



Chapter 2

Drawing Shapes and Text

*Not quite fingerpaint,
more like Acrylics...*

Objectives:

◆ Drawing basics

- Coordinate system

◆ Drawing text

- Fonts

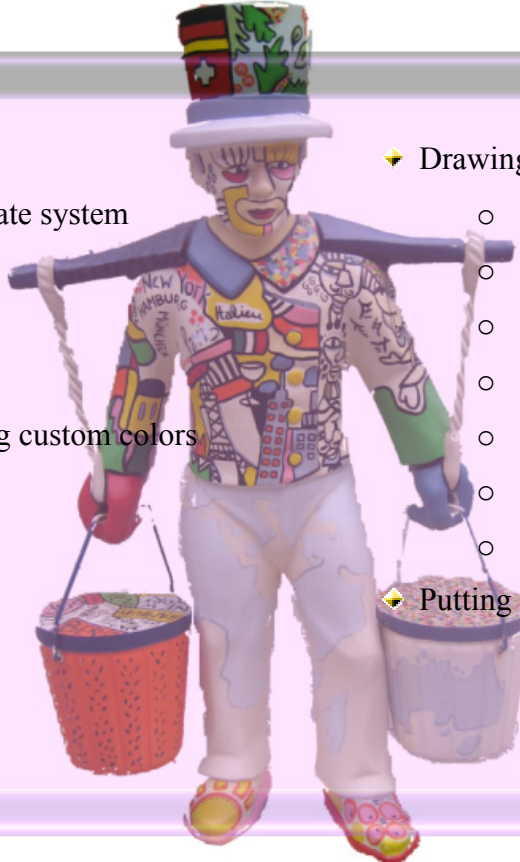
◆ Color basics

- Creating custom colors

◆ Drawing shapes

- Lines
- Ovals and circles
- Rectangles and squares
- Filling shapes
- Arcs
- Polygons
- Images

◆ Putting it together



Drawing Basic

We can draw things on an applet by writing a `paint` method. The `paint` method allows us access to a `Graphics` object, which is how we can specify what to draw and where to draw it. An outline of a program with a `paint` method is listed below:

```
import java.awt.*;           // access the Graphics object
import javax.swing.*;       // access to JApplet

public class DrawEx extends JApplet
{
    public void paint( Graphics g )
    {
        // put your code here!
    }
}
```

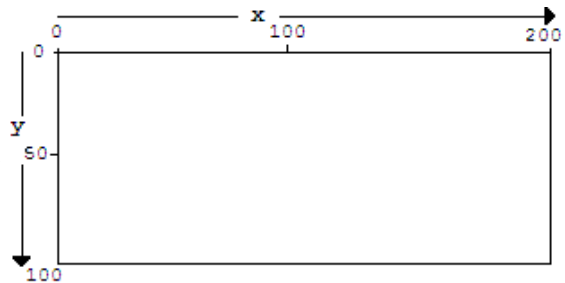
The `Graphics` object allows us to call *methods* for drawing. We can draw circles, ellipses, squares, rectangles and polygons, as well as write text.

Coordinate System

For all of these operations, we will need to specify where to draw it, using an `x` and `y` coordinate system.

The Java `x-y` coordinate system starts in the upper-left corner at `(0,0)`.

As you go right along the `x`-axis, the `x`-coordinate value goes up. As you go down along the `y`-axis, the `y`-coordinate value goes up.



Drawing Text

We can draw text on an applet by using the `drawString` method inside our `paint` method. The `drawString` method requires the text and `x` and `y` coordinates to specify where the text should begin.

```
graphicsObj.drawString( String, x-coordinate, y-coordinate );
```

Example inside the paint method:

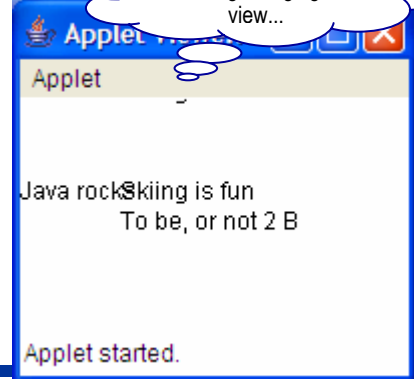
```
public void paint( Graphics grp )
{
    grp.drawString( "Some text to display", x, y );
}
```

