
Chapter 14

Graphics

What is in This Chapter ?

As programmers, we will likely all eventually come across a situation in which we need to display graphics. Graphics may be pictures or perhaps drawings consisting of lines, circles, rectangles etc... For example, if we want to have an application that displays a bar graph, there is no "magical" component in JAVA that does this for us. In this chapter, we will learn the basics of **drawing graphics**, **displaying images**, and **manipulating graphics** in our JAVA applications.



14.1 Doing Simple Graphics

Graphics are used in many applications to display graphs, statistics, diagrams, pictures etc... Some applications are even completely based on graphics such as games, paint programs, PowerPoint etc... We have already seen how to display images on our application window inside labels, buttons etc... Now we will see how to actually draw our own graphics, as when drawing graphs or diagrams. To draw something, we create a new **Canvas** object, specifying its width and height:

```
Canvas canvas = new Canvas(300, 300);
```

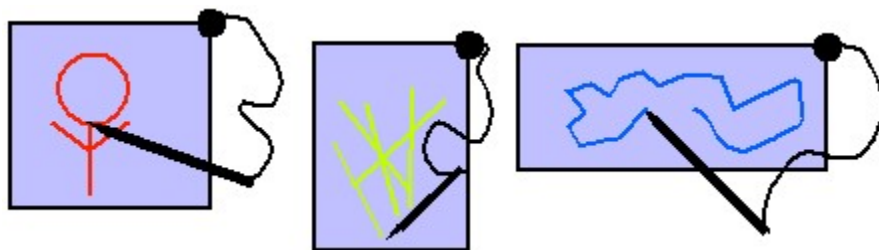
Normally, we add this to our root **Pane** when we create our application:

```
Pane p = new Pane();
Canvas canvas = new Canvas(300, 300);
p.getChildren().add(canvas);
```

Once the **Canvas** has been created, we then need to get a kind of "pen" for drawing onto it. We do this by asking the **Canvas** for its **GraphicsContext**:

```
GraphicsContext aPen = canvas.getGraphicsContext2D();
```

We can then use this "pen" to start drawing. Think of each canvas as having its own "pen" that can only be used to draw on it, just like the pens attached to kiosks at the bank.



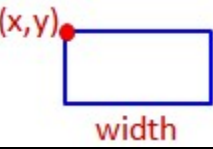
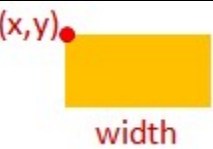
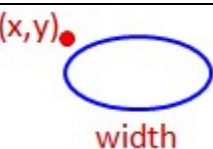
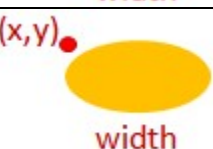
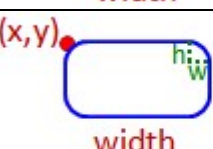
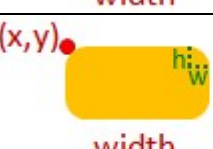
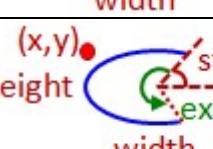
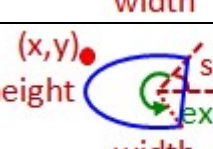
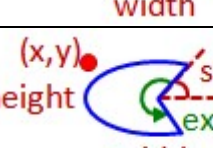
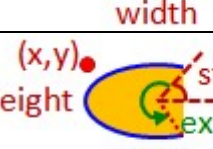
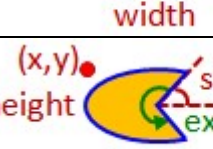
One of the things that we get to do is specify the outline color (i.e., stroke color) as well as the fill color (i.e., the shape's fill-in color) by using **setStroke()** and **setFill()**:

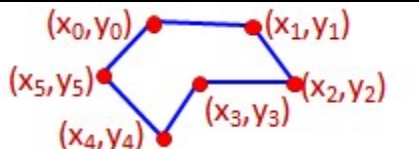
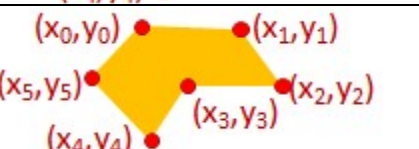
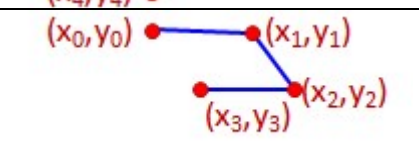


```
aPen.setStroke(Color.BLACK);
aPen.setFill(Color.YELLOW);
```



There are a set of drawing functions that allow you to draw onto a canvas. Most drawing functions allow you to specify **x** and **y** coordinates. The coordinate (x,y)=(0,0) is at the top left corner of the canvas. Here are just some of the methods available in the **GraphicsContext** class (look in the JavaFX API for more info):

| | |
|--|---|
| | <pre>// Draw a line from (x1, y1) to (x2, y2) public void strokeLine(double x1, double y1, double x2, double y2);</pre> |
|--|---|

| | |
|---|--|
|  <p>(x,y) width height</p> | <pre>// Draw a rectangle with its top left at (x, y) having // the given width and height public void strokeRect(double x, double y, double width, double height);</pre> |
|  <p>(x,y) width height</p> | <pre>// Draw a filled rectangle with its top left at (x, y) // having the given width and height public void fillRect(double x, double y, double width, double height);</pre> |
|  <p>(x,y) width height</p> | <pre>// Draw an oval with its top left at (x, y) having the // given width and height public void strokeOval(double x, double y, double width, double height);</pre> |
|  <p>(x,y) width height</p> | <pre>// Draw a filled oval with its top left at (x, y) // having the given width and height public void fillOval(double x, double y, double width, double height);</pre> |
|  <p>(x,y) width height w h</p> | <pre>// Draw a rounded rectangle with its top left at (x, y) // having given width & height and arc width & height public void strokeRoundRect(double x, double y, double width, double height, double w, double h);</pre> |
|  <p>(x,y) width height w h</p> | <pre>// Draw a filled rounded rectangle with its top left at // (x, y) having given & height and arc width & height public void fillRoundRect(double x, double y, double width, double height, double w, double h);</pre> |
|  <p>(x,y) height width startAngle extent</p> | <pre>// Draw a filled "Open" arc with its top left at // (x, y) starting at startAngle deg and going extent public void strokeArc(double x, double y, double width, double height, double startAngle, double extent, ArcType.OPEN);</pre> |
|  <p>(x,y) height width startAngle extent</p> | <pre>// Draw a filled "Chord" arc with its top left at // (x, y) starting at startAngle deg and going extent public void strokeArc(double x, double y, double width, double height, double startAngle, double extent, ArcType.CHORD);</pre> |
|  <p>(x,y) height width startAngle extent</p> | <pre>// Draw a filled "Pie Chart" arc with its top left at // (x, y) starting at startAngle deg and going extent public void strokeArc(double x, double y, double width, double height, double startAngle, double extent, ArcType.ROUND);</pre> |
|  <p>(x,y) height width startAngle extent</p> | <pre>// Draw an "Open" or "chord" arc with its top left at // (x, y) starting at startAngle deg and going extent public void fillArc(double x, double y, double width, double height, double startAngle, double extent, ArcType.OPEN);</pre> |
|  <p>(x,y) height width startAngle extent</p> | <pre>// Draw a "Pie Chart" arc with its top left at (x, y) // starting at startAngle deg and going extent public void fillArc(double x, double y, double width, double height, double startAngle, double extent, ArcType.ROUND);</pre> |

| | |
|---|---|
|  | <pre>// Draw a polygon with the given coordinates public void strokePolygon(double[] x, double[] y, int numEdges);</pre> |
|  | <pre>// Draw a filled polygon with the given coordinates public void fillPolygon(double[] x, double[] y, int numEdges);</pre> |
|  | <pre>// Draw a polyline with the given coordinates public void strokePolyline(double[] x, double[] y, int numEdges);</pre> |
|  | <pre>// Draw the given String with its bottom left at (x, y) public void strokeText(String str, double x, double y);</pre> |
|  | <pre>// Draw the given String with its bottom left at (x, y) public void fillText(String str, double x, double y);</pre> |

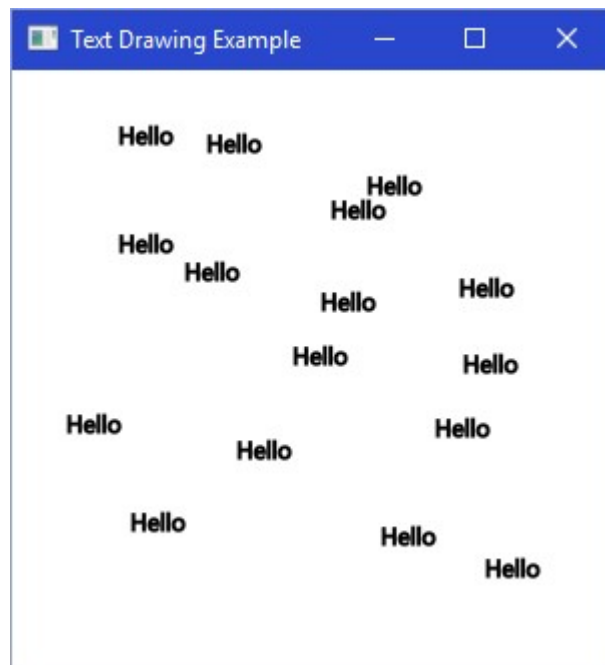
Example:

This code makes a simple **Application** and then draws some text on it wherever the user clicks the mouse. The code is straight forward. The text is drawn such that the bottom left corner of the text appears at the location in which the mouse is pressed. Here is the code:

```
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.canvas.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class TextDrawingExample extends
    Application {
    public void start(Stage primaryStage) {
        Pane p = new Pane();
        Canvas canvas = new Canvas(300, 300);
        p.getChildren().add(canvas);
        GraphicsContext aPen = canvas.getGraphicsContext2D();
        aPen.setStroke(Color.BLACK);
        aPen.setFill(Color.YELLOW);
        Scene scene = new Scene(p, 300, 300);

        // Draw a piece of text wherever the mouse is pressed
```



```

canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        aPen.strokeText("Hello", mouseEvent.getX(), mouseEvent.getY());
    }
});

primaryStage.setTitle("Text Drawing Example");
primaryStage.setScene(scene);
primaryStage.show();
}
public static void main(String[] args) { launch(args); }
}

```

Of course, we can adjust the **Font** as well. For example, we can change the font using **setFont()**:

```

aPen.setFont(Font.font("Arial", 40));
aPen.strokeText("Hello", mouseEvent.getX(), mouseEvent.getY());

