container origin) is positioned at **`(getAbsoluteX(), getAbsoluteY())`** relative to the screen origin.
- Hence a drawing coordinate positioned at "triangle point + **`pCmpRelPrnt`**" relative to parent origin is located at "triangle point + **`pCmpRelScrn`**" relative to screen origin where points **`pCmpRelPrnt and pCmpRelScrn`** contains **`(getX(), getY())`** and **`(getAbsoluteX(), getAbsoluteY())`** values, respectively.
- That is why, before we apply scale and translate associated with display mapping, we need to move the drawing coordinates by
  `translate(-getAbsoluteX(), -getAbsoluteY())`
  (**`translate()`**, like other transformation methods, expects us to provide coordinates relative to the screen origin).

Using `Graphics`’s Xform (contd.)

- After applying display mapping we need to move the drawing coordinates back to where they were by **`translate(getAbsoluteX(), getAbsoluteY())`** so that we can draw the triangle on the display (CustomContainer).

- We call these translations related with moving the drawing coordinates back and forth (so that local origin coincides with screen origin before the display mapping is done) as “**local origin**” transformation.

- After triangle is drawn, we need to restore the original xform (the xform before the display mapping and local origin transformations are applied) of the **`Graphics`** object since graphics object is used for other operations after the **`paint()`** returns. **`resetAffine()`** method of **`Graphics`** class is used for this purpose.

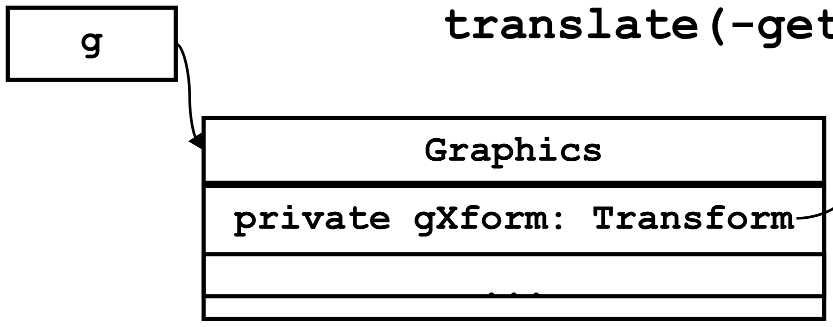
Using Graphics's Xform (cont.)

```
public class CustomContainer extends Container {
  private Triangle myTriangle ;
  public CustomContainer () {
    myTriangle = new Triangle (200, 200);
  }

  public void paint (Graphics g) {
    super.paint(g);
    Transform gXform = Transform.makeIdentity();
    g.getTransform(gXform);
    //move drawing coordinates back
    gXform.translate(getAbsoluteX(),getAbsoluteY());
    //apply translate associated with display mapping
    gXform.translate(0, getHeight());
    //apply scale associated with display mapping
    gXform.scale(1, -1);
    //move drawing coordinates so that the local origin coincides with the screen origin
    gXform.translate(-getAbsoluteX(),-getAbsoluteY());
    g.setTransform(gXform);
    myTriangle.draw(g, new Point(getX(), getY()));
    //restore the original xform in g
    g.resetAffine();
  }
}
```

Using `Graphics`'s Xform (cont.)

- Effect of modifying **`g`**'s transform in **`paint()`** :

```
translate(getAbsoluteX(),getAbsoluteY())
translate(0, getHeight());
scale(1,-1);
translate(-getAbsoluteX(),-getAbsoluteY())
```



gXform has the current xform of the Graphics object g.

Initially transformation matrix (TM) in gXform is not identity. It has a previous content [M]. After transformations are applied to gXform, TM would be equal to this:

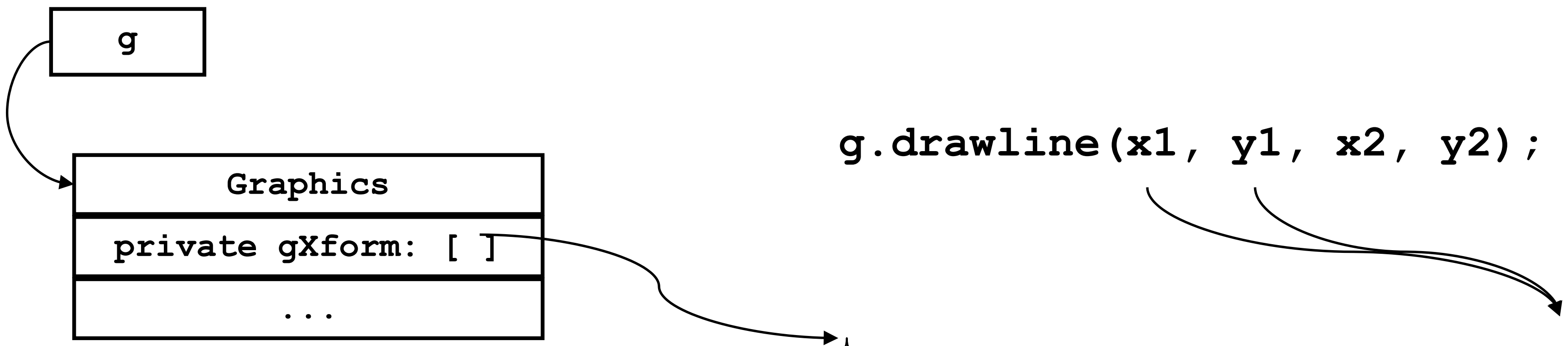
$$\underbrace{[M]}_{\text{Previous content}}\times[T(absX,absY)]\times[T_y(displayHeight)]\times[S_y(-1)]\times[T(-absX,-absY)]$$