The name of the Graphics object must match: example **grp**

Our first example draws text in various locations in our applet. Notice what happens when we specify a location of (0,0) for our x and y coordinates:

```
import java.awt.*;           // access the Graphics object
import javax.swing.*;       // access to JApplet

public class Text1 extends JApplet
{
    public void paint ( Graphics gr )
    {
        gr.drawString ( "Hello Worldling", 0, 0 );
        gr.drawString ( "Java rocks", 0, 50 );
        gr.drawString ( "Skiing is fun", 50, 50 );
        gr.drawString( "To be, or not 2 B", 50, 65 );
    }
}
```



The "Hello Worldling" is not visible in the applet because the x and y coordinates specify the bottom-left of where the text begins. So, if you are using 12-point font, your text begins at 0,0 and the top of your letters are at -12 in the y direction - not in the visible area for the applet! So, how would you fix this so that the text appeared at the top of the applet?

Also notice how the text can be written on top of other text. If you don't want this overlap, it is up to you to place them such that they don't overlap.



To draw text beneath text you've already drawn, we keep the x value the same and increase the y value. The default font size is 12 pt, which means 12 pixels. Add 12 (or a little more for nice spacing between the two lines) to the y value and you'll draw text beneath our other text!.

Fonts

We can change the *font* of our text by specifying a particular font name, style, and size. The structure for creating a font appears below:

```
Font fnt = new Font( type, style, size );  
g.setFont( fnt );
```

OR

```
g.setFont( new Font( type, style, size ) );
```



Although you can try to reference some fancy fonts, they may not be available on other systems that run your applet! The main fonts you can always depend on are: "Serif", "SansSerif", "Monospaced", and "Dialog".

There are four styles that are available:

- ✦ Font.PLAIN
- ✦ Font.BOLD
- ✦ Font.ITALIC
- ✦ Font.BOLD + Font.ITALIC

This last one is a combination of both **bold** and *italic* to get ***boldItalic***.

A range of sizes are available, but regular text is usually either 10-pt or 12-pt font. Anything below size 8 is nearly impossible to read.

So now we can create fonts, such as the following:

```
Font small = new Font( "Serif", Font.PLAIN, 8 );  
Font big = new Font( "SanSerif", Font.BOLD + Font.ITALIC, 36 );
```

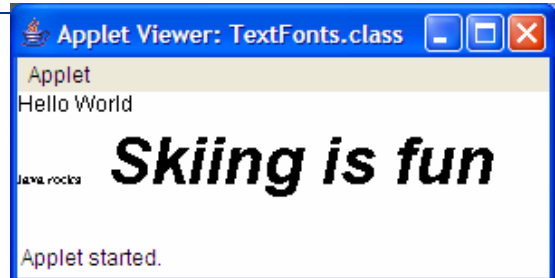
To use these fonts, we can apply them to our components. So far, we have only been working with the Graphics component. To apply a particular font, you call the method `setFont` on the object.

```

import java.awt.*;
import javax.swing.*;

public class TextFonts extends JApplet
{
    public void paint ( Graphics g )
    {
        g.drawString ("Hello World",0,10 );
        Font small = new Font( "Serif", Font.PLAIN, 8 );
        g.setFont( small );
        g.drawString ("Java rocks", 0,50 );
        Font big = new Font( "SanSerif", Font.BOLD + Font.ITALIC, 36 );
        g.setFont( big );
        g.drawString ( "Skiing is fun", 50, 50 );
    }
}

```



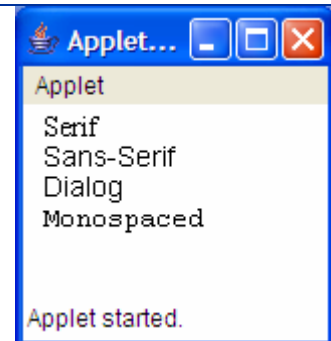
This next example shows the differences between the font types. Monospaced is similar to Courier – they are both fixed-width fonts. This means that each letter has the same amount of spacing, therefore lower-case i's have a lot of space on either side. The others are all *variable-width* fonts, such that i's only take up as much space as necessary. Variable-width fonts tend to be easier to read, but if we want to ensure the spacing then a fixed-width font may be desired.

```

import java.awt.*;
import javax.swing.*;

public class FontTypes extends JApplet
{
    public void paint ( Graphics g )
    {
        Font serif = new Font( "Serif", Font.PLAIN, 14 );
        g.setFont( serif );
        g.drawString ( "Serif", 10, 15 );
        g.setFont( new Font ( "Sans-Serif", Font.PLAIN, 14 ) );
        g.drawString ( "Sans-Serif", 10, 30 );
        g.setFont( new Font ( "Dialog", Font.PLAIN, 14 ) );
        g.drawString ( "Dialog", 10, 45 );
        g.setFont( new Font ( "Monospaced", Font.PLAIN, 14 ) );
        g.drawString ( "Monospaced", 10, 60 );
    }
}

