The methods used to display graphics and graphical user interfaces (GUI) in Java have gone through several evolutions since Java's introduction in 1996. The first toolkit to display GUIs in Java was the Abstract Window Toolkit, or AWT. AWT was implemented using platform-specific code. The successor to AWT is Swing. Swing is written in Java, which provides platform independence. Swing is complementary to AWT rather than a replacement. A typical Java program written using Swing would incorporate libraries from both AWT and Swing. While there are still many Java programs written today using Swing, the most recent graphics toolkit for Java is JavaFX.

JavaFX is a set of packages that allow Java programmers to create rich graphics and media applications. Potential applications include GUI interfaces, 2D and 3D games, animations, visual effects, touch-enabled applications, and multimedia applications.

A Sample JavaFX Application

JavaFX application that draws a happy face.

```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;
public class HappyFace extends Application
{
   public static void main(String[] args)
   {
      launch(args);
   }
   @Override
   public void start(Stage primaryStage) throws Exception
   {
       Group root = new Group();
       Scene scene = new Scene(root);
       Canvas canvas = new Canvas(400, 300);
       GraphicsContext gc = canvas.getGraphicsContext2D();
       gc.strokeOval(100, 50, 200, 200);
       gc.fillOval(155, 100, 10, 20);
       gc.fillOval(230, 100, 10, 20);
       gc.strokeArc(150, 160, 100, 50, 180, 180, ArcType.OPEN);
       root.getChildren().add(canvas);
       primaryStage.setTitle("HappyFace in JavaFX");
       primaryStage.setScene(scene);
       primaryStage.show();
   }
}
```

Program Output

The section

```
import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.shape.ArcType;
```

says that the application uses a number of library packages in the **JavaFX** library. They include classes for **Application, Canvas, Scene, Group, Stage, GraphicsContext, and ArcType.**

The line :

public class HappyFace extends Application

begins the class definition for the program. It is named HappyFace. The words extends Application indicate that we are defining a JavaFX application, as opposed to some other kind of class. We are using inheritance to create the class HappyFace based upon an existing class Application.

The application contains two methods—main and start. The main method is where a Java program normally begins.

```
public static void main(String[] args)
{
    launch(args);
}
```

A JavaFX program begins execution in the start method. The main method is ignored in a correctly deployed JavaFX application. However, it is common to include main and a call to launch as a fallback, which will end up launching the JavaFX program and the start method.

For a JavaFX application, programs begin in the start method.

```
@Override
public void start(Stage primaryStage) throws Exception
```

For now, you can ignore the @Override and the throws Exception code. What you do need to know is that this method is invoked automatically when the JavaFX application is run. JavaFX uses the metaphor of a stage and scenes, just like the stage and scene of a theater.

The next four lines set up a canvas on a scene for you to draw simple graphics.

```
Group root = new Group();
Scene scene = new Scene(root);
```

```
Canvas canvas = new Canvas(400, 300);
GraphicsContext gc = canvas.getGraphicsContext2D();
```

At this point, we can now use drawing operations on the canvas. The method invocation

```
gc.strokeOval(100, 50, 200, 200);
```

draws the big circle that forms the outline of the face. The first two numbers tell where on the screen the circle is drawn. The method strokeOval, as you may have guessed, draws ovals. The last two numbers give the width and height of the oval. To obtain a circle, you make the width and height the same size, as we have done here. The units for these numbers are called ***pixels***.

The two method invocations

```
gc.fillOval(155, 100, 10, 20);
gc.fillOval(230, 100, 10, 20);
```

draw the two eyes. The eyes are “real” ovals that are taller than they are wide. Also notice that the method is called fillOval, not strokeOval, which means it draws an oval that is filled in.

The next invocation

```
gc.strokeArc(150, 160, 100, 50, 180, 180, ArcType.OPEN);
```

draws the mouth. Finally, the block

```
root.getChildren().add(canvas);
primaryStage.setTitle("HappyFace in JavaFX");
primaryStage.setScene(scene);
primaryStage.show();
```

sets a title for the window and does some bookkeeping to set the stage and display the window.