Using Graphics's Xform (cont.)

```
/** This class defines a triangle, as before.
 *  The Graphics object applies its current xform to all drawing
 *  coordinates prior to performing any output operation.
 */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int color ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    color = ColorUtil.BLACK;
  }
  public void draw (Graphics g, Point pCmpRelPrnt) {
    g.setColor(color);
    g.drawLine (pCmpRelPrnt.getX()+ top.getX(), pCmpRelPrnt.getY()+top.getY(),
                pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY(),
                pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY(),
                pCmpRelPrnt.getX()+top.getX(),
                pCmpRelPrnt.getY()+top.getY());
  }
}
```

Using Graphics's Xform (cont.)

- Effect of using **`g`** to draw a line in
  `Triangle.draw()`:

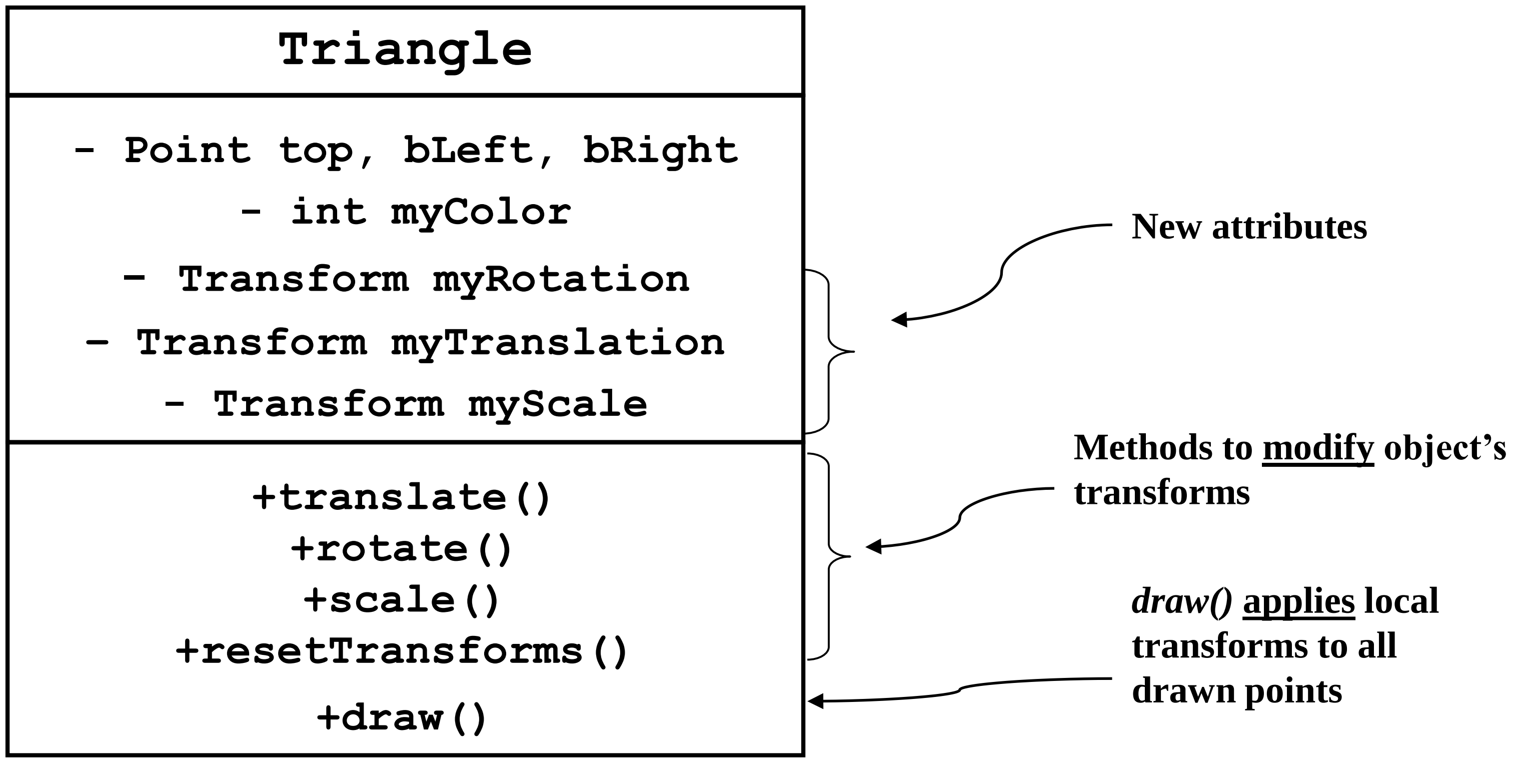


```
g.drawline(x1, y1, x2, y2);
```

$$\underbrace{\begin{bmatrix}x'\\y'\\1\end{bmatrix}}_{\text{Coordinate actually used in drawing}} = \overbrace{[M]\times[T\,(absX,absY)]\times[T_y(displayHeight)]\times[S_y(-1)]\times[T\,(-absX,-absY)]}\times\underbrace{\begin{bmatrix}x\\y\\1\end{bmatrix}}_{\text{Drawing coordinate passed to drawLine()}}$$

Transformable Objects

- Expand objects to contain *"local transforms" (LTs)*
- Arrange to *apply an object's transforms* when it is drawn


Triangle
- Point top, bLeft, bRight
- int myColor
- Transform myRotation
- Transform myTranslation
- Transform myScale
+translate()
+rotate()
+scale()
+resetTransforms()
+draw()
New attributes
Methods to modify object's transforms
draw() applies local transforms to all drawn points

```
/** This class defines a triangle with Local Transformations (LTs). Client
 *  code can apply arbitrary transformations to the triangle by invoking methods to
 *  update/modify the LTs; when the triangle is drawn it automatically
 *  applies its current LTs to drawing coordinates. */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int myColor ;
  private Transform myRotation, myTranslation, myScale ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    myColor = ColorUtil.BLACK ;
    myRotation = Transform.makeIdentity();
    myTranslation = Transform.makeIdentity();
    myScale = Transform.makeIdentity();
  }
  public void rotate (float degrees) {
    //pivot point (rotation origin) is (0,0), this means the rotation will be applied about
    //the screen origin
    myRotation.rotate ((float)Math.toRadians(degrees),0,0);
  }

  public void translate (float tx, float ty) {
    myTranslation.translate (tx, ty);
  }

  public void scale (float sx, float sy) {

    //remember that like other transformation methods, scale() is also applied relative to
    //screen origin