```



Color Basics

We can change the color when we draw shapes and text by creating `Color` objects. The simplest way to use colors is to reference one of the colors already created for Java programs.

The following colors are available:

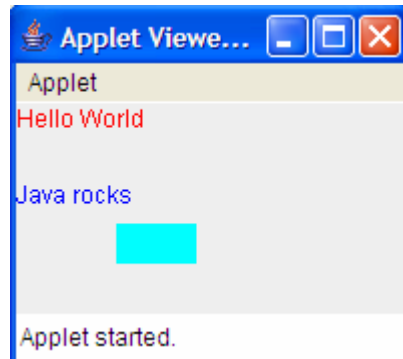
`Color.BLACK`
`Color.BLUE`
`Color.CYAN`
`Color.DARK_GRAY`
`Color.GRAY`

`Color.GREEN`
`Color.LIGHT_GRAY`
`Color.MAGENTA`
`Color.ORANGE`
`Color.PINK`

`Color.RED`
`Color.WHITE`
`Color.YELLOW`

If you want to use a color, you use the method `setColor(color)` on the component you wish to change colors.

```
import java.awt.*;
import javax.swing.*;
public class ColorEx extends JApplet
{
    public void paint ( Graphics g )
    {
        g.setColor( Color.RED );
        g.drawString ( "Hello World", 0,12 );
        g.setColor( Color.BLUE );
        g.drawString ( "Java rocks", 0,50 );
        g.setColor( Color.CYAN );
        g.fillRect ( 50,60,40,20 );
    }
}
```



A good way to think about how this `setColor` method works is to imagine we're using a marker. By default, we're holding on to a black marker, and anything we're asked to draw we do with the black marker. As soon as we call `setColor` to change the color, for example to red, then we put down the black marker and now pick up the red marker. Now, anything we're asked to draw will be in red, until we call `setColor` to change to another color.

Creating Custom Colors

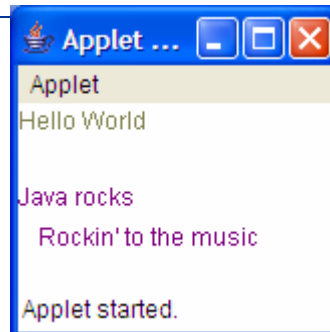
You can also create your own custom colors. You do so by specifying how much red, green and blue you want in your color.

```
Color mycolor = new Color( red, green, blue );
```

The max number for these values is 255, which means you want a lot of that color. Therefore, we can analyze some colors:

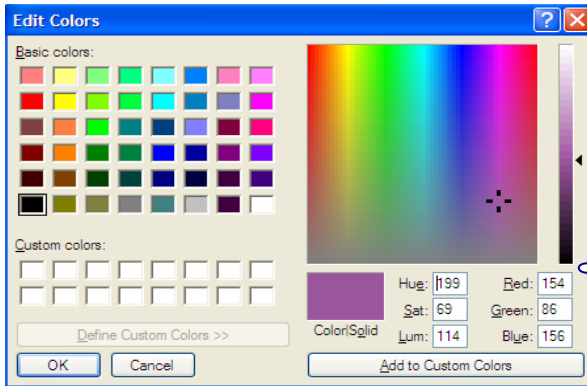
```
Color red = new Color( 255, 0, 0 );  
Color gray = new Color( 128, 128, 128 );  
Color yellow = new Color( 255, 255, 0 );  
Color white = new Color( 255, 255, 255 );  
Color black = new Color( 0, 0, 0 );
```

```
import java.awt.*;  
import javax.swing.*;  
public class ColorEx extends JApplet  
{  
    public void paint ( Graphics g )  
    {  
        g.setColor( new Color( 130, 130, 80 ) );  
        g.drawString ( "Hello World", 0,12 );  
        g.setColor( new Color( 128, 0, 128 ) );  
        g.drawString ( "Java rocks", 0,50 );  
        g.drawString ( "Rockin' to the music", 10,70 );  
    }  
}
```