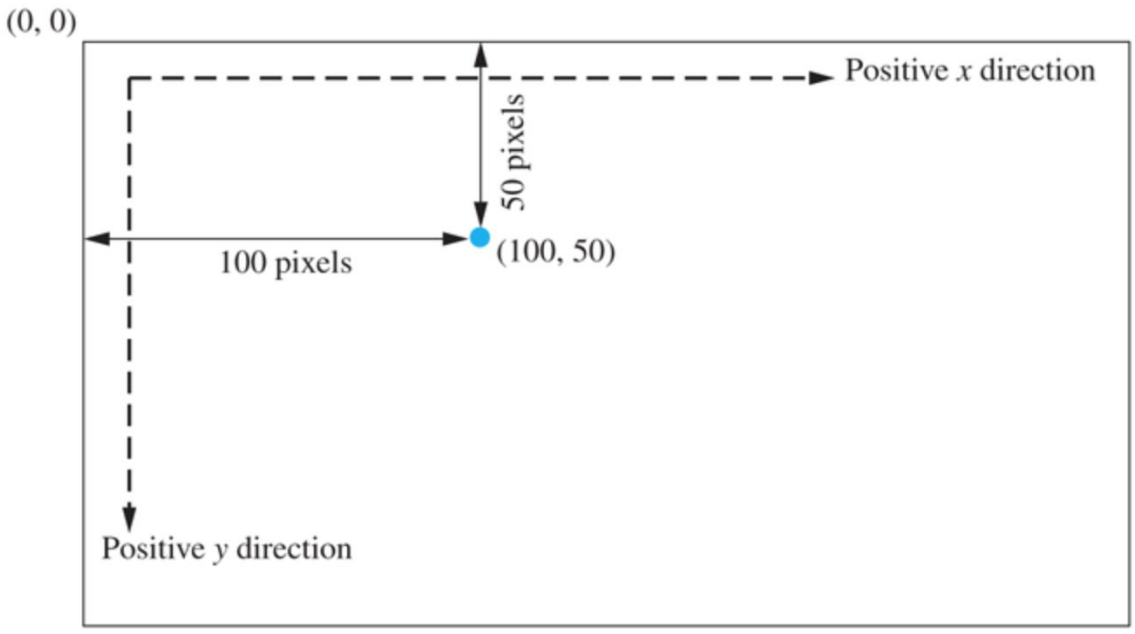
Size and Position of Figures

All measurements within a screen display are given not in inches or centimeters but in pixels. A **pixel**—short for picture element—is the smallest length your screen is capable of showing. A pixel is not an absolute unit of length like an inch or a centimeter. The size of a pixel can be different on different screens, but it will always be a small unit.

You can think of your computer screen as being covered by small squares, each of which can be any color. You cannot show anything smaller than one of these squares. A pixel is one of these squares, but when used as measure of length, a pixel is the length of the side of one of these squares.A **megapixel** is just a million pixels.

