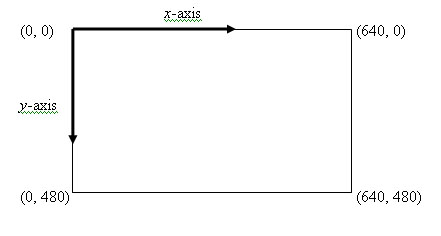
**Intro. to Graphics in Java**

**Window Organization**

The computer screen is made up of several **picture elements** or **pixels**. In turn, then, each window is made up of pixels.

Each pixel has a set of coordinates, which operate similarly to Cartesian coordinates. The key difference, though, is that the upper left corner is (0, 0) here (instead of the lower left).

Consider this illustration for a window that is 640 pixels wide by 480 pixels tall:



**Libraries**

In order to use graphics in Java programs, there are a number of libraries we need to import. For the sake of what will be covered in these notes, you need the following statements:

import javax.swing.JFrame;  
import java.awt.Graphics;  
import javax.swing.JPanel;  
import java.awt.Color;  
import javax.swing.JLabel;  
import javax.swing.ImageIcon;  
import java.awt.BorderLayout;

Once you've seen what's in these notes, it should be fairly obvious which statements do what. Of course, only import what you actually use in a program in practice.

**A Framework for Graphics Programs**

We'll use a framework for our programs where we'll launch a window where we'll do our drawing. This window will make use of some existing code via inheritance, thus our class must have JPanel as a superclass.

The key method in JPanel we must override for our programs is paintComponent(), which is void and takes in an object of the Graphics class. In fact, we will be passing a Graphics class object around anytime we do drawing. It's customary to call this object g. Note the superclass method call within within method.

We could opt to build a class that extends JPanel and have a separate main program. However, I'm opting to introduce this all tied into one program to focus on Graphics, so I'll include the main program in the framework for this introduction. We'll put the code there to set up and launch a window.

Finally, the constructor is optional here, but can be used to set the background color of the drawing window.

Here is a framework, with required code in color:

**public class** DrawPanel **extends JPanel**  
{

public DrawPanel() // set up graphics window  
 {  
 super();

setBackground(Color.WHITE);  
 }

**public void paintComponent(Graphics g)** // draw graphics in the panel **{**

int width = getWidth(); // width of window in pixels

int height = getHeight(); // height of window in pixels

**super.paintComponent(g);**  // call superclass to make panel display correctly

// Drawing code goes here

**}**

**public static void main(String[] args)**

**{**

**DrawPanel panel = new DrawPanel();**  // window for drawing

**JFrame application = new JFrame();**  // the program itself

**application.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);** // set frame to exit

// when it is closed

**application.add(panel);**

**application.setSize(**500, 400**);** // window is 500 pixels wide, 400 high

**application.setVisible(true);**

**}**

**}**

We'll now look at some of the drawing options you have while working with a Graphics object. As the drawing tools require a Graphics object, you'll need to use them in the paintComponent() method (or in a method called from the paintComponent() method with g passed to it).

***Note:*** These notes assume your Graphics object is called g.

**Text**

You can print text in a graphics context via:

**g.drawString(** *string* **,** *start x-coordinate* **,** *start y-coordinate* **);**

For example, the following code would print "Hello, World" at (100, 150):

g.drawString("Hello, World", 100, 150);

**Lines**

You can draw a line via a method call:

**g.drawLine(** *start x-coordinate* **,** *start y-coordinate* **,** *end x-coordinate* **,** *end y-coordinate* **);**

For example, the following code would draw a line from the top left of a window to (20, 40):

g.drawLine(0, 0, 20, 40);

**Problem:** Given the framework defined above, write a line of code to draw a line from the bottom left corner of the window to the upper right corner.