There are several ways to figure out the RGB colors for the color we want. We could do the trial-and-error method of trying different values until we figure it out. An easier way would be to look for a color wheel on the Internet with the RGB values given. We can also use any graphics program such as Microsoft Paint to tell us as well.

In MS Paint, select "Colors" then "Edit Colors..." then click on the button "Define Custom Colors..." It should display a whole spectrum of colors. When we click on a color, it shows us the RGB values in the lower right-hand corner of the dialog box.



RGB values as displayed inside MS Paint

Drawing Shapes - Lines

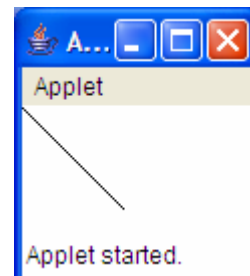
We use the coordinate system to specify the x and y coordinates of where our line should start and x and y coordinates for where the line should end.

The specification for the `drawLine` method follows:

```
g.drawLine( x, y, x2, y2 );
```

Our first example draws a diagonal line from the upper-left corner (0,0) to (50,50).

```
import java.awt.*; // access the Graphics object
import javax.swing.*; // access to JApplet
public class LineDiagnol extends JApplet
{
    public void paint( Graphics g )
    {
        g.drawLine( 0,0, 50, 50 );
    }
}
```



To draw a horizontal line, the y values need to stay the same. For example:

```
g.drawLine( 10, 40, 70, 40 ); // draws horizontal line 40 pixels down
g.drawLine( 30, 10, 50, 10 ); // draws horizontal line 10 pixels down
```

To draw a vertical line, the x values need to stay the same.

```
g.drawLine( 40, 10, 40, 30 ); // draws vertical line 40 pixels right
g.drawLine( 30, 10, 30, 50 ); // draws vertical line 30 pixels right
```

Experiment with different values to see what type of lines you can draw.

Ovals and Circles

Let's start off by drawing an oval/ellipse.

Ellipses are created using the method: `drawOval`

We need to send some *parameters* to the method, to designate where we want our ellipse drawn and the width and height.

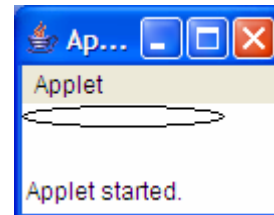
The specification for the `drawOval` method follows:

```
g.drawOval( x, y, width, height );
```

The x and y coordinates specify the location of the upper-left corner of the ellipse. Note that it does not specify the middle of the ellipse! If we place our ellipse in the top-left corner of the applet, we could use an x and y coordinate of zero. Let's make a really wide oval with a width of 100 and a height of 10.

The following code and example run is depicted below:

```
import java.awt.*;           // access the Graphics object
import javax.swing.*;       // access to JApplet
public class Ellipse extends JApplet
{
    public void paint( Graphics g )
    {
        g.drawOval( 0,0, 100, 10 );
    }
}
```



Now, if we wanted to draw a circle, how would we change the above code? There is no method called `drawCircle`, so we have to use the `drawOval` method. But that's ok, because we know that circles are ovals with the same width and height. Modify the above program to make it draw a circle.

Rectangles and Squares

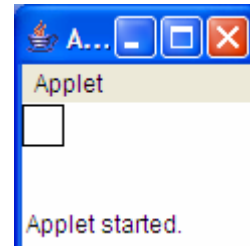
In a similar way, we can draw rectangles and squares. The method we use is called `drawRect` and we specify the `x` and `y` coordinates for the upper-left corner of the rectangle, and a width and height to specify the dimensions.

```
g.drawRect ( x, y, width, height );
```

Like the circle, we can draw a square by specify a rectangle with the same width and height. The following code and example run shows how to draw a square:

```
import java.awt.*;           // access the Graphics object
import javax.swing.*;       // access to JApplet

public class Square extends JApplet
{
    public void paint ( Graphics g )
    {
        g.drawRect ( 0,0, 20, 20 );
    }
}
```



We can also draw a rounded rectangle by calling the `drawRoundRect` method.