    myScale.scale (sx, sy);
  }
  //...continued...
```

Transformable Objects (cont.)

```
  // ... Triangle class, cont.

 public void resetTransform() {
  myRotation.setToIdentity();
  myTranslation.setToIdentity();
  myScale.setToIdentity();
  }
/* This method applies the triangle's LTs to the received Graphics object's xform, then uses this
xform (with the additional transformations) to draw the triangle. Note that we pass getAbsoluteX()
and getAbsoluteY() values of the container as pCmpRelScrn*/
 public void draw (Graphics g, Point pCmpRelPrnt, Point pCmpRelScrn) {
  // set the drawing color for the triangle
  g.setColor(myColor);
  //append the triangle's LTs to the xform in the Graphics object. But first move the drawing
  //coordinates so that the local origin coincides with the screen origin. After LTs are applied,
  //move the drawing coordinates back.
  Transform gXform = Transform.makeIdenity();
  g.getTransform(gXform);
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.concatenate(myRotation);
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the lines as before
  g.drawLine(pCmpRelPrnt.getX()+top.getX(), pCmpRelPrnt.getY()+top.getY(),
    pCmpRelPrnt.getX() + bottomLeft.getX(),pCmpRelPrnt.getY() + bottomLeft.getY());
  //...[draw the rest of the lines]
  }} //end of Triangle class
```

```
/** This class defines a container containing a triangle.  It applies a simple set of
transformations to the triangle (by calling the triangle's transformation methods when the
triangle is created).  The container's paint() method applies the "display mapping"
transformation to the Graphics object, and tells the triangle to "draw itself".  The triangle
applies its LTs to the Graphics object in its draw() method.
 */
public class CustomContainer extends Container {
 private Triangle myTriangle ;
 public CustomContainer () {
  myTriangle = new Triangle (200, 200) ;     //construct a Triangle
  //apply some transformations to the triangle
  myTriangle.translate (300, 300);
  myTriangle.rotate (90);
  myTriangle.scale (2, 1);
  }
 public void paint (Graphics g) {
  super.paint (g);
  //...[apply the "Display mapping" transformation to the Graphics object as before. But,
  //again as before, first move the drawing coordinates so that the local origin coincides with
  //the screen origin. After display mapping is applied, move the drawing coordinates back.]

  //origin location of the component (CustomContainer) relative to its parent container origin
  Point pCmpRelPrnt = new Point(getX(),getY());
  //origin location of the component (CustomContainer) relative to the screen origin
  Point pCmpRelScreen = new Point(getAbsoluteX(),getAbsoluteY());
  //tell the triangle to draw itself
  myTriangle.draw(g, pCmpRelPrnt, pCmpRelScreen);
  }
}
```