```

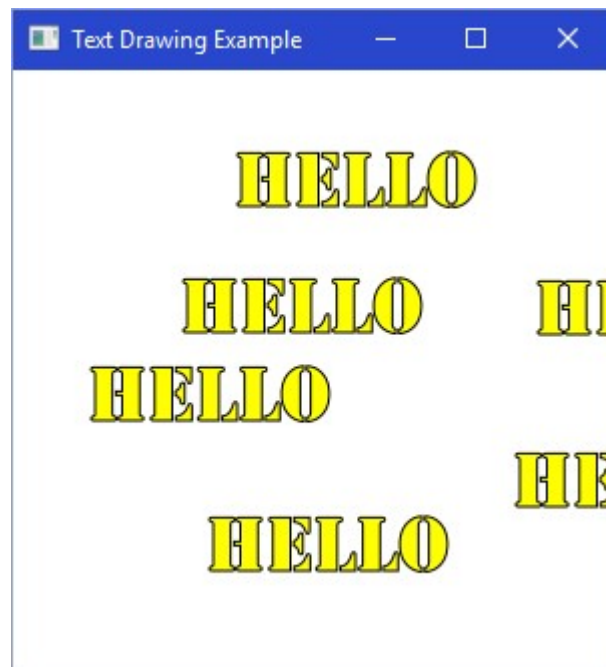
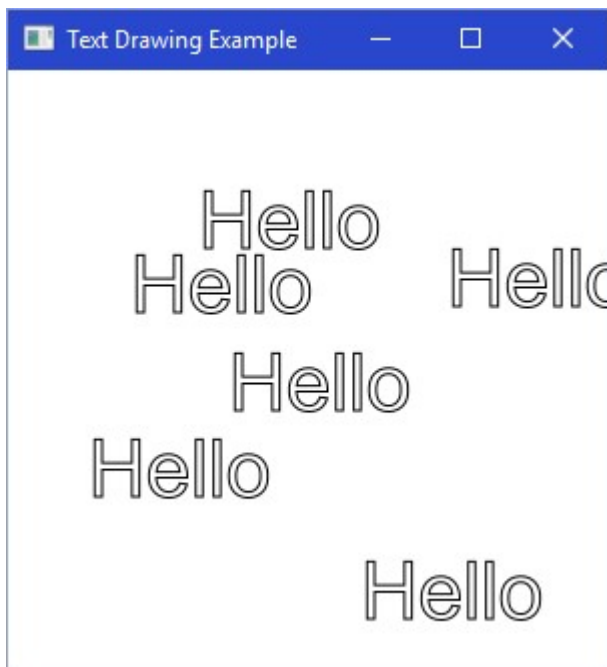
We can even mix and match things ... filling AND drawing the font:

```

aPen.setStroke(Color.BLACK);
aPen.setFill(Color.YELLOW);
aPen.setFont(Font.font("Stencil", 40));
aPen.fillText("Hello", mouseEvent.getX(), mouseEvent.getY());
aPen.strokeText("Hello", mouseEvent.getX(), mouseEvent.getY());

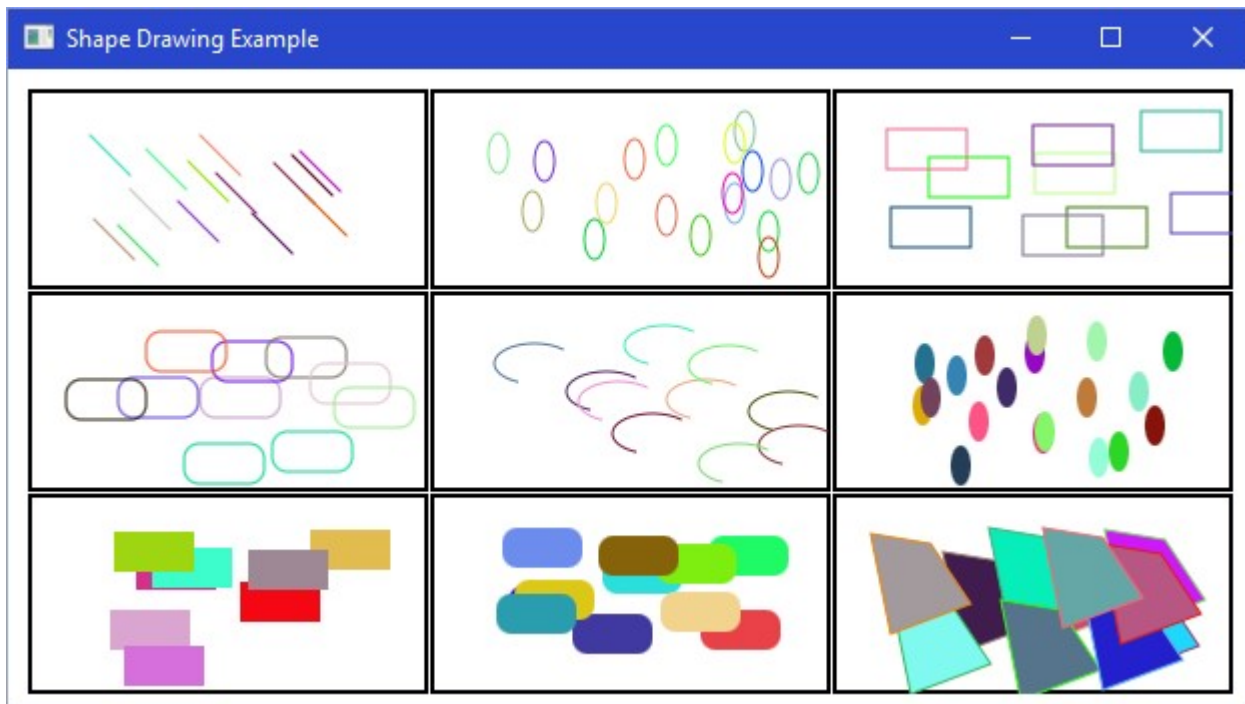
```

Here is the result of each of these:



Example:

In this example, we will set up nine **Canvases**, each one allowing a different shape to be drawn onto it. We will set up a single event handler for all mouse presses and within that method we will ask which canvas has been clicked on and then draw the corresponding shape onto the canvas. The shapes will be drawn with different colors each time. We use **Math.random()** to get a random number for creating a random color.



The code for this application is below. You will notice that:

1. We created an array of **GraphicsContext** objects to represent the 9 "pens" that correspond to the 9 canvases. This was made as an instance variable so that both the **start()** and **handlePress()** methods can access it freely. A black border is drawn around each canvas upon startup.
2. All canvases share the same event handler, so within it, we need to determine which canvas was clicked on to decide which shape to draw. We do this by comparing the 9 pens with the pen that comes in as a method parameter (which is the pen belonging to the canvas that generated the event). Then we can determine the row and column that the pen belongs to and determine which of the 9 shapes to draw.

```
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.canvas.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.ArcType;
import javafx.stage.Stage;
```

```

public class ShapeDrawingExample extends Application {
    private GraphicsContext[][] pens = new GraphicsContext[3][3];

    public void start(Stage primaryStage) {

        GridPane aPane = new GridPane();
        aPane.setPadding(new Insets(10, 10, 10, 10));
        aPane.setHgap(1);
        aPane.setVgap(1);

        for (int row=0; row<3; row++)
            for (int col=0; col<3; col++) {
                // Create and add a canvas and draw a border around it
                Canvas c = new Canvas(200, 100);
                pens[row][col] = c.getGraphicsContext2D();
                pens[row][col].setStroke(Color.BLACK);
                pens[row][col].setLineWidth(4);
                pens[row][col].strokeRect(0, 0, c.getWidth(), c.getHeight());
                pens[row][col].setLineWidth(1);
                aPane.add(c, col, row);

                // Handle a mouse press on each canvas
                c.setOnMousePressed(new EventHandler<MouseEvent>() {
                    public void handle(MouseEvent mouseEvent) {
                        handlePress(((Canvas)mouseEvent.getSource()).
                            getGraphicsContext2D(),
                            mouseEvent.getX(), mouseEvent.getY());
                    }
                });
            }

        Scene scene = new Scene(aPane, 620, 320);
        primaryStage.setTitle("Shape Drawing Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    // Handle the MousePress event on a canvas
    private void handlePress(GraphicsContext aPen, double x, double y) {
        // Get a random color
        aPen.setStroke(Color.color(Math.random(), Math.random(), Math.random()));
        aPen.setFill(Color.color(Math.random(), Math.random(), Math.random()));

        // Find the index of the canvas that caused this event
        int penNumber = 0;
        for (int row=0; row<3; row++)
            for (int col=0; col<3; col++) {
                if (aPen == pens[row][col]) {
                    penNumber = row*3 + col;
                    break;
                }
            }

        // Now decide what to draw
        switch (penNumber) {
            case 0: aPen.strokeLine(x, y, x+20, y+20); break;
            case 1: aPen.strokeOval(x, y, 10, 20); break;
            case 2: aPen.strokeRect(x, y, 40, 20); break;
            case 3: aPen.strokeRoundRect(x, y, 40, 20, 15, 15); break;
            case 4: aPen.strokeArc(x, y, 40, 20, 45, 200, ArcType.OPEN); break;
        }
    }
}

```

```

        case 5: aPen.fillOval(x, y, 10, 20); break;
        case 6: aPen.fillRect(x, y, 40, 20); break;
        case 7: aPen.fillRoundRect(x, y, 40, 20, 15, 15); break;
        case 8:
            double[] px = new double[4];
            px[0] = x;
            px[1] = x + 30;
            px[2] = x + 50;
            px[3] = x + 10;
            double[] py = new double[4];
            py[0] = y;
            py[1] = y + 5;
            py[2] = y + 35;
            py[3] = y + 50;
            aPen.fillPolygon(px, py, 4);
            aPen.strokePolygon(px, py, 4);
            break;
    }
}
public static void main(String[] args) { launch(args); }
}

```

14.2 Displaying Images and Manipulating Pixels

We have seen how to draw shapes of different colors onto components, now we will find out how to draw an image on the screen. JavaFX lets you load and display image files in a similar way in which we used them on Labels and Buttons earlier in the course. An **ImageView** object is used to display an image and the **ImageView** can be added to a **Pane** just like any other object:

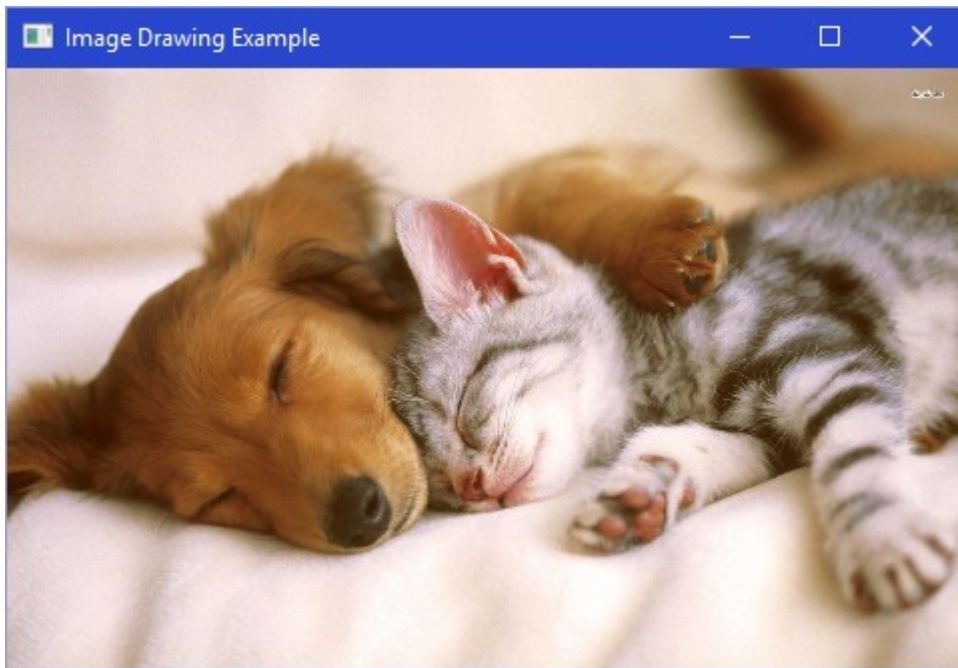
```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.*;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class ImageDrawingExample extends Application {
    public void start(Stage primaryStage) {
        Image image = new Image("KittenPuppy.png");
        ImageView imageView = new ImageView(image);

        Pane root = new Pane();
        root.getChildren().add(imageView);
        Scene scene = new Scene(root, 480, 300);
        primaryStage.setTitle("Image Drawing Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) { launch(args); }
}

```

We can also extract information from the image by accessing its pixel colors. We do this by using the `getPixelReader()` method, and then using the `getColor(x,y)` method on that to obtain the pixel's color at the specified coordinate.

```
// Determine the color of each pixel in the image
PixelReader pixelReader = image.getPixelReader();

for (int y = 0; y < image.getHeight(); y++) {
    for (int x = 0; x < image.getWidth(); x++) {
        Color color = pixelReader.getColor(x, y);

        System.out.println("\nPixel color at (" + x + "," + y + ")");
        System.out.println("R = " + color.getRed());
        System.out.println("G = " + color.getGreen());
        System.out.println("B = " + color.getBlue());
        System.out.println("Opacity = " + color.getOpacity());
        System.out.println("Saturation = " + color.getSaturation());
    }
}
```

The above code obtains color information from every pixel in the image. It reads in one pixel at a time, starting in the upper left corner (0,0) and progressing across the image from left to right. The Y-coordinate increments only after an entire row has been read. Information about each pixel is then printed. The R,G,B values represent the percentage of red, green and blue in the pixel, where (0.0, 0.0, 0.0) would be black, while (1.0, 1.0, 1.0) would be white. The output looks something like what is shown here.

```
Pixel color at (178,284)
R = 0.9843137264251709
G = 0.9411764740943909
B = 0.9254902005195618
Opacity = 1.0
Saturation = 0.059760952556502815
```

Unfortunately, **Image** objects are read-only. So, in order to change an image's pixels, we need an instance of **WritableImage** and then we can use **getPixelWriter()** to get a **PixelWriter** object that will allow us to change the pixel values:

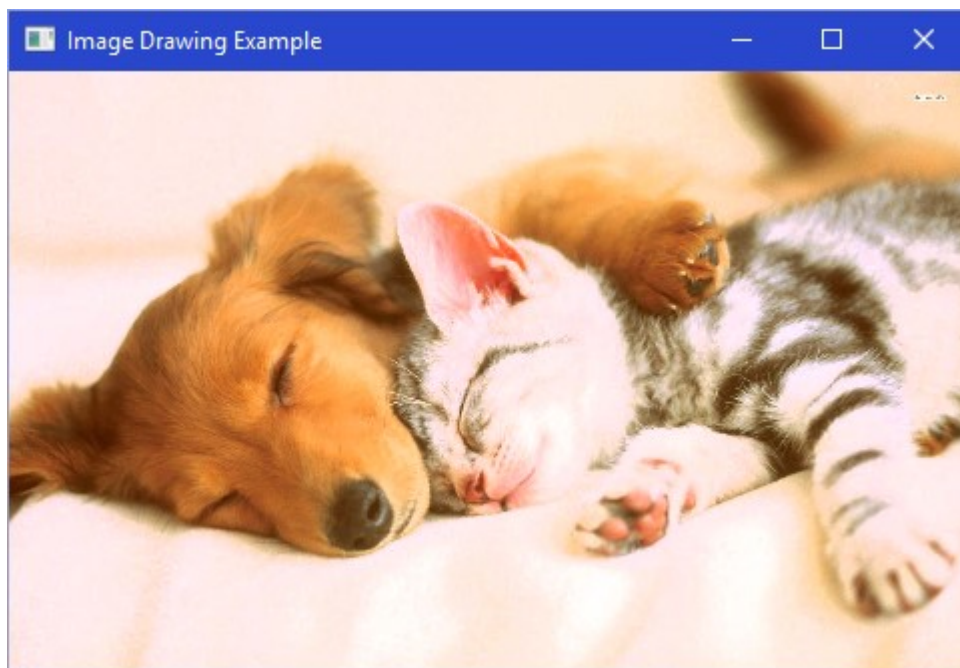
```
// Create WritableImage
WritableImage wImage = new WritableImage((int) image.getWidth(),
                                         (int) image.getHeight());
PixelWriter pixelWriter = wImage.getPixelWriter();
```

With the **PixelWriter**, we can use **setColor(x, y, newColor)** to change a pixel at location (x, y) to a newColor. Here is an example of how to brighten a pixel, which makes use of the color variable obtained from **pixelReader.getColor(x, y)**:

```
color = color.brighter();
pixelWriter.setColor(x, y, color);
```

We can apply this to each pixel to brighten the whole image. It should be noted however, that this does not change the original image. We are in fact brightening **wImage** ... which is the **WritableImage** object. If we want the changes to be visible, we would need to show the new image. One way to do this is to change the image on the **ImageView** from the original image to the **wImage** as follows:

```
imageView.setImage(wImage);
```

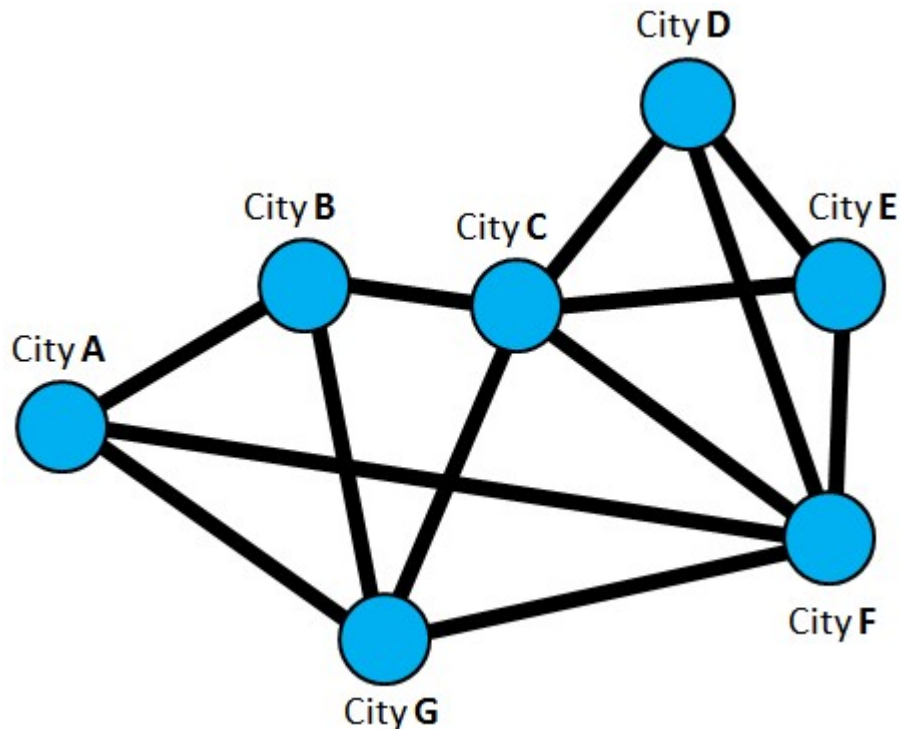


There are more efficient ways to manipulate pixels by using **setPixels()**, which allows you to define all the pixel values of an image via a **byte[]**. You can look further into the **pixelWriter** API for more details.

14.3 Graph Editor Example

This section of the notes describes a step-by-step approach for creating a simple graph editor. It introduces the notion of "drag and drop" as well as selecting objects.

What is a graph ? There are many types of graphs. We are interested in graphs that form topological and/or spatial information. Our graphs will consist of **nodes** and **edges**. The nodes may represent cities in a map while the edges may represent routes or flights between cities:



We would like to make a graph editor with the ability to:

- add/remove nodes
- add/remove edges
- move nodes around (edges between them will remain connected)
- "select" groups of nodes and edges for removal or moving
- do some other useful graph-manipulation features

The Graph Model:

We will begin our application as usual by developing the model. We know that our graph itself is going to be the model, but we must first think about what components make up the graph. These are the nodes and edges.

Let us begin by creating a **Node** class. What **state** should each node maintain ? Well, it depends on the application that will be using it. Since we know that the graph will be

displayed, each node will need to keep track of its **location**. Also, we may wish to **label** nodes (e.g., a city's name). Here is a basic model for the nodes:

```
import javafx.geometry.Point2D;

public class Node {
    private String    label;
    private Point2D   location;

    public Node() { this("", new Point2D(0,0)); }
    public Node(String aLabel) { this(aLabel, new Point2D(0,0)); }
    public Node(Point2D aPoint) { this("", aPoint); }
    public Node(double x, double y) { this("", new Point2D(x,y)); }

    public Node(String aLabel, Point2D aPoint) {
        label = aLabel;
        location = aPoint;
    }

    public String getLabel() { return label; }
    public Point2D getLocation() { return location; }
    public void setLabel(String newLabel) { label = newLabel; }
    public void setLocation(Point2D aPoint) { location = aPoint; }
    public void setLocation(double x, double y) { location = new Point2D(x, y); }

    // Nodes look like this: label(12,43)
    public String toString() {
        return(label + "(" + location.getX() + "," + location.getY() + ")");
    }
}
```

Notice that we don't have much in terms of behavior ... simply some get/set methods and a **toString()** method. Notice the two different set methods for location. This gives us flexibility in cases where the coordinates are either **Point2D** objects or **ints**.