Filling shapes

Sometimes we want our shapes to be filled in with color. It is easy to fill in our shapes, by simply changing the method call from *draw* to *fill*. . *This does not work on lines.*

For example, instead of `drawRect` we can call `fillRect`. The rest of the code is the same.

Examples:

```
g.drawRect( 0, 0, 20, 20 );    →    g.fillRect( 0, 0, 20, 20 );
g.drawOval( 0, 0, 100, 10 );  →    g.fillOval( 0, 0, 100, 10 );
```

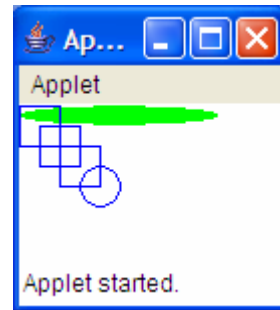
We can draw multiple objects on the same applet by making multiple calls to these methods. In the following example, we change the location of each shape that we draw.

```

import java.awt.*;           // access the Graphics object
import javax.swing.*;       // access to JApplet

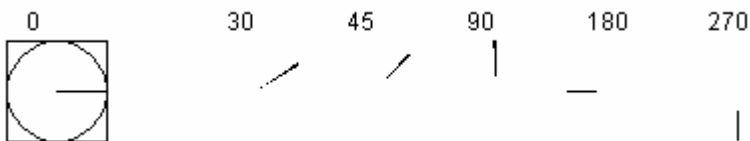
public class Shapes extends JApplet
{
    public void paint ( Graphics g )
    {
        g.setColor( Color.GREEN );
        g.fillOval( 0,0, 100, 10 );
        g.setColor( Color.BLUE );
        g.drawRect( 0,0, 20, 20 );
        g.drawRect ( 10,10, 20, 20 );
        g.drawRect ( 20,20, 20, 20 );
        g.drawOval ( 30, 30, 20, 20 );
    }
}

```



Arcs

Arcs are a bit more difficult to get your head around. When drawing an arc, we still need to specify the x and y coordinates and a width and height. But we also need to specify the starting angle and angle of the arc. The starting angle is based from the center of our x-y coordinates and width/height dimensions, with the angle of zero extending horizontally to the right. See the diagrams below for examples of starting angles:



Then we also define the angle for the arc. The following images depict different arc angles:



The code to create an arc is as follows:

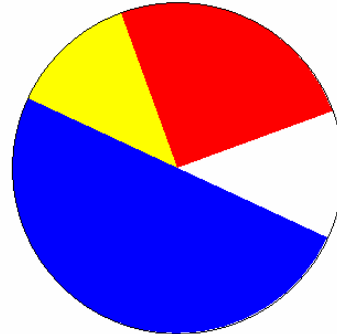
```
g.drawArc( x, y, width, height, startingAngle, archingAngle );
```

OR, filled in

```
g.fillArc( x, y, width, height, startingAngle, archingAngle );
```

The width and height are based on the concept that an arc of 360 degree angle is a full oval, like the ones we draw with drawOval.

```
import javax.swing.*;
import java.awt.*;
public class PieChart extends JApplet
{
    public void paint( Graphics g )
    {
        // pie
        g.setColor( Color.RED );
        g.fillArc( 20,20, 300, 300, 20, 90 );
        g.setColor( Color.YELLOW );
        g.fillArc( 20,20, 300,300, 110,45 );
        g.setColor( Color.BLUE );
        g.fillArc( 20,20, 300,300, 155, 180 );
        // outline
        g.setColor( Color.BLACK );
        g.drawArc( 20,20, 300, 300, 0, 360 );
    }
}
```



drawing the outline has to be
done last – Why?
To overlay it on **top** of color pies

Polygons

Polygons can include any numbers of points. To create a polygon, first we declare and instantiate a Polygon object:

```
Polygon poly;
poly = new Polygon( );
```

Now we can add as many points we want to our polygon. Keep in mind that the order in which we add points to the polygon is important – think of it like connect-the-dots. The polygon will be created by drawing lines from one point to the next. We add points to the polygon by calling the **addPoint** method:

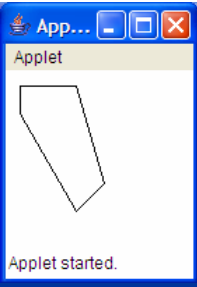
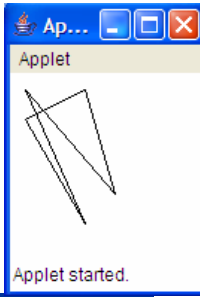
```
poly.addPoint( x, y );
```

where x and y are our x and y coordinates for the point.