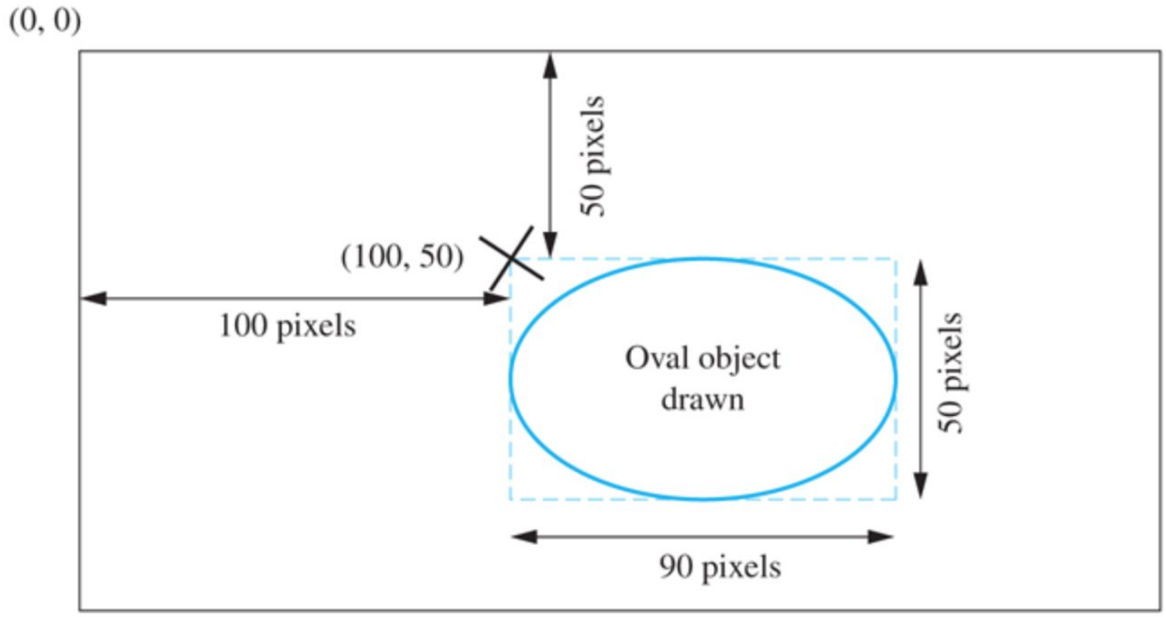
Note: A pixel is the smallest length shown on a screen.

Figure below shows the **coordinate system** used to position figures inside of an applet or other kind of Java window-like display. Think of the large rectangle as outlining the drawing area that is displayed on the screen. The coordinate system assigns two numbers to each point inside the rectangle. The numbers are known as the **x-coordinate** and the **y-coordinate** of the point. The **x-coordinate** is the number of pixels from the left edge of the rectangle to the point. The **y-coordinate** is the number of pixels from the top edge of the rectangle to the point. The coordinates are usually written within parentheses and separated by a comma, with the *x*-coordinate first. So the point marked with a blue dot in has the coordinates (100, 50); 100 is the *x*-coordinate and 50 is the *y*-coordinate.



Each coordinate in this system is greater than or equal to zero. The *x*-coordinate gets larger as you go to the right from point (0, 0). The *y*-coordinate gets larger as you go down from point (0, 0). You position a rectangle in this graphical coordinate system at coordinates (*x*, *y*) by placing its upper left corner at the point (*x*, *y*). For example, the rectangle given by the dashed blue lines in **Figure below** is positioned at point (100, 50), which is marked with a black X. You position a figure that is not a rectangle at point

(*x*, *y*) by first enclosing it in an imaginary rectangle that is as small as possible but still contains the figure and then by placing the upper left corner of this enclosing rectangle at (*x*, *y*). For example, in **Figure below** the oval is also positioned at point (100, 50). If the applet contains only an oval and no rectangle, only the oval shows on the screen. But an imaginary rectangle is still used to position the oval.



The Oval Drawn by gc.strokeOval (100, 50, 90, 50)

Drawing Ovals and Circles

The oval in **Figure above** is drawn by the Java statement

gc.strokeOval(100, 50, 90, 50);

The first two numbers are the *x*- and *y*-coordinates of the upper left corner of the imaginary rectangle that encloses the oval. That is, these two numbers are the coordinates of the position of the figure drawn by this statement. The next two numbers give the width and height of the rectangle containing the oval (and thus the width and height of the oval itself). **If the width and height are equal, you get a circle.**

Now let's return to the statements in the body of the method paint:

gc.strokeOval(100, 50, 200, 200);
gc.fillOval(155, 100, 10, 20);
gc.fillOval(230, 100, 10, 20);

In each case, the first two numbers are the *x*- and *y*-coordinates of the upper left corner of an imaginary rectangle that encloses the figure being drawn. The first statement draws the outline of the face at position (100, 50). Since the width and height—as given by the last two arguments—have the same value, 200, we get a circle whose diameter is 200. The next two statements draw filled ovals for the eyes positioned at the points (155, 100) and (230, 100). The eyes are each 10 pixels wide and 20 pixels high.