What state do we need for a graph edge? Well ... they must *start* at some node and *end* at another so we may want to know which nodes these are. Does it make sense for a graph edge to exist when one or both of its endpoints are not nodes? Probably not. So an edge should keep track of the node from which it starts and the node at which it ends. We will call them **startNode** and **endNode**. What about a label? Sure ... roads have names (as well as lengths). Here is a basic **Edge** class:

```
public class Edge {
    private String    label;
    private Node      startNode, endNode;

    public Edge(Node start, Node end) { this("", start, end); }
    public Edge(String aLabel, Node start, Node end) {
        label = aLabel;
        startNode = start;
        endNode = end;
    }

    public String getLabel() { return label; }
    public Node getStartNode() { return startNode; }
    public Node getEndNode() { return endNode; }
}
```

```

public void setLabel(String newLabel) { label = newLabel; }
public void setStartNode(Node aNode) { startNode = aNode; }
public void setEndNode(Node aNode) { endNode = aNode; }

// Edges look like this: sNode(12,43) --> eNode(67,34)
public String toString() {
    return(startNode.toString() + " --> " + endNode.toString());
}
}

```

Now what about the graph itself? What do we need for the state of the graph?
Well ... a graph is just a bunch of nodes and edges.

Still, we have a few choices for representing the graph:

1. Keep a collection of all nodes AND another collection of all edges
2. Keep only a collection of all nodes
3. Keep only a collection of all edges
4. Keep only 1 node OR 1 edge (this seems weird doesn't it?)

Let us examine each of these:

1. The 1st strategy would provide quick access for nodes and edges since they are readily available. However, it does take more space than the other strategies.
2. The 2nd strategy allows quick access to nodes, but if we ever needed to get all the edges, we would have to build up the collection, which takes time. This can be done by iterating through all *incident edges* (i.e., incoming and outgoing edges) of all nodes and adding the edges (this is slower, but more space efficient). So each node would have to keep track of the edges from/to it.
3. The 3rd strategy is similar to the 2nd except that the edges are efficiently accessible and the nodes are not.
4. The 4th strategy is weird. If we keep one node, we would have to traverse along one of its *incident edges* to the node at the other end and continue in this manner throughout the graph in order to build up a list of all the nodes and edges. However, this will ONLY work if the graph is **connected** (i.e., every node can be reached from every other node through a sequence of graph edges).

We will choose the 2nd strategy for our implementation, although you should realize that all four are possible.

Let us examine our **Node** and **Edge** classes a little further and try to imagine additional behavior that we may want to have.

Notice that each edge keeps track of the nodes that it connects together. But shouldn't a node also keep track of the edges are connected to it? Think of "real life". Wouldn't it be nice to know which roads lead "into" and "out of" a city?

Obviously, we can always consult the graph itself and check ALL edges to see if they connect to a given city. This is NOT what you would do if you had a map though. You don't find this

information out by looking at ALL roads on a map. You find the city of interest, then look at the roads around that area (i.e., only the ones heading into/out of the city).

The point is ... for time efficiency reasons, we will probably want each node to keep track of the edges that it is connected to. Of course, we won't make copies of these edges, we will just keep "pointers" to them so the additional memory usage is not too bad.

We should go back and add the following instance variable to the **Node** class:

```
private ArrayList<Edge>    incidentEdges;
```

We will also need the following "get method" and another for adding an edge:

```
public ArrayList<Edge> incidentEdges() {
    return incidentEdges;
}

public void addIncidentEdge(Edge e) {
    incidentEdges.add(e);
}
```

We will also have to add this line to the last of the **Node** constructors:

```
incidentEdges = new ArrayList<Edge>();
```

While we are making changes to the **Node** class, we will also add another interesting method called **neighbours()** that returns the nodes that are connected to the receiver node by a graph edge. That is, it will return an **ArrayList** of all nodes that share an edge with this receiver node. It is very much like asking: "Which cities can I reach from this one if I travel on only one highway?".

We can obtain these neighbors by iterating through the **incidentEdges** of the receiver and extracting the node at the other end of the edge. We will have to determine if this other node is the start or end node of the edge:

```
public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();

    for (Edge e: incidentEdges) {
        if (e.getStartNode() == this)
            result.add(e.getEndNode());
        else
            result.add(e.getStartNode());
    }

    return result;
}
```

As we write this method, it seems that we are writing a portion of code that is *potentially* useful for other situations. That code is the code responsible for finding the opposite node of an edge. We should extract this code and make it a method for the **Edge** class:

```

public Node otherEndFrom(Node aNode) {
    if (startNode == aNode)
        return endNode;
    else
        return startNode;
}

```

Now, we can rewrite the `neighbours()` method to use the `otherEndFrom()` method:

```

public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();

    for (Edge e: incidentEdges)
        result.add(e.otherEndFrom(this));

    return result;
}

```

Ok. Now we will look at the **Graph** class. We have decided that we were going to store just the nodes, and not the edges. We will also store a label for the graph ... after all ... provinces have names don't they ?

```

import java.util.*;

public class Graph {
    private String label;
    private ArrayList<Node> nodes;

    public Graph() { this("", new ArrayList<Node>()); }
    public Graph(String aLabel) { this(aLabel, new ArrayList<Node>()); }
    public Graph(String aLabel, ArrayList<Node> initialNodes) {
        label = aLabel;
        nodes = initialNodes;
    }

    public ArrayList<Node> getNodes() { return nodes; }
    public String getLabel() { return label; }
    public void setLabel(String newLabel) { label = newLabel; }

    // Graphs look like this: label(6 nodes, 15 edges)
    public String toString() {
        return(label + "(" + nodes.size() + " nodes, " +
            getEdges().size() + " edges)");
    }
}

```

Let us write a method to return all the edges of the graph. It will have to go and collect all the **Edge** objects from the incident edges of the **Node** objects and return them as an **ArrayList**. Can you foresee a small problem ?

```
// Get all the edges of the graph by asking the nodes for them
public ArrayList<Edge> getEdges() {
    ArrayList<Edge> edges = new ArrayList<Edge>();

    for (Node n: nodes) {
        for (Edge e: n.incidentEdges()) {
            if (!edges.contains(e)) //so that it is not added twice
                edges.add(e);
        }
    }

    return edges;
}
```

Now we need methods for adding/removing nodes/edges. Adding a node or edge is easy, assuming that we already have the node or edge:

```
public void addNode(Node aNode) {
    nodes.add(aNode);
}

public void addEdge(Edge anEdge) {
    // ?????? What ?????? ...
}
```

Wait a minute ! How do we add an edge if we do not store them explicitly ? Perhaps we don't want an `addEdge` method that takes an "already created" edge. Instead, we should have an `addEdge` method that takes the **startNode** and **endNode** as parameters, then it creates the edge:

```
public void addEdge(Node start, Node end) {
    // First make the edge
    Edge anEdge = new Edge(start, end);

    // Now tell the nodes about the edge
    start.addIncidentEdge(anEdge);
    end.addIncidentEdge(anEdge);
}
```

There ... that is better. What about removing/deleting a node or edge ? Deleting an **Edge** is easy, we just ask the edge's start and end nodes to remove the edge from their lists.

```
public void deleteEdge(Edge anEdge) {
    // Just ask the nodes to remove it
    anEdge.getStartNode().incidentEdges().remove(anEdge);
    anEdge.getEndNode().incidentEdges().remove(anEdge);
}
```

Removing a **Node** is a little more involved since all of the incident edges must be removed as well. After all ... we cannot have edges dangling with one of its **Nodes** missing !


```

public void deleteNode(Node aNode) {
    // Remove the opposite node's incident edges
    for (Edge e: aNode.incidentEdges())
        e.otherEndFrom(aNode).incidentEdges().remove(e);
    nodes.remove(aNode); // Remove the node now
}

```

OK. Let us write some code that now tests the model classes. Here is **static** method for the **Graph** class that creates and returns a graph:

```

public static Graph example() {
    Graph myMap = new Graph("Ontario and Quebec");
    Node ottawa, toronto, kingston, montreal;

    myMap.addNode(ottawa = new Node("Ottawa", new Point2D(250,100)));
    myMap.addNode(toronto = new Node("Toronto", new Point2D(100,170)));
    myMap.addNode(kingston = new Node("Kingston", new Point2D(180,110)));
    myMap.addNode(montreal = new Node("Montreal", new Point2D(300,90)));
    myMap.addEdge(ottawa, toronto);
    myMap.addEdge(ottawa, montreal);
    myMap.addEdge(ottawa, kingston);
    myMap.addEdge(kingston, toronto);

    return myMap;
}

```

We can test it by writing **Graph.example()** anywhere. This looks fine and peachy, but if we have 100 nodes, we would need 100 local variables (or a big array) just for the purpose of adding edges !! Maybe this would be a better way to write the code:

```

public static Graph example() {
    Graph myMap = new Graph("Ontario and Quebec");

    myMap.addNode(new Node("Ottawa", new Point2D(250,100)));
    myMap.addNode(new Node("Toronto", new Point2D(100,120)));
    myMap.addNode(new Node("Kingston", new Point2D(200,130)));
    myMap.addNode(new Node("Montreal", new Point2D(300,70)));
    myMap.addEdge("Ottawa", "Toronto");
    myMap.addEdge("Ottawa", "Montreal");
    myMap.addEdge("Ottawa", "Kingston");
    myMap.addEdge("Kingston", "Toronto");

    return myMap;
}

```

This way, we can access the nodes of the graph by their names (assuming that they are all unique names). How can we make this happen? We just need to make another **addEdge()** method that takes two **String** arguments and finds the nodes that have those labels. Perhaps we could make a nice little helper method in the **Graph** class that will find a node with a given name (label):

```

public Node nodeNamed(String aLabel) {
    for (Node n: nodes)
        if (n.getLabel().equals(aLabel))
            return n;
    return null; // If we don't find one
}

```

Now we can write another `addEdge()` method that takes **String** parameters representing **Node** names:

```

public void addEdge(String startLabel, String endLabel) {
    Node start = nodeNamed(startLabel);
    Node end = nodeNamed(endLabel);

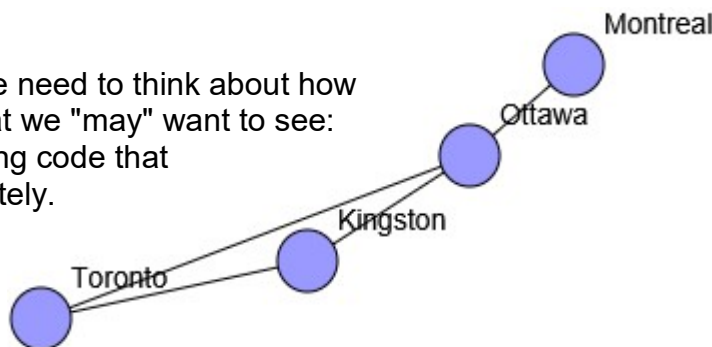
    if ((start != null) && (end != null))
        addEdge(start, end);
}

```

Notice the way we share code by making use of the "already existing" `addEdge()` method. Also notice the careful checking for valid node labels. After this new addition, the 2nd `main()` method that we created above will now work.

Displaying the Graph:

If we are going to be displaying the graph, we need to think about how we want to draw it. Here, to the right, is what we "may" want to see: So where do we start? Let us work on writing code that draws each of the graph components separately.