When we're ready to draw a polygon, we can use either the draw or fill methods from the Graphics class:

```
g.drawPolygon( poly );  
g.fillPolygon( poly );
```

The following two examples show the same points in the polygon, but in different order:

<pre>import javax.swing.*; import java.awt.*; public class PolyEx1 extends JApplet { public void paint(Graphics g) { Polygon pg = new Polygon (); pg.addPoint(10, 10); pg.addPoint(50, 10); pg.addPoint(70, 80); pg.addPoint(50, 100); pg.addPoint(10, 30); g.drawPolygon(pg); } }</pre> 	<pre>import javax.swing.*; import java.awt.*; public class PolyEx2 extends JApplet { public void paint(Graphics g) { Polygon pg = new Polygon (); pg.addPoint(10, 10); pg.addPoint(50, 100); pg.addPoint(10, 30); pg.addPoint(50, 10); pg.addPoint(70, 80); g.drawPolygon(pg); } }</pre> 
--	--

Images

Images can be painted on to our applet with a call to the **drawImage** method. But first we need to load the image in to the applet. To load the image, we call the method **getImage** and specify the location of where we're running the applet, which is obtained through a call to **getCodeBase()** and then the name of the image file.

```
Image imageVariable = getImage( getCodeBase( ), filename );
```

To draw the image on to our applet, we call **drawImage** specifying the image variable from above, the x, y coordinates of 0,0 (top-left corner) of where to draw it, and the keyword **this**.

```
g.drawImage( imageVariable, 0, 0, this );
```

```
import javax.swing.*;
import java.awt.*;
public class ImageEx extends JApplet
{
    public void paint( Graphics g )
    {
        Image img = getImage( getCodeBase( ), "Lion.jpg" );
        g.drawImage( img, 0,0, this );
    }
}
```



The following lists the types of drawing you can do:

Lines	drawLine(x, y, x2, y2)
Rectangles	drawRect(x, y, width, height) fillRect(x, y, width, height) clearRect(x, y, width, height)
Rounded rectangles	drawRoundRect(x, y, width, height, arcWidth, arcHeight) fillRoundRect(x, y, width, height, arcWidth, arcHeight)
3-D Raised or lowered rectangles	Raised: draw3DRect(x, y, width, height, true) Lowered: draw3DRect(x, y, width, height, false) Raised: fill3DRect(x, y, width, height, true) Lowered: fill3DRect(x, y, width, height, false)
Ovals	drawOval(x, y, width, height) fillOval(x, y, width, height)
Arcs	drawArc(x, y, width, height, startAngle, arcAngle) fillArc(x, y, width, height, startAngle, arcAngle)
Polygons	drawPolygon(Polygon) fillPolygon(Polygon)
Images	drawImage(Image, x, y, this)

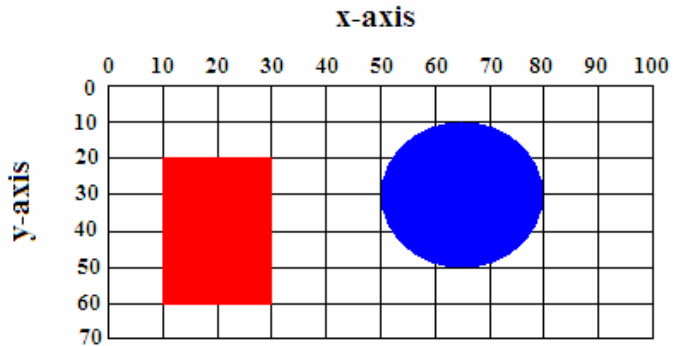
Putting it Together

The figure to the right depicts a grid view of the Java coordinate system. Each point along the x-axis (horizontal) and y-axis (vertical) is labeled at 10-pixel segments. To draw the rectangle, we specify the x and y coordinates of where it begins (10, 20) then specify the width of 20 pixels and height of 40 pixels. The red rectangle can be drawn using the fill method fillRect as:

```
g.setColor( Color.RED );  
g.fillRect ( 10, 20, 20, 40 );           // x, y, width, height
```