**Rectangles, Ovals, and Arcs**

You can draw a rectangle via a method call:

**g.drawRect(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**   
 *width* **,** *height* **);**

For example, the following code would draw a rectangle starting at (10, 10) that is 100 pixels high and 200 pixels wide:

g.drawRect(10, 10, 200, 100);

Ovals operate under a similar principle. They have a *bounding rectangle;* in other words, you send the same arguments to the oval method call as you would to the rectangle call, but you don't see that rectangle. Instead, the oval with fit inside the rectangle. Here's the general form:

**g.drawOval(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**   
 *width* **,** *height* **);**

Using the same example, the following code would draw an oval in the bounding rectangle we drew above:

g.drawOval(10, 10, 200, 100);

**Problem:** Write a line of code that draws a square with sides of length 20 at the center of the window.

 Arcs operate under a similar principle to ovals. There is once again a bounding rectangle. As the arc is part of an oval, we need to add additional arguments for what part of the oval we want to draw. These are the starting angle and ending angle in degrees. Imagine a set of axes centered at the center of the oval. Then 0 degrees would correspond to the positive *x*-axis. Here's a general form for drawing an arc:

**g.drawArc(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**   
 *width* **,** *height* **,** *start angle* **,** *angle swept out* **);**

**Problem:** Sketch on paper the arc drawn by this code:

g.drawArc(10, 10, 200, 100, 45, 90);

**Colors & The RGB Model**

Java defines a Color class; instances of this class represent various colors.

At the simplest level, we could pick one of 13 predefined colors from Java's virtual box of crayons:

* Color.BLACK
* Color.BLUE
* Color.CYAN
* Color.DARK\_GRAY
* Color.GRAY
* Color.GREEN
* Color.LIGHT\_GRAY
* Color.MAGENTA
* Color.ORANGE
* Color.PINK
* Color.RED
* Color.WHITE
* Color.YELLOW

**RGB**

Of course, there are many more colors we might want. We can specify other colors using the **RGB model**, which specifies a color with a red value, a green value, and a blue value (each called **channels**). For example, the red value is how much red we want in our color. In different graphics system, the individual color values might be represented in different ways, but in Java, we use 8 bits to each color value. This gives us 256 (28) choices for each of the R, G, and B values, and choices range from 0 to 255.

So, we'd specify a color as follows:

|  |  |  |
| --- | --- | --- |
| **Component** | **Minimum Value** | **Maximum Value** |
| Red Value | 0 (no red) | 255 (all red) |
| Green Value | 0 (no green) | 255 (all green) |
| Blue Value | 0 (no blue) | 255 (all blue) |

So, a color is represented as an ordered triple. (0, 0, 0) is the absence of color, or black. (255, 255, 255) means all colors are at their maximum value, which corresponds to white.

We can create Color objects using RGB values:

**Color** colorName **= new Color(** red value **,** green value **,** blue value **);**

So, the following code would create the color black (note that there's a constant though):

Color blackEx = new Color(0, 0, 0);

More interestingly, the following code would create a color that is a dark shade of blue:

Color darkBlue = new Color(0, 0, 102);

**Tangential Note:** 256 is also 162, which means a component of color can also be represented as 2 hexadecimal digits. Going with this, FF == 255. In other graphics systems, a triple is represented by 6 hex digits, where the first two are the red value, the second two are the green value, and the last two are the blue value. Thus, FF0000 is red, 00FF00 is green, and 0000FF is blue. We could combine colors: FF00FF is magenta . (Be careful, though, it's more involved than this.)

**Problem:** Recalling that 256 is 28, how many different colors does the RGB model allow us to choose from?

**Problem:** Define a Color object for magenta (disregarding the existing constant).

**Using Colors in Graphics Contexts**

We use the following line of code to change the color in which drawing will occur:

**g.setColor(** colorObject **);**

The sequential flow of programs is key here. This color setting persists until is changed. We could draw one object in this color or 20; but it stays until the next time we call setColor().

