Understanding Shapes

- The simplest way to draw 2-D graphical content in a WPF user interface is to use shapes—dedicated classes that represent simple lines, ellipses, rectangles, and polygons.
- Shapes are known as drawing primitives.
Understanding Shapes

Shapes are elements. This has the following important consequences:

1. Shapes draw themselves. You don’t need to manage the invalidation and painting process.
2. Shapes are organized in the same way as other elements. In other words, you can place a shape in any of the layout containers.
3. Shapes support the same events as other elements. That means you don’t need to go to any extra work to deal with focus, key presses, mouse movements, and mouse clicks.

The Shape Classes

Every shape derives from the abstract System.Windows.Shapes.Shape class.

Although the Shape class can’t do anything on its own, it defines a small set of important properties, which are listed in Table 12-1.
Rectangle and Ellipse

- To create either one, set the familiar **Height** and **Width** properties (inherited from FrameworkElement) to define the size of your shape,
- And then set the **Fill** or **Stroke** property (or both) to make the shape visible.
- You’re also free to use properties such as **MinHeight**, **MinWidth**, **HorizontalAlignment**, **VerticalAlignment**, and **Margin**.
- **Note** If you fail to set the Stroke or Fill property, your shape won’t appear at all.

```xml
<StackPanel>
  <Ellipse Fill="Yellow" Stroke="Blue"
  Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Ellipse>
  <Rectangle Fill="Yellow" Stroke="Blue"
  Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Rectangle>
</StackPanel>
```
The Ellipse class doesn’t add any properties. The Rectangle class adds just two: RadiusX and RadiusY.

When set to nonzero values, these properties allow you to create nicely rounded corners.

You can think of $\text{RadiusX}$ and $\text{RadiusY}$ as describing an ellipse that’s used just to fill in the corners of the rectangle.

For example, if you set both properties to 10, WPF draws your corners by using the edge of a circle that’s 10 units wide.
Rectangle and Ellipse

- If you don’t supply the Height and Width properties, the shape is sized based on its container.
- A better example can be made with the Grid container.

```xml
<Grid>
  <Ellipse Fill="Yellow" Stroke="Blue"></Ellipse>
</Grid>
```

- Here, the Grid fills the entire window. The Grid contains a single proportionately sized row, which fills the entire Grid. Finally, the ellipse fills the entire row.

---

Rectangle and Ellipse

- So far, you’ve seen how to size a Rectangle and an Ellipse, but what about placing them exactly where you want them?
- The ideal container is the Canvas, which forces you to specify the coordinates of each shape by using the attached Left, Top, Right, or Bottom properties.
- This gives you complete control over how shapes overlap:

```xml
<Canvas>
  <Ellipse Fill="Yellow" Stroke="Blue" Canvas.Left="100" Canvas.Top="50" Width="100" Height="50"></Ellipse>
  <Rectangle Fill="Yellow" Stroke="Blue" Canvas.Left="30" Canvas.Top="40" Width="100" Height="50"></Rectangle>
</Canvas>
```
**Line**

- The Line shape represents a straight line that connects one point to another.
- The starting and ending points are set by four properties: X1 and Y1 (for the first point) and X2 and Y2 (for the second point).
  
  ```xml
  <Line Stroke="Blue" X1="0" Y1="0" X2="10" Y2="100"></Line>
  ```
- The Fill property has no effect for a line. You must set the Stroke property.
- There's no way to create a curved line with Line or Polyline shapes. Instead, you need the more advanced Path class described in Chapter 13.

**Polyline**

- The Polyline class allows you to draw a sequence of connected straight lines.
- You simply supply a list of X and Y coordinates by using the Points property.
- Technically, the Points property requires a `PointCollection` object, but you fill this collection in XAML by using a lean string-based syntax.
- You simply need to supply a list of points and add a space or a comma between each coordinate.
The **Polygon** is virtually the same as the **Polyline**. Like the **Polyline** class, the **Polygon** class has a Points collection that takes a list of coordinates.

- The only difference is that the **Polygon** adds a final line segment that connects the final point to the starting point.
- You can fill the interior of this shape by using the Fill brush.
Using Transforms

- Many drawing tasks can be made simpler with the use of a **transform** an object that alters the way a shape or element is drawn by quietly shifting the coordinate system it uses.
- In WPF, transforms are represented by classes that derive from the abstract System.Windows.Media.Transform class, as listed in Table 12-5.