We will start by writing methods for drawing **Nodes** and **Edges**, then use these to draw the **Graph**. We can pass around the **GraphicsContext** object that corresponds to the "pen" that belongs to the canvas. Here is a method for the **Node** class that will instruct a **Node** to draw itself using the given **GraphicsContext** object:

```

public void draw(GraphicsContext aPen) {
    int r = 15; // Node radius

    aPen.setFill(Color.color(0.6, 0.6, 1.0)); // Fill circle around center of node
    aPen.fillOval(location.getX() - r, location.getY() - r, r*2, r*2);

    aPen.setStroke(Color.BLACK); // Draw border around the circle
    aPen.strokeOval(location.getX() - r, location.getY() - r, r*2, r*2);

    aPen.setFont(Font.font("Arial", 14)); // Draw label at top right of node
    aPen.setFill(Color.BLACK);
    aPen.fillText(label, location.getX() + r, location.getY() - r);
}

```

Notice that we draw the node twice ... once for the blue color ... once for the black border. Here is now a similar method for the **Edge** class that draws an edge:

```

public void draw(GraphicsContext aPen) {
    // Draw black line from center of startNode to center of endNode
    aPen.setStroke(Color.BLACK);
    aPen.strokeLine(startNode.getLocation().getX(),
                    startNode.getLocation().getY(),
                    endNode.getLocation().getX(),
                    endNode.getLocation().getY());
}

```

When drawing the graph, we should draw edges first, then draw the nodes on top. Why not the other way around? Here is the corresponding draw method for the **Graph** class:

```

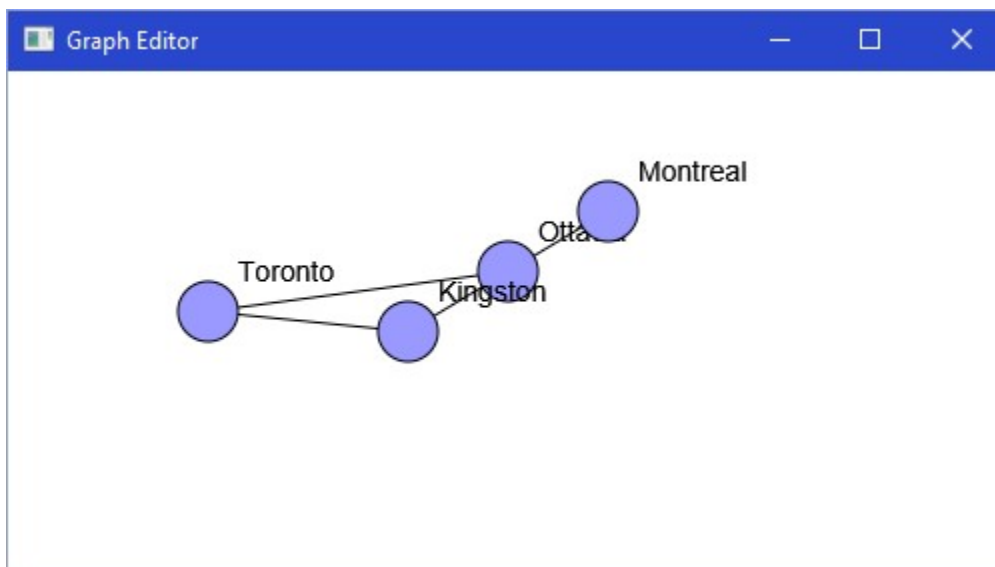
public void draw(GraphicsContext aPen) {
    ArrayList<Edge> edges = getEdges();

    for (Edge e: edges) // Draw the edges first
        e.draw(aPen);
    for (Node n: nodes) // Draw the nodes second
        n.draw(aPen);
}

```

The User Interface:

Now we can start the creation of our **GraphEditorApp** user interface. To begin, we will just create a window that will have a **Canvas** in it and we will then draw the example **Graph** from within the **start()** method:



The code to do this is straight forward. Notice that the **Graph** is stored as an instance variable and that it is drawn simply by calling the graph's **draw()** method with the canvas' pen.

```

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;

```

```

import javafx.scene.canvas.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class GraphEditorApp extends Application {
    public static int WIDTH = 500;
    public static int HEIGHT = 250;

    private Graph graph; // The model (i.e. the graph)

    public void start(Stage primaryStage) {

        graph = Graph.example();

        Pane p = new Pane();
        Canvas canvas = new Canvas(WIDTH, HEIGHT);
        p.getChildren().add(canvas);
        Scene scene = new Scene(p, WIDTH, HEIGHT);

        // Display the graph onto the canvas
        GraphicsContext aPen = canvas.getGraphicsContext2D();
        graph.draw(aPen);

        primaryStage.setTitle("Graph Editor");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

Manipulating Nodes:

What kind of action should the user perform to add a node to the graph? There are many possibilities (i.e., menu options, buttons, mouse clicks). We will allow nodes to be added to the graph via double clicks of the mouse. When the user double-clicks on the canvas, a new node will be added at that mouse-click location.

```

// Handle a double-click in the canvas
canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        if (mouseEvent.getClickCount() == 2) {
            // Change the model, by adding a new Node
            graph.addNode(new Node(mouseEvent.getX(), mouseEvent.getY()));

            // Update the view, by redrawing the Graph
            update(aPen);
        }
    }
});

```

Here is the **update()** method. It simply erases the old graph and draws the new one:

```
// Update the view by redrawing the graph onto the Canvas
private void update(GraphicsContext aPen) {
    aPen.setFill(Color.WHITE);
    aPen.fillRect(0, 0, WIDTH, HEIGHT);
    graph.draw(aPen);
}
```

If we run our code, we will notice something that is not so pleasant. Our strategy of using the double click allows us to add nodes on top of each other, making them possibly indistinguishable, as shown here on the right.

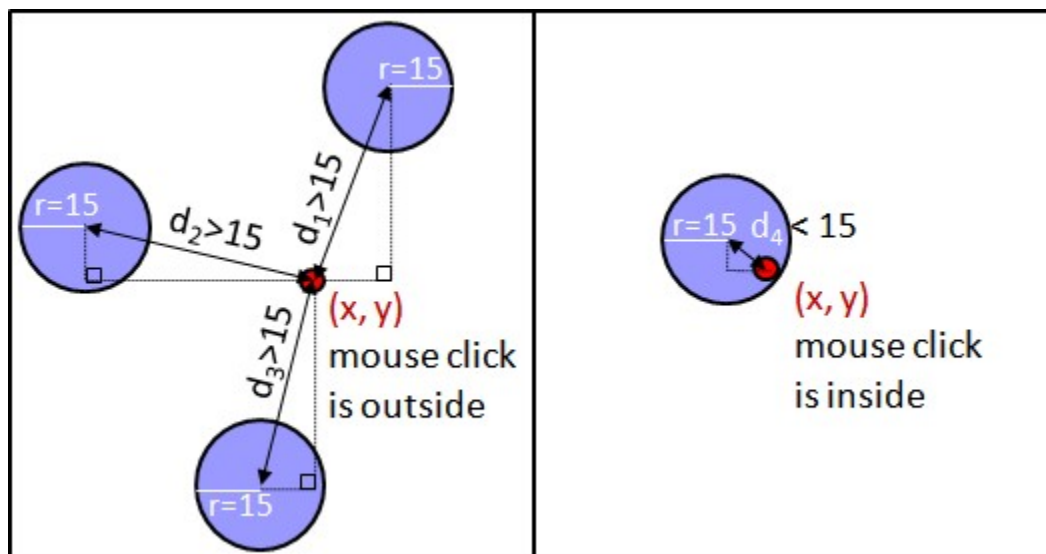


Perhaps instead of having nodes lying on top of each other, we could check to determine whether or not the user clicks within a node. Then we can decide to "not add" the node if there is already one there. What do we do then ... ignore the click? Maybe we should cause the node to be somehow "selected" so that we can move it around.

To do this, we will need to add functionality that allows nodes to be selected and unselected.

If we attempt to re-select an "already selected" node, it should probably become unselected (i.e., toggle on/off). We should make the node appear different as well (perhaps red). We will need to detect which node has been selected. This sounds like it could be a nice little helper method in the **Graph** class.

We can just check the distance from the given point to the center of all nodes. If the distance is \leq the radius, then we are inside that node.



In fact, we are not really computing the distance, we are computing the "square" of the distance. This is more efficient since we do not need to compute a square root. We need to add the following to the **Graph** class:

```
// Return the first node in which point p is contained, if none, return null
public Node nodeAt(double x, double y) {
    for (Node n: nodes) {
        Point2D c = n.getLocation();
        double d = (x-c.getX())*(x-c.getX()) + (y-c.getY())*(y-c.getY());
    }
}
```

```

        if (d <= (15*15))
            return n;
    }
    return null;
}

```

The 15 looks like a "magic" number. It seems like this number may be used a lot. We should define a static constant in the **Node** class. Go back and change the draw method as well to use this new static value:

```

public static int    RADIUS = 15;

```

Here is the better code:

```

public Node nodeAt(double x, double y) {
    for (Node n: nodes) {
        Point2D c = n.getLocation();
        double d = (x-c.getX())*(x-c.getX()) + (y-c.getY())*(y-c.getY());
        if (d <= (Node.RADIUS*Node.RADIUS))
            return n;
    }
    return null;
}

```

We should go back into our drawing routines and adjust the code so that it uses this new RADIUS constant.

One more point ... the code goes through the Nodes from the first to the last. It would be better to go in reverse order, because our drawing routine will draw the first one first and the last one last. That means, that if two Nodes overlap, the one that was last added will appear "on top" ... and this is likely the one that we want to select. So, this is better code:

```

public Node nodeAt(double x, double y) {
    for (int i=nodes.size()-1; i>=0; i--) {
        Node n = nodes.get(i);
        Point2D c = n.getLocation();
        double d = (x - c.getX()) * (x - c.getX()) +
            (y - c.getY()) * (y - c.getY());
        if (d <= (Node.RADIUS*Node.RADIUS))
            return n;
    }
    return null;
}

```

Now since we are allowing **Nodes** to be selected, we will have to somehow keep track of all the selected nodes. We have two choices:

- Let the graph keep track of the selected nodes separately
- Let each node keep track of whether or not it is selected

We will choose the second strategy (do you understand the tradeoffs of each ?).

Add the following instance variable and methods to the **Node** class:

```

private boolean    selected;

public boolean isSelected() { return selected; }
public void setSelected(boolean state) { selected = state; }
public void toggleSelected() { selected = !selected; }

```

Now we should modify the draw method to allow nodes to be selected and unselected:

```

// Draw the Node using the given pen
public void draw(GraphicsContext aPen) {
    // Draw a blue-filled circle around the center of the node
    if (selected)
        aPen.setFill(Color.RED);
    else
        aPen.setFill(Color.color(0.6, 0.6, 1.0));

    aPen.fillOval(location.getX() - RADIUS, location.getY() - RADIUS,
                  RADIUS*2, RADIUS*2);

