WPF Shapes
Shapes are elements

Shapes draw themselves, no invalidation or repainting needed when shape moves, window is resized, or shape’s properties change.

Shapes are organized like other elements. Place a shape in any of the layout containers, Canvas is useful because it allows shapes to be placed at specific coordinates.

Shapes support the same events as other elements: focus, key presses, mouse movements, mouse clicks, tooltips, context menus, and drag-and-drop operations.

Here are examples on how to draw Rectangle and Ellipse.
## Shape Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fill</strong></td>
<td>Sets brush that paints the surface of the shape</td>
</tr>
<tr>
<td><strong>Stroke</strong></td>
<td>Sets brush object that paints the shape's border</td>
</tr>
<tr>
<td><strong>StrokeThickness</strong></td>
<td>Sets the thickness of the border. WPF splits the width on each side; value of 10 units wide gets 5 units of space on each side.</td>
</tr>
<tr>
<td><strong>StrokeStartLineCap</strong></td>
<td>Determine the contour of the edge of the beginning and end of the line. These properties have an effect only for the Line, the Polyline, and (sometimes) the Path shapes.</td>
</tr>
<tr>
<td><strong>StrokeEndLineCap</strong></td>
<td></td>
</tr>
<tr>
<td><strong>StrokeDashArray,</strong></td>
<td>Create a dashed border around shape, set the size and frequency of the dashes, and the contour of the edge where each dash liner begins and ends.</td>
</tr>
<tr>
<td><strong>StrokeDashOffset,</strong></td>
<td></td>
</tr>
<tr>
<td><strong>StrokeDashCap</strong></td>
<td>Determine the contour of the shape’s corners</td>
</tr>
<tr>
<td><strong>StrokeLineJoin</strong></td>
<td></td>
</tr>
<tr>
<td><strong>StrokeMiterLimit</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Stretch</strong></td>
<td>Determines how a shape fills its available space. Creates a shape that expands to fit its container. A shape that expands in one direction by using Horizontal or VerticalAlignment properties.</td>
</tr>
<tr>
<td><strong>DefiningGeometry</strong></td>
<td>Provides a Geometry object for the shape. A Geometry object describes the coordinates and size of a shape without including the UIElement support for keyboard and mouse events.</td>
</tr>
<tr>
<td><strong>GeometryTransform</strong></td>
<td>Apply a Transform: skew, rotate, or displace a shape.</td>
</tr>
<tr>
<td><strong>RenderedGeometry</strong></td>
<td>Geometry object that describes the rendered shape.</td>
</tr>
</tbody>
</table>

No Stroke or Fill property ➔ your shape won’t appear.
Stretch enumeration

**Fill**  
Shape is stretched in width and height to fit its container exactly. (If you set an explicit height and width, this setting has no effect.)

**None**  
The shape is not stretched. Unless you set a nonzero width and height (using the Height and Width or MinHeight and MinWidth properties), your shape won’t appear.

**Uniform**  
The width and height are sized up proportionately until the shape reaches the edge of the container. With explicit height and width, shape is sized within those bounds.

**UniformToFill**  
The width and height are sized proportionately until the shape fills all the a variable height and width.

Usually size a shape explicitly or allow it to stretch to fit.
**Brushes**

Brushes fill an area:
- background, foreground, or border of an element
- fill or stroke of a shape.

Brushes have change notification, on change any element that uses the brush repaints themself.

Brushes support partial transparency.

SystemBrushes class provides access to brushes using colors defined in the Windows system preferences for the current computer.

The simplest type of brush is SolidColorBrush ...

```csharp
/// Assume cmd is an element (aka Button)

cmd.Background = new SolidColorBrush(Colors.AliceBlue);

int red = 0; int green = 255; int blue = 0;

cmd.Foreground = new SolidColorBrush(Color.FromRgb(red, green, blue));
```
Mouse Canvas Demo

MessageBox shown on rightButtonDown on "cast test" button

MessageBox shown on rightButtonUp event

releaseEllipse:
   sender = System.Windows.Controls.Canvas
   source = System.Windows.Shapes.Ellipse
   original source = System.Windows.Shapes.Ellipse

canvas children = 5
   redEllipse
   greenEllipse
   blueEllipse
   cyanEllipse
   aButton

VisualTree children = 5
   blueEllipse
   aButton
   greenEllipse
   cyanEllipse
   redEllipse

OK
<Window x:Class="MouseCanvas.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Mouse Canvas Demo: right button drags, or just move mouse"
Height="300" Width="485">
    <DockPanel x:Name="dockPanel" LastChildFill="true"
        Margin="0,0,0,-5">
        <Label x:Name="statusLabel" Content="Drag or move the mouse in the canvas ..." VerticalAlignment="Bottom"
            DockPanel.Dock="Bottom" FontSize="16"
            BorderBrush="Black" IsEnabled="False" Height="39"/>
        ...  // canvas xaml next page should be inserted here
    </DockPanel>
</Window>
<Canvas x:Name="canvas" MouseMove="mouseMove"
MouseRightButtonDown="selectEllipse"
MouseRightButtonDownUp="releaseEllipse">
  <Ellipse x:Name="redEllipse" Fill="Red" Height="100"
    Canvas.Left="111" Stroke="Black" Canvas.Top="99"
    Width="100" Panel.ZIndex="4" />
  <Ellipse x:Name="greenEllipse" Fill="Lime" Height="47"
    Canvas.Left="326" Stroke="Black" Canvas.Top="99"
    Width="97" Panel.ZIndex="2" />
  <Ellipse x:Name="blueEllipse" Fill="Blue" Height="130"
    Canvas.Left="257" Stroke="Black" Canvas.Top="84"
    Width="41" />
  <Ellipse x:Name="cyanEllipse" Fill="Cyan" Height="60"
    Canvas.Left="30" Stroke="Black" Canvas.Top="30"
    Width="60" Panel.ZIndex="2" />
  <Button x:Name="aButton" Content="cast test"
    Canvas.Left="376" Canvas.Top="172" Width="75"/>
</Canvas>
mouse drag values