Note: StrokeOval and fillOval draw ovals or circles

Recap The Methods StrokeOval and fillOval

Syntax

gc.strokeOval(*x*, *y*, *Width*, *Height*);
gc.fillOval(*x*, *y*, *Width*, *Height*);

The method strokeOval draws the outline of an oval that is *Width* pixels wide and *Height*pixels high. The oval is placed so that the upper left corner of a tightly enclosing rectangle is at the point (*x*, *y*).

The method fillOval draws the same oval as strokeOval but fills it in.

Drawing Arcs

Arcs, such as the smile on the happy face , are specified as a portion of an oval. For example, the following statement from **example above** draws the smile on the happy face:

gc.strokeArc(150, 160, 100, 50, 180, 180, ArcType.Open);

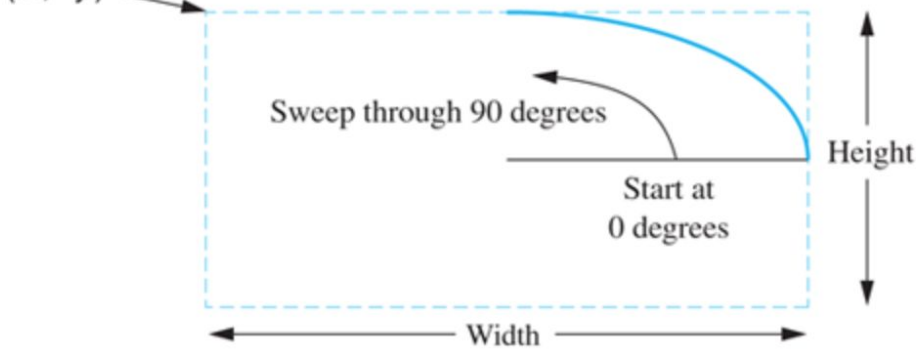
The first two arguments give the position of an invisible rectangle. The upper left corner of this rectangle is at the point (150, 160). The next two arguments specify the size of the rectangle; it has width 100 and height 50. Inside this invisible rectangle, imagine an invisible oval with the same width and height as the invisible rectangle. The next two arguments specify the portion of this invisible oval that is made visible. In this example,

the bottom half of the oval is visible and forms the smile. Let's examine these next two arguments more closely.

The fifth argument of strokeArc specifies a start angle in degrees. The last argument specifies how many degrees of the oval's arc will be made visible. The rightmost end of the oval's horizontal equator is at zero degrees. As you move along the oval's edge in a counterclockwise direction, the degrees increase in value. For example, **in Figure(**a) shows a start angle of 0 degrees; we measure 90 degrees along the oval in a counterclockwise direction, making one quarter of the oval visible. Conversely, as you move along the oval in a clockwise direction, the degrees decrease in value. For example, in **Figure(**b), we start at 0 and move −90 degrees in a clockwise direction, making a different quarter of the oval visible. If the last argument is 360, you move counterclockwise through 360 degrees, making the entire oval visible, as **Figure(**c) shows.