Composite Transforms

- Transformations applied to triangle’s drawing coordinates:

$$\underset{\substack{\uparrow\\\textbf{Coordinate}\\\textbf{actually}\\\textbf{used in}\\\textbf{drawing}}}{\begin{bmatrix}x'\\y'\\1\end{bmatrix}} = \underbrace{\begin{bmatrix}\quad M \quad\end{bmatrix}}_{\substack{\textbf{Original gXform}\\\textbf{content}}} \times \underbrace{\begin{bmatrix}\quad T_{display} \quad\end{bmatrix} \times \begin{bmatrix}\quad S_{display} \quad\end{bmatrix}}_{\substack{\textbf{Display mapping}\\\textbf{transformations}}} \times \underbrace{\begin{bmatrix}\quad T_{tri} \quad\end{bmatrix} \times \begin{bmatrix}\quad R_{tri} \quad\end{bmatrix} \times \begin{bmatrix}\quad S_{tri} \quad\end{bmatrix}}_{\substack{\textbf{Triangle's local}\\\textbf{transformations}}} \times \underset{\substack{\uparrow\\\textbf{drawing}\\\textbf{coordinate}\\\textbf{passed to}\\\textbf{drawLine()}}}{\begin{bmatrix}x\\y\\1\end{bmatrix}}$$

$$\xleftarrow[\textbf{Order of application of transformations}]{\hspace{20em}}$$

Also called the “Graphics Transform Stack”

<u>Note:</u> there are also translations applied before and after “display mapping” and “local” transformations which belong to the “local origin” transformations. For brevity, they are not indicated in the above formula.

On Transform Order and Number of LTs

- Suppose an interactive program implements:

  Click = translate (10,10),   Drag = rotate (45°)

- "Suppose" the expected result (want) for the interactive sequence "Drag_1, Click_1, Drag_2, Click_2" is:
  - Rotation by a total of 90°, Translation by a total of (20,20)

(One might instead want the transformations applied "in sequence", but suppose that is not what we want here…)

- If we only have one LT object, after the above interaction it would look like:

$$[I]\times[R_1(45)]\times[T_1(10,10)]\times[R_2(45)]\times[T_2(10,10)]$$

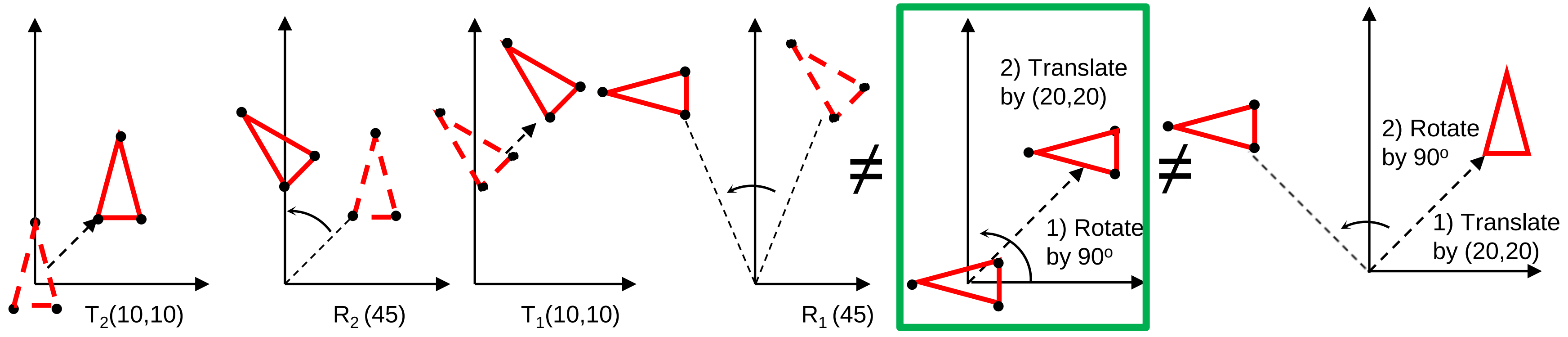
(by default xform is an identity matrix and it is modified by multiplications ***on the right***)

On Transform Order and Number of LTs (cont.)

- When LT is applied to the points defined in the local coordinates, it has the following effect:

$$[I] \times [R_1(45)] \times [T_1(10,10)] \times [R_2(45)] \times [T_2(10,10)] \times \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

(multiply ***from right to left***: last transform is applied first)

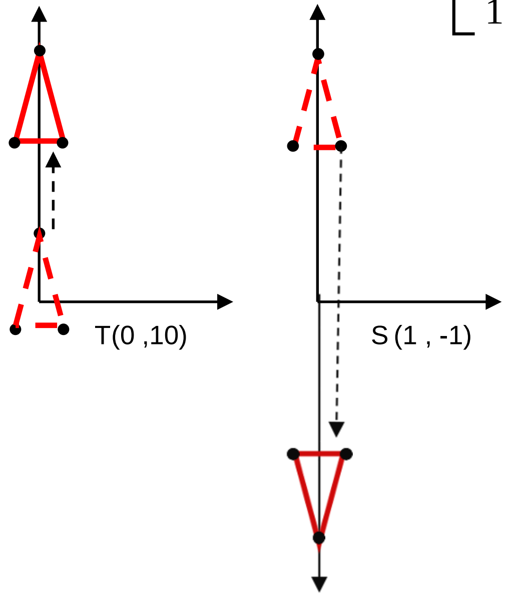


- So to get the expected result we need to accumulate translations and rotations in two separate LTs and rotate the points before translating them (just like the above mentioned Triangle class).