    // Draw a black border around the circle
    aPen.setStroke(Color.BLACK);
    aPen.strokeOval(location.getX() - RADIUS, location.getY() - RADIUS,
                   RADIUS*2, RADIUS*2);

    // Draw a label at the top right corner of the node
    aPen.setFont(Font.font("Arial", 14));
    aPen.setFill(Color.BLACK);
    aPen.fillText(label, location.getX() + RADIUS, location.getY() - RADIUS);
}

```

To make it all work, we must now select any Nodes that are clicked on by updating the code in the `mouseClicked` event handler of the `GraphEditorApp`:

```

// Handle a double-click in the canvas
canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        if (ev.getClickCount() == 2) {
            Node aNode = graph.nodeAt(ev.getX(), ev.getY());
            if (aNode == null)
                // Change the model, by adding a new Node
                graph.addNode(new Node(ev.getX(), ev.getY()));
            else
                aNode.toggleSelected();
            // Update the view, by redrawing the Graph
            update(aPen);
        }
    }
});

```

Now how do we allow nodes to be deleted? Perhaps, the user must select the node(s) first and then press the **delete** key. Perhaps when the **delete** key is pressed, ALL of the currently selected nodes should be deleted. So we will make a method that first returns all the selected nodes. We will need to add this method to the `Graph` class which returns an `ArrayList<Node>` of all the selected nodes:

```
// Get all the nodes that are selected
public ArrayList<Node> selectedNodes() {
    ArrayList<Node> selected = new ArrayList<Node>();
    for (Node n: nodes)
        if (n.isSelected())
            selected.add(n);
    return selected;
}
```

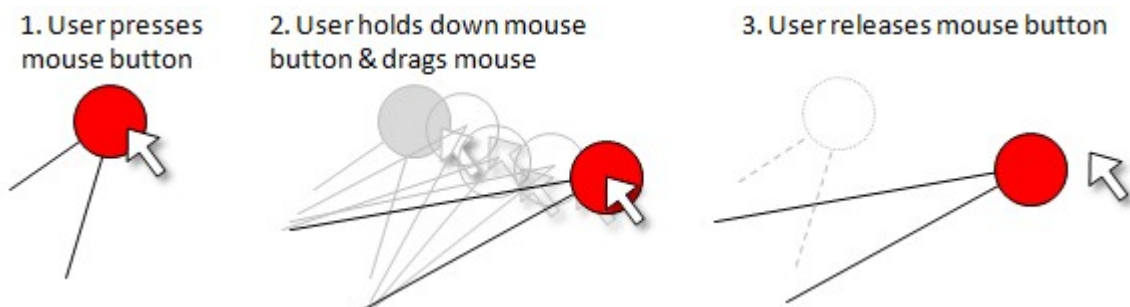
We already took care of the node selection, now we must handle the **delete** key. We should have the **GraphEditorApp** implement a **KeyReleased** event handler. We will delete all selected Nodes and then update as follows:

```
canvas.setOnKeyReleased(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ev) {
        if (ev.getCode() == KeyCode.DELETE) {
            // Delete all selected Nodes
            for (Node n: graph.selectedNodes())
                graph.deleteNode(n);
            // Update the view, by redrawing the Graph
            update(aPen);
        }
    }
});
```

There is a SLIGHT problem. It seems that even though we have only one component in our app, this component does not have the **focus** by default. In order for the keystrokes to be detectable, the component **MUST** have the focus. So we will add the following line to the **start()** method, perhaps before setting the event handler:

```
canvas.requestFocus(); // Needed to handle key events
```

Now, how can we move nodes around once they are created? Once again, we must decide how we want the interface to work. It is most natural to allow the user to move nodes by pressing the mouse down while on top of a node and holding it down while dragging the node to the new location, then release the mouse button to cause the node to appear in the new location. We will need **mousePressed** and **mouseDragged** event handlers, respectively. Here is what we will have to do:



When the user presses the mouse (i.e., a "press", not a "click"), then determine if he/she pressed on top of a node. If yes, then remember this node as being the one selected,

otherwise do nothing. As the mouse moves (while button being held down), we must update the chosen node's location.

We will have to remember which node is being dragged so that we can keep changing its location as the mouse is dragged. We will add an instance variable in the **GraphEditorApp** called **dragNode** to remember this node:

```
private Node    dragNode;
```

Here is the updated `mousePressed` handler and the new `mouseReleased` and `mouseDragged` event handlers which must be written:

```
// Handle a mouse-press in the canvas
canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        Node    aNode = graph.nodeAt(ev.getX(), ev.getY());
        if (ev.getClickCount() == 2) {
            if (aNode == null)
                // Change the model, by adding a new Node
                graph.addNode(new Node(ev.getX(), ev.getY()));
            else
                aNode.toggleSelected();
            // Update the view, by redrawing the Graph
            update(aPen);
        }
        else {
            if (aNode != null) {
                dragNode = aNode; // If we pressed on a node, store it
            }
        }
    }
});
```

```
// Handle a mouse-release in the canvas
canvas.setOnMouseReleased(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        dragNode = null; // We must have let go of a Node
    }
});
```

```
// Handle a mouse-drag in the canvas
canvas.setOnMouseDragged(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        if (dragNode != null)
            dragNode.setLocation(new Point2D(ev.getX(), ev.getY()));
        update(aPen); // We have changed the model, so now update
    }
});
```

Notice that the pressing of the mouse merely stores the node to be moved. The releasing of the mouse button merely resets this stored node to **null**. All of the moving occurs in the drag event handler. If we drag the mouse, we just make sure that we had first clicked on a node by examining the stored node just mentioned. If this stored node is not **null**, we then update its position and then update the rest of the graph. Notice that all the edges connected to a node move along with the node itself. Can you explain why?

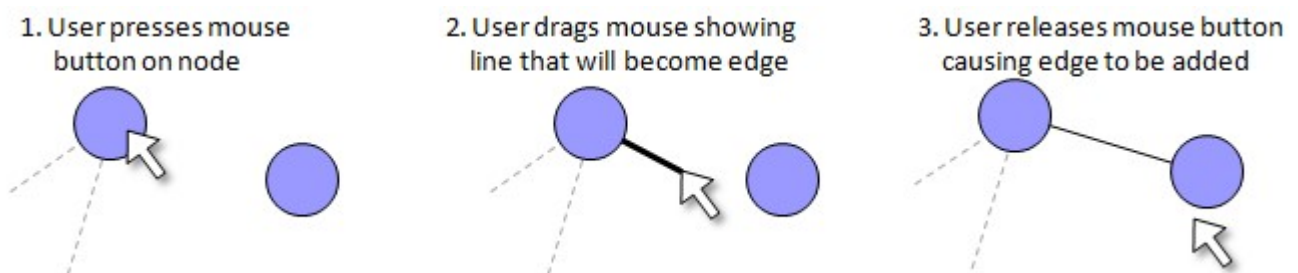
Manipulating Edges:

We have exhausted almost all the fun out of manipulating the graph nodes and we are now left with the "fun" of adding/deleting/selecting and moving edges. First we will consider adding edges. We must decide again on what action the user needs to perform in order to add the edge:

1. We can have the user double-click on the **startNode**, double click on the **endNode** and then have the edge "suddenly" appear.
2. We can select any two nodes of the graph and then perform some user-action (menu item, button press, triple click) to cause an edge to appear between the two selected edges.
3. We can click on a node and then drag the mouse to the destination node while showing the created edge as we drag the mouse.

I hope you will agree that the 3rd approach is nicer in that it is more intuitive and provides the user with a nice user-friendly interface. We will see that this strategy is called **elastic banding**. To start, we will need to make the following assumptions:

- When the user presses and holds the mouse button down on a node, this node becomes the **startNode** for the edge to be created. As the user moves the mouse (i.e., **mouseDragged** event) a line should be drawn from this **startNode** to the current mouse position. When the user lets go of the mouse button on top of a different node, an edge is created between the two.
- We should **abort** the process of adding an edge if the user releases the mouse button while: a) not on a node or b) on the same node as he/she started.



We will have to modify the **mousePressed**, **mouseReleased** and **mouseDragged** event handlers.

As it turns out, the **mousePressed** event handler already stores the "start" node in the **dragNode** variable. But now look at the **mouseDragged** event handler. Currently, if we press the mouse on a node and then drag it, this will end up causing the node to be moved. But we need to allow an elastic band edge to be drawn instead of moving the node. So, we now have two behaviors that we want to do from the same action of pressing the mouse on a node. This presents a conflict since we cannot do both behaviors.

Let us modify our node-moving behavior as follows:

- If the node initially clicked on is a **selected** node, only then we will move it, otherwise we will assume that an edge is to be added.

The **mousePressed** event handler currently just stores the selected node, upon a single press. There is really nothing more to do there.

But now during the **mouseDragged** event handler, we will have to make a decision so as to either move the node (if it was a "selected" Node) or to merely draw an edge from the pressed node to the current mouse location. We will need to draw this "elastic line" within our **update()** method after we draw the Graph. Although the start location for the elastic line is the center of the start Node (i.e. the `dragNode`), we will need to also know the location of the other end of the elastic, which is the current mouse location while we are dragging. Since we need this information within a different method outside of the event handler, we will need to store it in an instance variable. All we will do here is just store the current mouse location in this instance variable and use it within the **update()** method:

```
private Point2D elasticEndLocation;
```

Here are the new changes:

```
// Handle a mouse-drag in the canvas
canvas.setOnMouseDragged(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        elasticEndLocation = new Point2D(ev.getX(), ev.getY());
        if (dragNode != null) {
            if (dragNode.isSelected())
                dragNode.setLocation(elasticEndLocation);
        }
        update(aPen); // We have changed the model, so now update
    }
});
```

Notice how the **elasticEndLocation** is updated and stored as the mouse is dragged.

Here is the new **update()** method with the added elastic line being drawn:

```
// Update the view by redrawing the graph onto the Canvas
private void update(GraphicsContext aPen) {
    aPen.setFill(Color.WHITE);
    aPen.fillRect(0, 0, WIDTH, HEIGHT);
    graph.draw(aPen);

    // Draw the elastic band
    if (dragNode != null)
        if (!dragNode.isSelected())
            aPen.strokeLine(dragNode.getLocation().getX(),
                dragNode.getLocation().getY(),
                elasticEndLocation.getX(),
                elasticEndLocation.getY());
}
```

Notice that this method makes use of the **dragNode** and **elasticEndLocation** variables but still needs to decide whether or not to draw the elastic line. We draw the elastic line ONLY if we are adding an edge. How do we know we are adding an edge? Well, we must have pressed on a starting node, so the **dragNode** must not be **null**. Also, that **dragNode** must not be selected, otherwise we are in the middle of a "node moving" operation, not an "edge adding" one.

Our last piece to this trilogy of event handler changes is to have the **mouseReleased** event handler add the new edge ONLY if we let go of the mouse button on top of a node that is not the same as the one we started with. If it is, or we let go somewhere off of a node, then we must redraw everything either way to erase the elastic band. We just need to check that there was indeed a starting **Node** (i.e., **dragNode != null**) and that we release on a different **Node**:

```
// Handle a mouse-release in the canvas
canvas.setOnMouseReleased(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        Node aNode = graph.nodeAt(ev.getX(), ev.getY());
        // Check to see if we have let go on a node
        if ((dragNode != null) && (aNode != null) && (aNode != dragNode))
            // Change the model, by adding a new Edge
            graph.addEdge(dragNode, aNode);
        // Update the view, by redrawing the Graph
        dragNode = null; // No need to remember this anymore
        update(aPen);
    }
});
```

One of our last tasks is to allow edges to be selected and removed. We can similarly add an instance variable and some methods to the **Edge** class:

```
private boolean selected;

public boolean isSelected() { return selected;}
public void setSelected(boolean state) { selected = state;}
public void toggleSelected() { selected = !selected;}
```

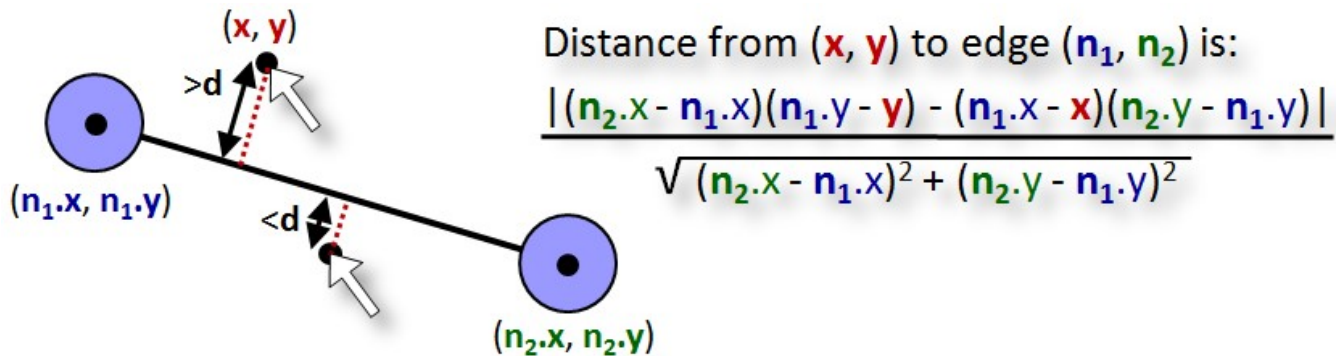
Of course ... again we should initialize the instance variable to **false** within the constructor. Now we make selected edges appear different (i.e., red).

```
public void draw(GraphicsContext aPen) {
    // Draw line from center of startNode to center of endNode
    if (selected)
        aPen.setStroke(Color.RED);
    else
        aPen.setStroke(Color.BLACK);

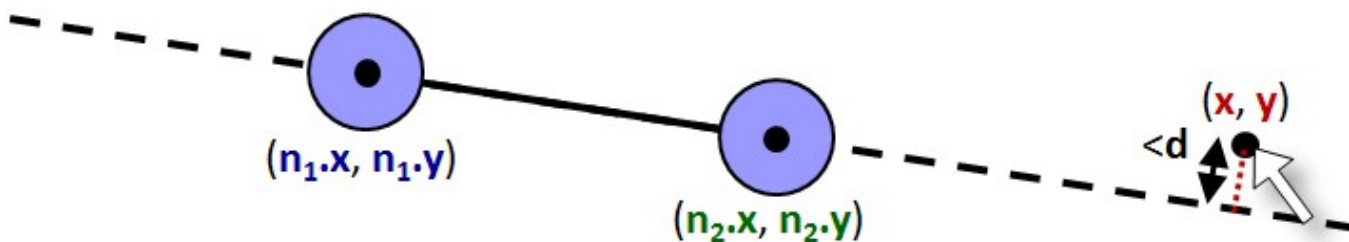
    aPen.strokeLine(startNode.getLocation().getX(),
                    startNode.getLocation().getY(),
                    endNode.getLocation().getX(),
                    endNode.getLocation().getY());
}
```

How does the user select an edge? Likely, by clicking on or near it. We can accomplish this by determining the distance between the point clicked at and the edge itself.

If the distance is smaller than some pre-decided value (e.g., 5 pixels) then we can assume that this edge was just clicked on... otherwise we can assume that the edge was not clicked on. The equation to find the distance from a point to an edge is indicated below:



However, the above equation actually computes the distance from (x, y) to the **line** that passes through the two edge nodes. So, if we click anywhere close to that line, we will be a small distance value and we will think that the edge was selected:



Certainly, we do not want such an (x, y) point to be considered as "close to" the edge. We can avoid this problem situation by examining the **x**-coordinate of the point that the user clicked on. The **x**-coordinate must be greater than the left node's **x**-coordinate and smaller than the right node's **x**-coordinate.

```

IF (distance < 3) THEN {
    IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
        this edge has been selected
}

```

Can you foresee any further problems with the algorithm? What if the edge is vertical? The above checking will never select the edge! Instead, if the edge is vertical, we should compare the **y**-coordinates. In fact, if the line is "more horizontal" we should check the **x**-coordinates and if it is "more vertical" we should check the **y**-coordinates.

To determine if a line segment is more vertical or horizontal, we can compare the difference in **x** and the difference in **y**.

Here is the code:

```

IF (distance < 3) THEN {
    xDiff ← abs(n2.x - n1.x)
    yDiff ← abs(n2.y - n1.y)
    IF (xDiff > yDiff) THEN
        IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
            this edge has been selected
        OTHERWISE
            IF ((y > n1.y) AND (y < n2.y)) OR ((y > n2.y) AND (y < n1.y)) THEN
                this edge has been selected
    }
}

```

Add the following method to the **Graph** class:

```

// Return first edge in which point p is near midpoint; if none, return null
public Edge edgeAt(double x, double y) {
    for (Edge e: getEdges()) {
        Node n1 = e.getStartNode();
        Node n2 = e.getEndNode();
        double xDiff = n2.getLocation().getX() - n1.getLocation().getX();
        double yDiff = n2.getLocation().getY() - n1.getLocation().getY();
        double distance = Math.abs(xDiff*(n1.getLocation().getY() - y) -
            (n1.getLocation().getX() - x)*yDiff) /
            Math.sqrt(xDiff*xDiff + yDiff*yDiff);
        if (distance <= 5) {
            if (Math.abs(xDiff) > Math.abs(yDiff)) {
                if (((x < n1.getLocation().getX()) &&
                    (x > n2.getLocation().getX())) ||
                    ((x > n1.getLocation().getX()) &&
                    (x < n2.getLocation().getX())))
                    return e;
            }
            else
                if (((y < n1.getLocation().getY()) &&
                    (y > n2.getLocation().getY())) ||
                    ((y > n1.getLocation().getY()) &&
                    (y < n2.getLocation().getY())))
                    return e;
        }
    }
    return null;
}

```

Now, upon a double click, we must check for edges. We will first check to see if we clicked on a node, then if we find that we did not click on a node, we will check to see if we clicked on an edge. So, we must go back and make changes to the **mousePressed** event handler:

```

// Handle a mouse-press in the canvas
canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        Node aNode = graph.nodeAt(ev.getX(), ev.getY());
        if (ev.getClickCount() == 2) {
            if (aNode == null) {
                // We missed a node, now try for an edge midpoint
                Edge anEdge = graph.edgeAt(ev.getX(), ev.getY());
                if (anEdge == null)
                    graph.addNode(new Node(ev.getX(), ev.getY()));
                else
                    anEdge.toggleSelected();
            }
            else
                aNode.toggleSelected();

            // Update the view, by redrawing the Graph
            update(aPen);
        }
        else {
            if (aNode != null) {
                dragNode = aNode; // If we pressed on a node, store it
            }
        }
    }
});

```

We can change the `keyPressed` event handler to delete all selected **Nodes AND Edges**. Of course, we will need a method to get the "selected" edges in the **Graph** class first:

```

// Get all the edges that are selected
public ArrayList<Edge> selectedEdges() {
    ArrayList<Edge> selected = new ArrayList<Edge>();
    for (Edge e: getEdges())
        if (e.isSelected())
            selected.add(e);
    return selected;
}

```

```

canvas.setOnKeyReleased(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ev) {
        if (ev.getCode() == KeyCode.DELETE) {
            // Delete all selected Edges
            for (Edge e: graph.selectedEdges())
                graph.deleteEdge(e);
            // Delete all selected Nodes
            for (Node n: graph.selectedNodes())
                graph.deleteNode(n);
            // Update the view, by redrawing the Graph
            update(aPen);
        }
    }
});

```

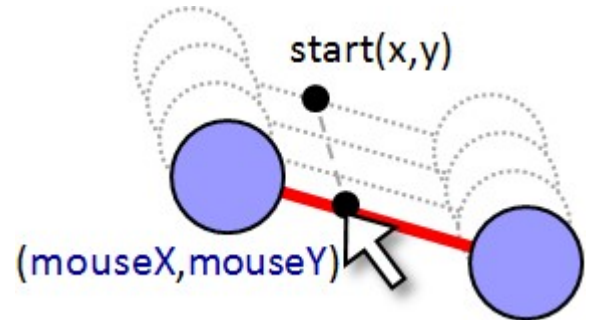
14.4 Adding Features to the Graph Editor

We have implemented a basic graph editor. There are many features that can be added. Below are solutions to some added features to the **GraphEditor**. You may want to try to add these features yourself without looking at the solutions.

Dragging Edges:

- Add the following two instance variables to the **GraphEditor** class:

```
private Edge      dragEdge;
private Point2D   dragPoint;
```



- Add code to the **mousePressed** event handler in the **GraphEditorApp** class to store the edge to be dragged:

```
canvas.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        Node aNode = graph.nodeAt(ev.getX(), ev.getY());
        if (ev.getClickCount() == 2) {
            if (aNode == null) {
                Edge anEdge = graph.edgeAt(ev.getX(), ev.getY());
                if (anEdge == null)
                    graph.addNode(new Node(ev.getX(), ev.getY()));
                else
                    anEdge.toggleSelected();
            }
            else
                aNode.toggleSelected();

            update(aPen);
        }
        else {
            if (aNode != null) {
                dragNode = aNode;
                dragEdge = null;
            }
            else
                dragEdge = graph.edgeAt(ev.getX(), ev.getY());
                dragPoint = new Point2D(ev.getX(), ev.getY());
        }
    }
});
```

- Add code to the **mouseDragged** event handler in the **GraphEditorApp** class to store the edge to be dragged:

```
canvas.setOnMouseDragged(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent ev) {
        elasticEndLocation = new Point2D(ev.getX(), ev.getY());
        if (dragNode != null) {
            if (dragNode.isSelected())
```