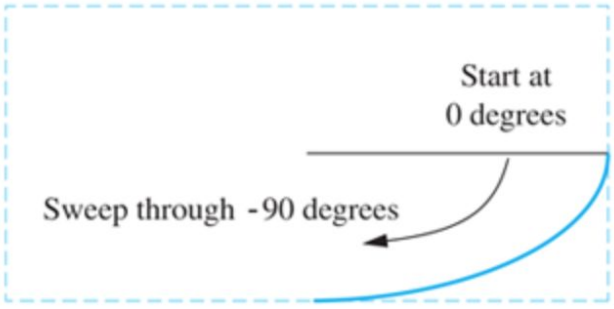
(a)

```
gc.strokeArc(x, y, width, height, 0, 90, ArcType.OPEN);
```



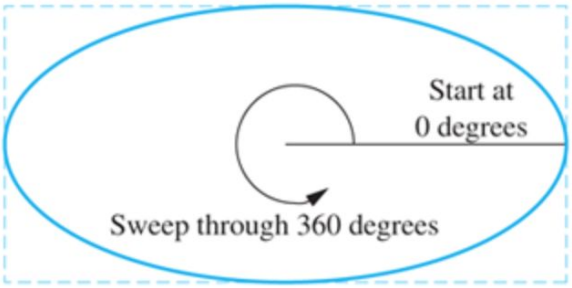
(b)

```
gc.strokeArc(x, y, width, height, 0, -90, ArcType.OPEN);
```



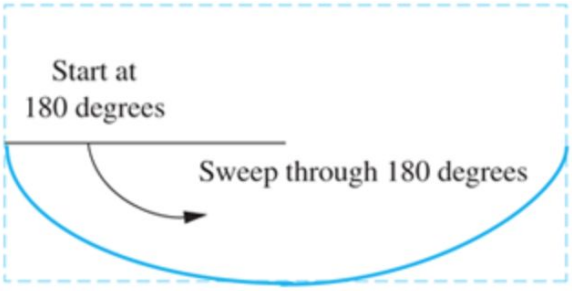
(c)

```
gc.strokeArc(x, y, width, height, 0, 360, ArcType.OPEN);
```



(d)

```
gc.strokeArc(x, y, width, height, 180, 180, ArcType.OPEN);
```



Note: drawArc draws part of an oval

Recap strokeArc

Syntax

gc.strokeArc(*x, y, Width, Height, StartAngle, ArcAngle, ArcType*);

Draws an arc that is part of an oval placed so the upper left corner of a tightly enclosing rectangle is at the point (*x*, *y*). The oval's width and height are *Width* and *Height,* both in pixels. The portion of the arc drawn is given by *StartAngle* and *ArcAngle,* both given in degrees. The rightmost end of the oval's horizontal equator is at 0 degrees. You measure positive angles in a counterclockwise direction and negative angles in a clockwise direction. Beginning at *StartAngle,* you measure *ArcAngle* degrees along the oval to form the arc. **ArcType can be ArcType.OPEN, ArcType.CHORD, or ArcType ROUND.**

What is Graphics Context gc?

The identifier gc names an object that does the drawing. Note that gc is a “dummy variable” that stands for an object that Java supplies to do the drawing. You need not use the identifier gc, but you do need to be consistent. If you change one occurrence of gc to, say, pen, you must change all occurrences of gc to pen. Thus, the method start shown in **example** could be written as follows:

```
GraphicsContext pen = canvas.getGraphicsContext2D();
pen.strokeOval(100, 50, 200, 200);
pen.fillOval(155, 100, 10, 20);
pen.fillOval(230, 100, 10, 20);
pen.strokeArc(150, 160, 100, 50, 180, 180, ArcType.OPEN);
```

Finally, **Figure** (d) illustrates an arc that begins at 180 degrees, so it starts on the left end of the invisible oval. The sixth argument is also 180, so the arc is made visible through 180 degrees in the counterclockwise direction, or halfway around the oval. Finally, the last argument specifies the type of arc. Use ArcType.OPEN to leave an open arc that is not connected. Use ArcType.CHORD to close the arc by drawing a line segment from the start to the end. Use ArcType.ROUND to close the arc by drawing line segments from the start to the center to the end of the segment. The smile on the happy face in **example** uses ArcType.OPEN.