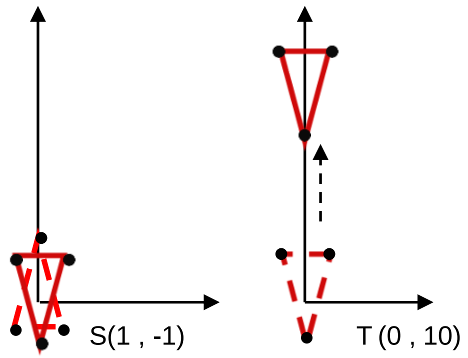
- When we apply scale (e.g., before or after translation) is also equally important…

On Transform Order and Number of LTs (cont.)

$$[I]\times[S(1,-1)]\times[T(0,10)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix} \neq [I]\times[T(0,10)]\times[S(1,-1)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix}$$



$\neq$

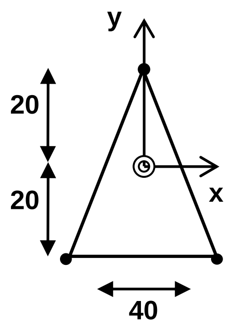


If Click = translate (0,10), Drag = scale (1, 2) and the expected result for "Drag$_1$, Click$_1$, Drag$_2$, Click$_2$" is: "Scaling the height of triangle by x4, and Translation by a total of (0,20)"

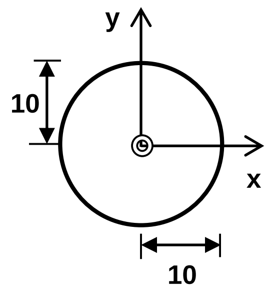
Then we should have a separate transform for accumulating scaling transformations too…(Then we would use these separate LTs, in a way that the points would be scaled before they are translated. If we use a single LT, the height would still be scaled by x4, but the triangle would be translated more than 20 units along the Y axis.)

Hierarchical Objects

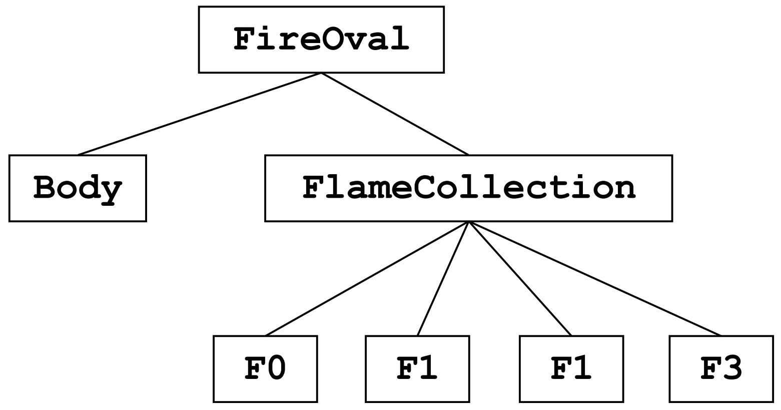
- We can build an object by combining
  - Simpler "parts"
  - Transformations to "orient" the parts