```

        dragNode.setLocation(elasticEndLocation);
    }
    if (dragEdge != null) {
        if (dragEdge.isSelected()) {
            dragEdge.getStartNode().setLocation(
                dragEdge.getStartNode().getLocation().getX() +
                    ev.getX() - dragPoint.getX(),
                dragEdge.getStartNode().getLocation().getY() +
                    ev.getY() - dragPoint.getY());
            dragEdge.getEndNode().setLocation(
                dragEdge.getEndNode().getLocation().getX() +
                    ev.getX() - dragPoint.getX(),
                dragEdge.getEndNode().getLocation().getY() +
                    ev.getY() - dragPoint.getY());
            dragPoint = new Point2D(ev.getX(), ev.getY());
        }
    }
    update(aPen);
}
});

```

Moving Multiple Nodes:

- Add the following instance variable to the **GraphEditor** class (if not already there):

```
private Point2D dragPoint;
```

- Add the following line at the bottom of the **mousePressed** event handler in the **GraphEditorApp** class (if not already there):

```
dragPoint = new Point2D(ev.getX(), ev.getY());
```

- In the **mouseDragged** event handler for the **GraphEditorApp** class, change

```

if (dragNode != null) {
    if (dragNode.isSelected())
        dragNode.setLocation(elasticEndLocation);
}

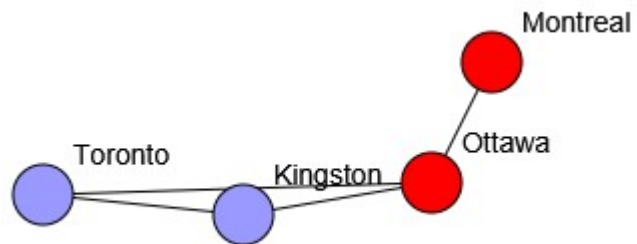
```

to this:

```

if (dragNode != null) {
    if (dragNode.isSelected()) {
        for (Node n: graph.selectedNodes()) {
            n.setLocation(n.getLocation().getX()+ev.getX()-dragPoint.getX(),
                n.getLocation().getY()+ev.getY()-dragPoint.getY());
        }
        dragPoint = new Point2D(ev.getX(), ev.getY());
    }
}

```



Drawing Selected Edges with Different Thicknesses:

- Add the following instance variable to the **Edge** class:

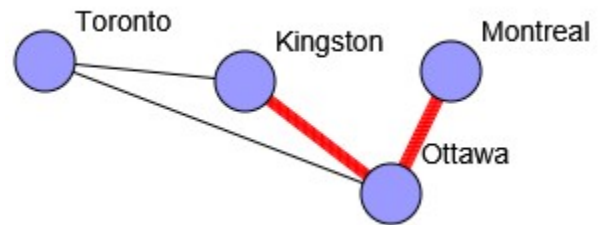
```
public static final int WIDTH = 7;
```

Modify the **draw()** method in the **Edge** class:

```
public void draw(GraphicsContext aPen) {
    // Draw black line from center of startNode to center of endNode
    if (selected) {
        aPen.setStroke(Color.RED);

        double xDiff = Math.abs(startNode.getLocation().getX() -
                                endNode.getLocation().getX());
        double yDiff = Math.abs(startNode.getLocation().getY() -
                                endNode.getLocation().getY());

        for (int i = -WIDTH/2; i <= WIDTH/2; i++) {
            if (yDiff > xDiff)
                aPen.strokeLine(startNode.getLocation().getX()+i,
                                startNode.getLocation().getY(),
                                endNode.getLocation().getX()+i,
                                endNode.getLocation().getY());
            else
                aPen.strokeLine(startNode.getLocation().getX(),
                                startNode.getLocation().getY()+i,
                                endNode.getLocation().getX(),
                                endNode.getLocation().getY()+i);
        }
    }
    else {
        aPen.setStroke(Color.BLACK);
        aPen.strokeLine(startNode.getLocation().getX(),
                        startNode.getLocation().getY(),
                        endNode.getLocation().getX(),
                        endNode.getLocation().getY());
    }
}
```



Loading and Saving Graphs:

- Add the following methods to the **Node** class:

```
// Save node to given file. Note that incident edges are not saved.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);
    aFile.println((int) location.getX());
    aFile.println((int) location.getY());
    aFile.println(selected);
}
```

```

// Load a node from given file. Note that incident edges are not connected
public static Node loadFrom(BufferedReader aFile) throws IOException {
    Node    aNode = new Node();

    aNode.setLabel(aFile.readLine());
    aNode.setLocation(Integer.parseInt(aFile.readLine()),
                      Integer.parseInt(aFile.readLine()));
    aNode.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());
    return aNode;
}

```

Add the following methods to the **Edge** class:

```

// Save edge to given file. Note that nodes themselves are not saved.
// We assume here that node locations are unique identifiers for the nodes.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);
    aFile.println((int)startNode.getLocation().getX());
    aFile.println((int)startNode.getLocation().getY());
    aFile.println((int)endNode.getLocation().getX());
    aFile.println((int)endNode.getLocation().getY());
    aFile.println(selected);
}

```

```

// Load an edge from given file. Note that nodes themselves are not loaded.
// We actually make temporary nodes here that don't correspond to actual
// graph nodes that this edge connects. We'll have to throw out these TEMP
// nodes later and replace them with graph nodes that connect to this edge.
public static Edge loadFrom(BufferedReader aFile) throws IOException {
    Edge    anEdge;
    String  aLabel = aFile.readLine();
    Node    start = new Node("TEMP");
    Node    end = new Node("TEMP");

    start.setLocation(Integer.parseInt(aFile.readLine()),
                     Integer.parseInt(aFile.readLine()));
    end.setLocation(Integer.parseInt(aFile.readLine()),
                  Integer.parseInt(aFile.readLine()));

    anEdge = new Edge(aLabel, start, end);
    anEdge.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());

    return anEdge;
}

```

- Add the following methods to the **Graph** class:

```

// Save the graph to the given file.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);

    // Output the nodes
    aFile.println(nodes.size());
    for (Node n: nodes)
        n.saveTo(aFile);
}

```

```

    // Output the edges
    ArrayList<Edge> edges = getEdges();
    aFile.println(edges.size());
    for (Edge e: edges)
        e.saveTo(aFile);
}

```

```

// Load a Graph from the given file. After the nodes and edges are loaded,
// We'll have to go through and connect the nodes and edges properly.
public static Graph loadFrom(BufferedReader aFile) throws IOException {
    // Read the label from the file and make the graph
    Graph aGraph = new Graph(aFile.readLine());

    // Get the nodes and edges
    int numNodes = Integer.parseInt(aFile.readLine());
    for (int i=0; i<numNodes; i++)
        aGraph.addNode(Node.loadFrom(aFile));

    // Now connect them with new edges
    int numEdges = Integer.parseInt(aFile.readLine());
    for (int i=0; i<numEdges; i++) {
        Edge tempEdge = Edge.loadFrom(aFile);
        Node start = aGraph.nodeAt(
            tempEdge.getStartNode().getLocation().getX(),
            tempEdge.getStartNode().getLocation().getY());
        Node end = aGraph.nodeAt(tempEdge.getEndNode().getLocation().getX(),
            tempEdge.getEndNode().getLocation().getY());
        aGraph.addEdge(start, end);
    }
    return aGraph;
}

```

- Add this instance variable to the **GraphEditorApp** class:

```
private GraphicsContext aPen;
```

and alter the code in the **start()** method to use this variable:

```
aPen = canvas.getGraphicsContext2D();
```

Then remove the `GraphicsContext aPen` parameter in the **update()** method and make use of this new instance variable from within the method. Also, remove the parameter in all the calls to **update()**. The Canvas' pen is now available from everywhere in the code.

- Add the following methods to the **GraphEditorApp** class:

```
public Graph getGraph() { return graph; }
public void setGraph(Graph g) { graph = g; update(); }
```

- In the **start()** menu of the **GraphEditorApp** class, change the **Pane** to be a **VBox**:

```
VBox p = new VBox();
```

Then increase the size of the scene by 25:

```
Scene scene = new Scene(p, WIDTH, HEIGHT+25);
```

And add the following to the top of the **start()** menu before the **Canvas** is created and added:

```
// Create the File menu
Menu fileMenu = new Menu("_File");
MenuItem loadItem = new MenuItem("Load");
loadItem.setAccelerator(KeyCombination.keyCombination("Ctrl+L"));
MenuItem saveItem = new MenuItem("Save");
saveItem.setAccelerator(KeyCombination.keyCombination("Ctrl+S"));
fileMenu.getItems().addAll(loadItem, saveItem);

// Set up the event handlers for the File menu
loadItem.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
    }
});
saveItem.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
    }
});

// Add the menus to a menubar and then add the menubar to the pane
MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(fileMenu);
p.getChildren().add(menuBar);
```

Now, we need to insert the code for the **Load** and **Save** menu selections. Insert the code into the event handlers given above as shown below:

```
loadItem.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
        FileChooser chooser = new FileChooser();
        chooser.setInitialDirectory(new File("C:\\"));
        chooser.setTitle("Load Graph");
        File f = chooser.showOpenDialog(primaryStage);
        if (f != null) {
            try {
                BufferedReader file = new BufferedReader(
                    new FileReader(f.getAbsolutePath()));
                graph = Graph.loadFrom(file);
                file.close();
                update();
            }
            catch (Exception ex) {
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error !");
                alert.setHeaderText(null);
                alert.setContentText("Error Loading Graph From File !");
                alert.showAndWait();
            }
        }
    }
});
```

```

saveItem.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
        FileChooser chooser = new FileChooser();
        chooser.setInitialDirectory(new File("C:\\"));
        chooser.setTitle("Save Graph");
        File f = chooser.showSaveDialog(primaryStage);
        if (f != null) {
            try {
                PrintWriter file = new PrintWriter(
                    new FileWriter(f.getAbsolutePath()));
                graph.saveTo(file);
                file.close();
            }
            catch (Exception ex) {
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error !");
                alert.setHeaderText(null);
                alert.setContentText("Error Saving Graph To File !");
                alert.showAndWait();
            }
        }
    }
});

```

Other Features:

There are also other features we can add. Feel free to experiment:

- Allow all selected edges and nodes to be moved by dragging a selected edge.
- Press <CNTRL><A> to select all nodes and edges and <CNTRL><U> to unselect them.
- Right-click the mouse on a **Node** and prompt the user for a label to put on that node.
- Scale the entire graph up or down by holding the <SHIFT> key while pressing the mouse on an empty spot on the window and then dragging the mouse up or down to enlarge or shrink the graph.
- Press <CNTRL><D> to duplicate all selected nodes and edges and have the new portion of the graph appear a little below and to the right of the original nodes/edges.
- Show labels on edges
- Adjust labels so that they don't overlap