Consider the following example calls to setColor():

g.setColor(Color.ORANGE); // use Color constant

g.setColor(darkBlue); // use the Color object defined above

g.setColor(new Color(0, 102, 0)); // create a Color object on the fly

**Filled Shapes**

Now that we know about colors, we may also want to draw shapes that are filled in, instead of just the outlines of shapes we drew above. Drawing filled shapes involves the same arguments as above, but the method name is slightly different. Here are the general forms:

**g.fillRect(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**   
 *width* **,** *height* **);**

**g.fillOval(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**   
 *width* **,** *height* **);**

**g.fillArc(** *x-coordinate of top left corner* **,** *y-coordinate of top left corner* **,**

*width* **,** *height* **,** *start angle* **,** *angle swept out* **);**

**Example:** Consider the following example of using colors and filled shapes:

g.setColor(Color.RED);  
g.fillRect(10, 10, 200, 200);  
g.setColor(new Color(0, 0, 102));  
g.fillOval(20, 20, 180, 180);

**A Few Window Modifications**

Let's look at two settings we can change *in the main function* to change our window. This assumes the framework above, where application is the JFrame object.

We can change the name of the window:

**application.setTitle("**Name of your window**")**

We can change the background color:

**application.setBackground(** colorObject **);**

Note that the latter sets the JFrame's background color, but in our framework above, we have a JPanel sitting on top of the JFrame, so you'd need to use the setBackground of the constructor in the framework above to change that color if desired.

**Basic drawing**

In this part of the Java 2D tutorial, we will do some basic drawing.

**Points**

The most simple graphics primitive is a point. It is a single dot on the window. There is a Point class for representing a point in a coordinate space, but there is no method to to draw a point. We used the drawLine() method, where we supplied one point twice.

Points.java

package com.zetcode;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Insets;

import java.util.Random;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.SwingUtilities;

class Surface extends JPanel {

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.setColor(Color.blue);

Dimension size = getSize();

Insets insets = getInsets();

int w = size.width - insets.left - insets.right;

int h = size.height - insets.top - insets.bottom;

Random r = new Random();

for (int i = 0; i < 1000; i++) {

int x = Math.abs(r.nextInt()) % w;

int y = Math.abs(r.nextInt()) % h;

g2d.drawLine(x, y, x, y);

}

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

}

public class Points extends JFrame {

public Points() {

initUI();

}

private void initUI() {

setTitle("Points");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

add(new Surface());

setSize(350, 250);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

Points ps = new Points();

ps.setVisible(true);

}

});

}

}

One point is difficult to observe. Therefore, we will randomly draw 1000 points on the panel surface.

g2d.setColor(Color.blue);

We will paint our points in blue color.

Dimension size = getSize();

Insets insets = getInsets();

The size of the window includes borders and a titlebar. We don't paint there.

int w = size.width - insets.left - insets.right;

int h = size.height - insets.top - insets.bottom;

Here we calculate the area where we will effectively paint our points.

Random r = new Random();

int x = Math.abs(r.nextInt()) % w;

int y = Math.abs(r.nextInt()) % h;

We get a random number in range of the size of area that we computed above.

g2d.drawLine(x, y, x, y);

Here we draw the point. As we already mentioned, we use the drawLine() method. We specify the same point twice.

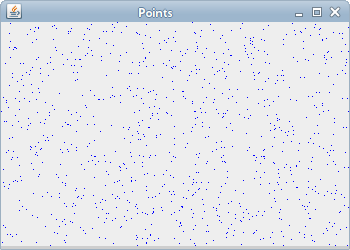


Figure: Points

**Lines**

A line is a simple graphics primitive. A line is an object which connects two points. Lines are drawn with the drawLine() method.