A "Flame" object



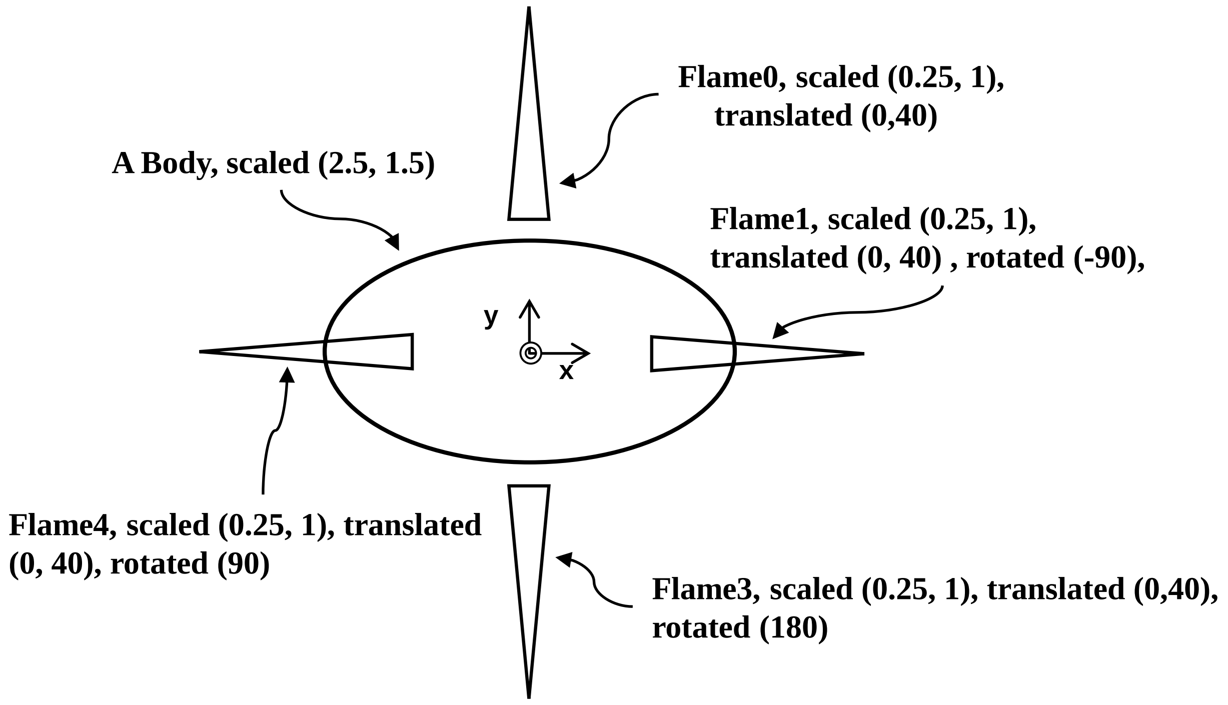
A "Body" object



A hierarchical "FireOval" object

Hierarchical Objects (cont.)

- FireOval Transformations


Flame0, scaled (0.25, 1), translated (0,40)
A Body, scaled (2.5, 1.5)
Flame1, scaled (0.25, 1), translated (0, 40) , rotated (-90),
y
x
Flame4, scaled (0.25, 1), translated (0, 40), rotated (90)
Flame3, scaled (0.25, 1), translated (0,40), rotated (180)

A hierarchical "FireOval" object

Then we scale the FireOval object with (2, 2) and rotate with 45 degrees and translate it by (400, 200) and apply "display mapping" and "local origin" transformations to it!

Hierarchical Objects (cont.)

```
/** Defines a single "flame" to be used as an arm of a FireOval.
 *  The Flame is modeled after the "Triangle" class, but specifies
 *  fixed dimensions of 40 (base) by 40 (height) in local space.
 *  Clients using the Flame can scale it to have any desired proportions.
 */
public class Flame {
    private Point top, bottomLeft, bottomRight ;
    private int myColor ;
    private Transform myTranslation ;
    private Transform myRotation ;
    private Transform myScale ;

    public Flame (){
        // define a default flame with base=40, height=40, and origin in the center.
        top = new Point (0, 20);
        bottomLeft = new Point (-20, -20);
        bottomRight = new Point (20, -20);
        // initialize the transformations applied to the Flame
        myTranslation = Transform.makeIdentity();
        myRotation = Transform.makeIdentity();
        myScale = Transform.makeIdentity();
    }
    public void setColor(int iColor){
        myColor = iColor;
    }
    //...continued
```

```
  // Flame class, continued...
public void rotate (double degrees)       {
  myRotation.rotate (Math.toRadians(degrees), 0, 0);}
public void scale (double sx, double sy) {
  myScale.scale (sx, sy);}
public void translate (double tx, double ty) {
  myTranslation.translate (tx, ty);}

public void draw (Graphics g, Point pCmpRelPrnt, Point pCmpRelScrn) {
  //append the flames's LTs to the xform in the Graphics object (do not forget to do "local
  //origin" transformations). ORDER of LTs: Scaling LT will be applied to coordinates FIRST,
  //then Translation LT, and lastly Rotation LT. Also restore the xform at the end of draw() to
  //remove this sub-shape's LTs from xform of the Graphics object. Otherwise, we would also
  //apply these LTs to the next sub-shape since it also uses the same Graphics object.
  Transform gXform = Transform.makeIdentity();
  g.getTransform(gXform);
  Transform gOrigXform = gXform.copy(); //save the original xform
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.concatenate(myRotation); ← Rotation is LAST
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the lines as before
  g.drawLine(pCmpRelPrnt.getX()+top.getX(), pCmpRelPrnt.getY()+top.getY(),
    pCmpRelPrnt.getX() + bottomLeft.getX(),pCmpRelPrnt.getY() + bottomLeft.getY());
  //...[draw the rest of the lines]
  g.setTransform(gOrigXform); //restore the original xform (remove LTs)
  //do not use resetAffine() in draw()! Instead use getTransform()/setTransform(gOrigForm)
  }
} // end of Flame class
```

```
/** Defines a "Body" for a FireOval; the "body" is just a scalable circle with its origin in the
center. Lower left corner in local space would correspond to upper left corner on screen */
public class Body {
  private int myRadius, myColor ;
  private Transform myTranslation, myRotation, myScale ;
 public Body () {
   myRadius = 10;
   Point lowerLeftInLocalSpace = new Point(-myRadius, -myRadius);
   myColor = Color.yellow ;
   myTranslation = Transform.makeIdentity();
   myRotation = Transform.makeIdentity();
   myScale = Transform.makeIdentity(); }
  // ...[code here implementing rotate(), scale(), and translate() as in the Flame class]
 public void draw (Graphics g , Point pCmpRelPrnt, Point pCmpRelScrn) {
  g.setColor(myColor);
  Transform gXform = Transform.makeIdentity();
  g.getTransform(gXform);
  Transform gOrigXform = gXform.copy(); //save the original xform
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.concatenate(myRotation); ← Rotation is not LAST
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the body
  g.fillArc( pCmpRelPrnt.getX() + lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY() + lowerLeftInLocalSpace.getY(),
             2*myRadius, 2*myRadius, 0, 360);
  g.setTransform(gOrigXform); //restore the original xform
  }}
```

```
/** This class defines a "FireOval", which is a hierarchical object composed
 *  of a scaled "Body" and four scaled, translated, and rotated "Flames".
 */
public class FireOval {
  private Body myBody ;
  private Flame [] flames ;
  private Transform myTranslation, myRotation, myScale ;
  public FireOval () {
    myTranslation = Transform.makeIdentity();
    myRotation = Transform.makeIdentity();
    myScale = Transform.makeIdentity();
    myBody = new Body();           // create a properly-scaled Body for the FireOval
    myBody.scale(2.5, 1.5);
    flames = new Flame [4];        // create an array to hold the four flames
    // create four flames, each scaled, translated “up” in Y, and then rotated
    // relative to the local origin
    Flame f0 = new Flame();  f0.translate(0, 40);   f0.scale (0.25, 1);
    flames[0] = f0 ;   f0.setColor(ColorUtil.BLACK);
    Flame f1 = new Flame(); f1.translate(0, 40);f1.rotate(-90);f1.scale(0.25, 1);
    flames[1] = f1 ;   f1.setColor(ColorUtil.GREEN);
    Flame f2 = new Flame(); f2.translate(0, 40);f2.rotate(180);f2.scale(0.25, 1);
    flames[2] = f2 ;   f2.setColor(ColorUtil.BLUE);
    Flame f3 = new Flame(); f3.translate(0, 40);f3.rotate(90);f3.scale(0.25, 1);
    flames[3] = f3;    f3.setColor(ColorUtil.MAGENTA);
  }
  // continued...
```