Mouse drags rely on:
selecting an element to drag – usually with a mouseDown event when selecting element positions are also initialized.
dragging the mouse – responding to mouseMove events while mouseDown is true.
stopping the drag – responding to a mouseUp event

During the drag there are many mouseMove events. Each event has a mouse position property.

To know the current event's movement, the previous value must be known.

Let:
- x_shape and y_shape are the shape's current position
- x and y are the mouse's current position values
- x_canvas and y_canvas are the previous mouse positions

Then:
- x_shape += x - x_canvas;  x_canvas = x;
- y_shape += y - y_canvas;  y_canvas = y;
// Application variables
Ellipse selectedEllipse;
bool captured = false;
double x_shape, x_canvas, y_shape, y_canvas;

private void selectEllipse(object sender, MouseButtonEventArgs e) // canvas MouseRightButtonDown
{
    selectedEllipse = (Ellipse)e.Source;
    if (selectedEllipse != null) {
        Mouse.Capture(selectedEllipse);
        // alternate version
        // ((UIElement) e.OriginalSource).CaptureMouse();
        captured = true;
        x_shape = Canvas.GetLeft(selectedEllipse);
        x_canvas = e.GetPosition(canvas).X;
        y_shape = Canvas.GetTop(selectedEllipse);
        y_canvas = e.GetPosition(canvas).Y;
        statusLabel.Content = "Selected " +
            selectedEllipse.Name;
    }
}
private void releaseEllipse(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show("releaseEllipse: \r\t sender = " + sender.ToString() + " \r\t source = " + e.Source.ToString() + " \r\t original source = " + e.OriginalSource.ToString());
    if (captured) {
        Mouse.Capture(null);
        // alternate version
        // ((UIElement) e.OriginalSource).ReleaseMouseCapture();
        captured = false;
        statusLabel.Content = "Released " + selectedEllipse.Name;
    }
}
private void mouseMove(object sender, MouseEventArgs e) {
    double x = e.GetPosition(canvas).X;
    double y = e.GetPosition(canvas).Y;
    if (captured) {  // an Ellipse has mouse capture
        x_shape += x - x_canvas;
        Canvas.SetLeft(selectedEllipse, x_shape);
        x_canvas = x;
        y_shape += y - y_canvas;
        Canvas.SetTop(selectedEllipse, y_shape);
        y_canvas = y;
        statusLabel.Content = String.Format("{0} at ({1}, {2}) zIndex = {3}",
            selectedEllipse.Name, x, y,
            Panel.GetZIndex(selectedEllipse));
    }
    else statusLabel.Content = 
        String.Format("Mouse at ({0},{1})", x, y);
}
Canvas uses explicit positioning; graphics won’t resize to fit larger or smaller windows.

Viewbox class derives from Decorator (like the Border class) and can have a single child.

Viewbox stretches or shrinks its child to fit available space. Viewbox's default is proportional scaling that preserves the aspect ratio of its contents.

Place a Canvas inside a Viewbox, and place your shapes inside the Canvas.
WPF Window class derives from ContentControl contains a single child (a layout container) paint the background with a brush (Background property). You can use the BorderBrush and BorderThickness properties to add a border around your window, this border is added inside the window frame (client area)

Client area is the surface inside the window boundaries where you place your content.

Nonclient area includes the border and the title bar at the top of the window. Operating system manages this area.

Has many properties: Title, WindowStyle, WindowState, Window Icon (at top left corner), Top and Left (position), ResizeMode, ShowInTaskbar, SizeToContent, ....
To display a window, create an instance of the Window class and use:
Show() a modeless window
ShowDialog() a modal window

TaskWindow winTask = new TaskWindow();
winTask.ShowDialog();
// Execution reaches this point after winTask is closed.

.NET windows can “own” other windows
floating toolbox and command windows.
for example, the Find and Replace window in Microsoft Word.
Custom Dialogs

Showing a window modally offers the user a choice. The code that displays the window waits for the result of that choice and then acts on it.

Windows have a **DialogResult** property: a true, false, or null value.

Set the DialogResult, it’s returned to the calling code as the return value of `ShowDialog()`.

```csharp
dialogWindow dialog = new DialogWindow();
if (dialog.ShowDialog() == true) {
    // The user accepted the action. Full speed ahead.
} else { // The user canceled the action. }
```

Set a button as the accept button (by setting `IsDefault` to true). Clicking that button sets the DialogResult to true.

Set a button as the cancel button (by setting `IsCancel` to true), clicking will set the DialogResult to Cancel.