Lines2.java

package com.zetcode;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.SwingUtilities;

class Surface extends JPanel {

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

g2d.drawLine(30, 30, 200, 30);

g2d.drawLine(200, 30, 30, 200);

g2d.drawLine(30, 200, 200, 200);

g2d.drawLine(200, 200, 30, 30);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

}

public class Lines2 extends JFrame {

public Lines2() {

initUI();

}

private void initUI() {

setTitle("Lines");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

add(new Surface());

setSize(350, 250);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

Lines2 lines = new Lines2();

lines.setVisible(true);

}

});

}

}

We draw a simple object with four lines.

g2d.drawLine(30, 30, 200, 30);

A straight line is drawn. The parameters of the method are the x, y coordinates of the two points.

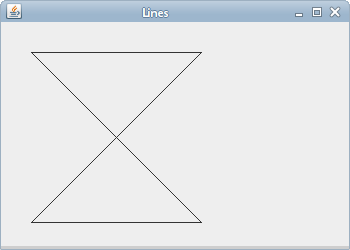


Figure: Lines

**BasicStroke**

The BasicStroke class defines a basic set of rendering attributes for the outlines of graphics primitives. These rendering attributes include width, end caps, line joins, miter limit, and dash attributes.

BasicStrokes.java

package com.zetcode;

import java.awt.BasicStroke;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.SwingUtilities;

class Surface extends JPanel {

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

float[] dash1 = {2f, 0f, 2f};

float[] dash2 = {1f, 1f, 1f};

float[] dash3 = {4f, 0f, 2f};

float[] dash4 = {4f, 4f, 1f};

g2d.drawLine(20, 40, 250, 40);

BasicStroke bs1 = new BasicStroke(1, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_ROUND, 1.0f, dash1, 2f);

BasicStroke bs2 = new BasicStroke(1, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_ROUND, 1.0f, dash2, 2f);

BasicStroke bs3 = new BasicStroke(1, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_ROUND, 1.0f, dash3, 2f);

BasicStroke bs4 = new BasicStroke(1, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_ROUND, 1.0f, dash4, 2f);

g2d.setStroke(bs1);

g2d.drawLine(20, 80, 250, 80);

g2d.setStroke(bs2);

g2d.drawLine(20, 120, 250, 120);

g2d.setStroke(bs3);

g2d.drawLine(20, 160, 250, 160);

g2d.setStroke(bs4);

g2d.drawLine(20, 200, 250, 200);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

}

public class BasicStrokes extends JFrame {

public BasicStrokes() {

initUI();

}

private void initUI() {

setTitle("Basic strokes");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

add(new Surface());

setSize(280, 270);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

BasicStrokes bs = new BasicStrokes();

bs.setVisible(true);

}

});

}

}

In this example, we show various types of dashes. A dash attribute is a pattern, which is created by mixing opaque and transparent sections.

float[] dash1 = { 2f, 0f, 2f };

float[] dash2 = { 1f, 1f, 1f };

float[] dash3 = { 4f, 0f, 2f };

float[] dash4 = { 4f, 4f, 1f };

Here we define four different dash patterns.

BasicStroke bs1 = new BasicStroke(1, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_ROUND, 1.0f, dash1, 2f );

This line constructs a BasicStroke object.

g2d.setStroke(bs1);

We use the setStroke() method to apply the BasicStroke to the current graphics context.

g2d.drawLine(20, 80, 250, 80);

Finally, we draw the line.

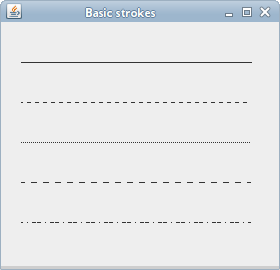


Figure: Basic strokes

**Caps**

Caps are decorations applied to the ends of unclosed subpaths and dash segments. There are three different end caps in Java 2D: CAP\_BUTT, CAP\_ROUND and CAP\_SQUARE.