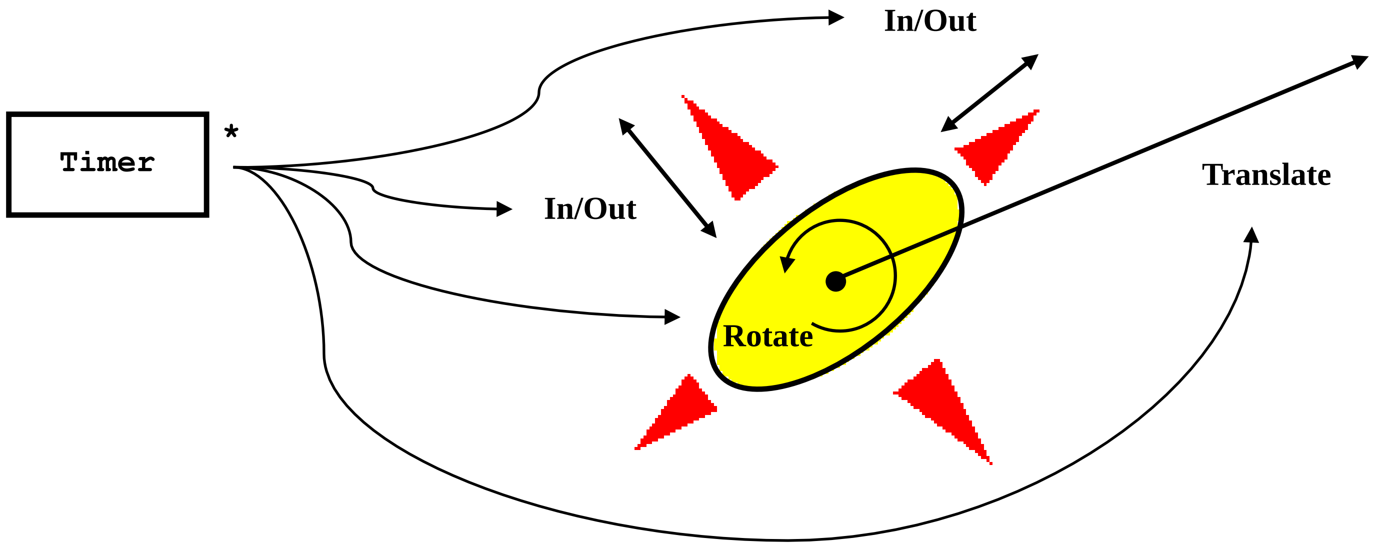
Hierarchical Objects (cont.)

```
   // FirebOval class, continued...
   // ...[code here implementing rotate(), scale(), and translate() as in the Flame class]
public void draw (Graphics g) {
   Transform gXform = Transform.makeIdentity();
   g.getTransform(gXform);
   Transform gOrigXform = gXform.copy(); //save the original xform
   //move the drawing coordinates back
   gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
   // append FireOval's LTs to the graphics object's transform
   gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
   gXform.concatenate(myRotation);
   gXform.scale(myScale.getScaleX(), myScale.getScaleY());
   //move the drawing coordinates so that the local origin coincides with the screen origin
   gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
   g.setTransform(gXform);
   //draw sub-shapes of FireOval
   myBody.draw(g, pCmpRelPrnt, pCmpRelScrn);
   for (Flame f : flames) {
        f.draw(g, pCmpRelPrnt, pCmpRelScrn);
     }
   g.setTransform(gOrigXform);//restore the original xform
   }
} //end of FireOval class
```

```
/** This class displays a "FireOval" object, scaling, rotating, and translating and it into
position on the screen, and telling it to draw itself. Note that CustomContainer object is
created by a form. Code for the form is not provided. It basically sets up GUI using border
layout,adds buttons to north, south, and west containers, and CustomContainer object to center.*/
public class CustomContainer extends Container {
  FireOval myFireOval ;
  public CustomContainer () {
    // create a FireOval to display
    myFireOval = new FireOval ();
    // rotate, scale, and translate this FireOval on the container
    myFireOval.scale(2,2);
    myFireOval.rotate (45) ;
    myFireOval.translate (400, 200) ;     }
  public void paint (Graphics g) {
    super.paint (g);
    Transform gXform = Transform.makeTransform();
    g.getTransform(gXform);
    //move the drawing coordinates back
    gXform.translate(getAbsoluteX(),getAbsoluteY());
    //apply display mapping
    gXform.translate(0, getHeight());
    gXform.scale(1, -1);
    //move the drawing coordinates as part of the "local origin" transformations
    gXform.translate(-getAbsoluteX(),-getAbsoluteY());
    g.setTransform(gXform);
    Point pCmpRelPrnt = new Point(this.getX(), this.getY());
    Point pCmpRelScrn = new Point(getAbsoluteX(),getAbsoluteY());
    // tell the fireball to draw itself
    myFireOval.draw(g, pCmpRelPrnt, pCmpRelScrn);
    g.resetAffine(); //restore the xform in Graphics object
  } } //do not use getTransform()/setTranform(gOrigXform) in paint()!
      //instead use resetAffine()
```