### Table 12-5. Transform Classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Important Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>TranslateTransform</td>
<td>Displaces your coordinate system by some amount. This transform is useful if you want to draw the same shape in different places.</td>
<td>X, Y</td>
</tr>
<tr>
<td>RotateTransform</td>
<td>Rotates your coordinate system. The shapes you draw normally are turned around a center point you choose.</td>
<td>Angle, CenterX, CenterY</td>
</tr>
<tr>
<td>ScaleTransform</td>
<td>Scales your coordinate system up or down, so that your shapes are drawn smaller or larger. You can apply different degrees of scaling in the X and Y dimensions, thereby stretching or compressing your shape.</td>
<td>ScaleX, ScaleY, CenterX, CenterY</td>
</tr>
<tr>
<td>SkewTransform</td>
<td>Warps your coordinate system by slanting it a number of degrees. For example, if you draw a square, it becomes a parallelogram.</td>
<td>AngleX, AngleY, CenterX, CenterX</td>
</tr>
<tr>
<td>MatrixTransform</td>
<td>Modifies your coordinate system by using matrix multiplication with the matrix you supply. This is the most complex option; it requires some mathematical skill.</td>
<td>Matrix</td>
</tr>
<tr>
<td>TransformGroup</td>
<td>Combines multiple transforms so they can all be applied at once. The order in which you apply transformations is important because it affects the final result. For example, rotating a shape (with RotateTransform) and then moving it (with TranslateTransform) sends the shape off in a different direction than if you move it and then rotate it.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Using Transforms

- Technically, all transforms use matrix math to alter the coordinates of your shape.
- However, using the prebuilt transforms such as TranslateTransform, RotateTransform, ScaleTransform, and SkewTransform is far simpler than using the MatrixTransform and trying to work out the correct matrix for the operation you want to perform.
- When you perform a series of transforms with the TransformGroup, WPF fuses your transforms together into a single MatrixTransform, ensuring optimal performance.

Using Transforms

- Transforms are one of those quirky concepts that turn out to be extremely useful in a variety of contexts. Some examples include the following:
  - Angling a shape: So far, you've been stuck with horizontally aligned rectangles, ellipses, lines, and polygons. Using the RotateTransform, you can turn your coordinate system to create certain shapes more easily.
  - Repeating a shape: Many drawings are built using a similar shape in several places. Using a transform, you can take a shape and then move it, rotate it, resize it, and so on.
  - Animation: You can create sophisticated effects with the help of a transform, such as by rotating a shape, moving it from one place to another, and warping it dynamically.
To transform a shape, you assign the RenderTransform property to the transform object you want to use.

Depending on the transform object you’re using, you’ll need to fill in different properties to configure it, as detailed in Table 12-5.

For example, if you’re rotating a shape, you need to use the RotateTransform, and supply the angle in degrees. Here’s an example that rotates a square by 25 degrees:

```xml
<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow">
  Canvas.Left="100" Canvas.Top="100">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>
```
Transforming Shapes

- Sometimes you’ll want to rotate a shape around a different point.
- The RotateTransform, like many other transform classes, provides a `CenterX` property and a `CenterY` property.

```xml
<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow"
    Canvas.Left="100" Canvas.Top="100">
    <Rectangle.RenderTransform>
        <RotateTransform Angle="25" CenterX="45" CenterY="5" />
    </Rectangle.RenderTransform>
</Rectangle>
```

Transforming Shapes

- There’s a clear limitation to using the `CenterX` and `CenterY` properties of the RotateTransform.
- These properties are defined using absolute coordinates, which means you need to know the exact center point of your content.
- This property sets the center point by using a proportional coordinate system that stretches from 0 to 1 in both dimensions. In other words, the point (0, 0) is designated as the top-left corner and (1, 1) is the bottom-right corner.
Transforming Shapes

<Rectangle Width="80" Height="10" Stroke="Blue" Fill="Yellow"
Canvas.Left="100" Canvas.Top="100" RenderTransformOrigin="0.5,0.5">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="25" />
  </Rectangle.RenderTransform>
</Rectangle>