Caps.java

package com.zetcode;

import java.awt.BasicStroke;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.RenderingHints;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.SwingUtilities;

class Surface extends JPanel {

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

RenderingHints rh = new RenderingHints(

RenderingHints.KEY\_ANTIALIASING,

RenderingHints.VALUE\_ANTIALIAS\_ON);

rh.put(RenderingHints.KEY\_RENDERING,

RenderingHints.VALUE\_RENDER\_QUALITY);

g2d.setRenderingHints(rh);

BasicStroke bs1 = new BasicStroke(8, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs1);

g2d.drawLine(20, 30, 250, 30);

BasicStroke bs2 = new BasicStroke(8, BasicStroke.CAP\_ROUND,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs2);

g2d.drawLine(20, 80, 250, 80);

BasicStroke bs3 = new BasicStroke(8, BasicStroke.CAP\_SQUARE,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs3);

g2d.drawLine(20, 130, 250, 130);

BasicStroke bs4 = new BasicStroke();

g2d.setStroke(bs4);

g2d.drawLine(20, 20, 20, 140);

g2d.drawLine(250, 20, 250, 140);

g2d.drawLine(254, 20, 254, 140);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

}

public class Caps extends JFrame {

public Caps() {

initUI();

}

private void initUI() {

setTitle("Caps");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

add(new Surface());

setSize(280, 270);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

Caps caps = new Caps();

caps.setVisible(true);

}

});

}

}

In our example, we show all three types of end caps.

BasicStroke bs1 = new BasicStroke(8, BasicStroke.CAP\_BUTT,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs1);

A basic stroke with a butt cap is created and applied.

g2d.drawLine(20, 20, 20, 140);

g2d.drawLine(250, 20, 250, 140);

g2d.drawLine(254, 20, 254, 140);

We draw three vertical lines to explain the differences among the end caps. Lines with CAP\_ROUND and CAP\_SQUARE are bigger than the line with CAP\_BUTT. Exactly how much bigger depends on the line size. In our case a line is 8px thick. Lines are bigger by 8px, 4px on the left and 4px on the right. It is clear from the picture.

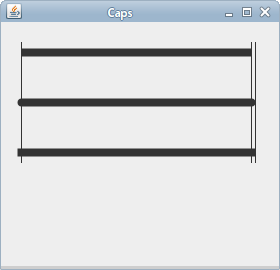


Figure: Caps

**Joins**

Line joins are decorations applied at the intersection of two path segments and at the intersection of the endpoints of a subpath. There are three decorations are. JOIN\_BEVEL, JOIN\_MITER and JOIN\_ROUND.

Joins.java

package com.zetcode;

import java.awt.BasicStroke;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.SwingUtilities;

class Surface extends JPanel {

private void doDrawing(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

BasicStroke bs1 = new BasicStroke(8, BasicStroke.CAP\_ROUND,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs1);

g2d.drawRect(15, 15, 80, 50);

BasicStroke bs2 = new BasicStroke(8, BasicStroke.CAP\_ROUND,

BasicStroke.JOIN\_MITER);

g2d.setStroke(bs2);

g2d.drawRect(125, 15, 80, 50);

BasicStroke bs3 = new BasicStroke(8, BasicStroke.CAP\_ROUND,

BasicStroke.JOIN\_ROUND);

g2d.setStroke(bs3);

g2d.drawRect(235, 15, 80, 50);

}

@Override

public void paintComponent(Graphics g) {

super.paintComponent(g);

doDrawing(g);

}

}

public class Joins extends JFrame {

public Joins() {

initUI();

}

private void initUI() {

setTitle("Joins");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

add(new Surface());

setSize(340, 110);

setLocationRelativeTo(null);

}

public static void main(String[] args) {

SwingUtilities.invokeLater(new Runnable() {

@Override

public void run() {

Joins js = new Joins();

js.setVisible(true);

}

});

}

}

This code example show three different line joins in action.

BasicStroke bs1 = new BasicStroke(8, BasicStroke.CAP\_ROUND,

BasicStroke.JOIN\_BEVEL);

g2d.setStroke(bs1);

g2d.drawRect(15, 15, 80, 50);

Here we create a rectangle with a JOIN\_BEVEL join.