Dynamic Transformations

- We can alter an object's transforms "on-the-fly"
  - Vary sub-shapes (i.e., body and flames) local transforms
  - Vary entire object (i.e., FireOval) local transforms



Dynamic Transformations (cont.)

```
/** This class defines a Form containing a CustomContainer object that displays
 * the FireOval. It uses a Timer to call updateLTs() which modify FireOval’s and
 * its Flames’ local transformations.
 * CustomContainer class looks exactly like the one used in static FireOval
 * example expect it also has a getFireOval() method that returns FireOval object.
 */
public class DynamicFireOvalForm extends Form implements Runnable {
   private CustomContainer myCustomContainer = new CustomContainer();

   public DynamicFireOvalForm () {
      //...[set up GUI using border layout, add buttons to north, south, and
      //west containers, and CustomContainer object to the center container.]
      UITimer timer = new UITimer(this);
      timer.schedule(10, true, this);
   }
   public void run () {
      myCustomContainer.getFireOval().updateLTs() ;
      myCustomContainer.repaint() ;
   }
```

Dynamic Transformations (cont.)

```
/** This class defines a FireOval object which supports dynamic alteration
 *  of both the FireOval position & orientation, and also of the offset of
 *  the flames from the body.
 */
public class FireOval {
    //...declarations here for Body, Flames, and FireOval transforms, as before;
    // and code here to define the FireOval body and flames, and to define
    // methods for applying transformations, as before...draw() method is as before too…
    private double flameOffset = 0 ;          // current flame distance from FireOval
    private double flameIncrement = 1 ;       // change in flame distance each tick
    private double maxFlameOffset = 10 ;      // max distance before reversing

    // Invoked to update the local transforms of FireOval and its sub-shapes, flames.
    public void updateLTs () {
    // update the FireOval position and orientation
    this.translate(1,1);
    this.rotate(1) ;
    // update the flame positions (move them along their local Y axis)
    // this is why flames are TRANSLATED before they are ROTATED
    for (Flame f:flames) {
        f.translate ((float)0, (float)flameIncrement);
    }
    flameOffset += flameIncrement ;
    // reverse direction of flame movement for next time if we've hit the max
    if (Math.abs(flameOffset) >= maxFlameOffset) {
        flameIncrement *= -1 ;
    } }
```