The blue circle is drawn similarly, with the x-coordinate as the top-left corner of where we want to begin the circle (50), and the y-coordinate at the top-left corner as well (10). The width of the circle is 30 pixels, and height is 40 pixels.

```
g.setColor( Color.BLUE );  
g.fillOval( 50, 10, 30, 40 );           // x, y, width, height
```



Summary

- Coordinate system starts in the upper left corner with (0,0).
- Drawing shapes is accomplished by calling methods on the Graphics object inside the paint method.
- Lines are drawn using **drawLine**. `drawLine(x, y, x2, y2)`
- Horizontal lines are drawn with two different x values and the y values unchanged.
- Vertical lines are drawn with the x values unchanged and different y values.
- Circles and ellipses are drawn using **drawOval**. `drawOval(x, y, width, height)`
- Circles have the same value for the width and height of the oval.
- Rectangles and squares are drawn using **drawRect**. `drawRect(x, y, width, height)`
- Squares have the same value for both the width and height.
- Arcs are drawn using **drawArc**. `drawArc(x, y, width, height, startAngle, arcAngle)`
- Polygons are drawn by creating a **Polygon** object, adding points to the Polygon object with **addPoint**, then drawn by calling **drawPolygon** on the Graphics object.
- Draw/fill methods on the Graphics object are drawn in order of the program, where subsequent drawings are drawn on top of the previously drawn ones.
- Colors can be set by calling the method **setColor** on the Graphics object. `setColor(Color)`
- There is a set of colors that can be referenced as `Color.name` (e.g. `Color.RED`).
- Custom colors can be created by specifying the amount of red, green and blue. **Color newColor = new Color(red, green, blue);**
- Text can be drawn by calling **drawString** on the Graphics object. `drawString(string, x, y)`
- Fonts are created by specifying the type (e.g., "Serif"), style (e.g., `Font.BOLD` or `Font.PLAIN`) and the size in pixels. `Font fnt = new Font("Serif", Font.PLAIN, 12);`
- Font of bold and italic can be specified with a style of **Font.BOLD + Font.ITALIC**.
- Set the font for drawing by calling the **setFont** method on the Graphics object.

Drawing Exercises

1. In a window that is 100x100 pixels, position 95,95 is nearest to the _____ corner.
2. If you draw a rectangle with the same value for width and height, what did you draw?
 - a. parallelogram
 - b. square
 - c. rounded square
 - d. rounded rectangle
 - e. polygon
 - f. hexagon
3. What two steps are necessary to paint an image?
4. True or False: The paint method is called automatically by the browser whenever it needs to redraw the screen
5. How do you create a new font that is both bold and italic?
6. In what package is the Graphics object?
7. True/False: To draw a circle, call the method drawCirc and send the x,y coordinates and the radius of the circle?
8. Take the Hello World example and center the text.
9. Draw a car. Draw a house. Draw a self-portrait. Draw Pac-Man. Draw a spaceship.
10. Draw a smiley face. Draw a pirate face. Draw a farm house with animals.
11. Draw a bar graph (histogram). Draw differnt chart types on the same data values.
12. Write out a multi-line blog. If you're unsure what a blog is, google it.
13. True/False: You can create colors by specifying the amount of red, blue and yellow.
14. In the arc example, what would happen if you draw the outline before filling in each of the pie sections?
15. What colors do the following statements produce?
Color mycolor = new Color(255, 0, 255);
Color mycolor = new Color(0, 0, 255);
Color mycolor = new Color(128, 128, 128);

16. Correct the errors in the following program: (best to rewrite it to the right) about 15 errors. Find as many as you can =>

```
// *****  
My First Applet – I'm sooo proud!  ***/  
  
Import java.swing;  
  
Public class MyfirstApplet  
{  
    private void paint ( graphics theGraphics);  
    {  
        graphics.drawstring( 'chocolate', 20 )  
    }  
}
```