Graphics

Graphics

We are going to have a look at one of Java's graphics libraries

And how to write programs with a Graphical User Interface

And along the way we'll have a quick look at inner classes, and lambda expressions, and concurrency

Graphical User Interfaces

In Java, graphics are part of the ***standard*** libraries and (almost) ***platform independent***

A graphical user interface (GUI) is difficult in any language, partly because of the number of issues:

- *Drawing:* lines, shapes, icons, ...
- *Widgets:* buttons, menus, sliders, ...
- *Events:* keys, mouse, interactions, timings, ...
- *Media:* photos, videos, sound, ...
- *Concurrency:* threads, doing several things at once

Applications versus Applets

An application is a complete Java program; an applet is a subprogram executed by a surrounding program

Applets run in browsers on the Web, but are obsolete

Warning: many books and tutorials start with applets

We are only interested in applications

Graphics libraries

Excluding Android and other non-standard libraries, there are three standard graphics libraries in Java:

- AWT (Abstract Window Toolkit, oldest)
- Swing
- JavaFX (newest)

We will concentrate on ***JavaFX***

JavaFX is included in the official JDK, but not yet in OpenJDK, at the time of writing, unless you build it yourself

Info

Documentation: Java client technologies (left column)

API: JavaFX API (743 classes, but well organised)

Beware obsolete tutorials, from before JavaFX was reworked, standardised, and included in the libraries

Hello World

Hi.java

```
/* Say hello to the world... */

import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.control.*;

public class Hi extends Application {
  public void start(Stage stage) {
    Label hello = new Label("Hello JavaFX world!");
    Group root = new Group(hello);
    Scene scene = new Scene(root);
    stage.setTitle("Hello World!");
    stage.setScene(scene);
    stage.show();
  }
}
```

Imports

Hi.java

```
/* Say hello to the world... */

import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.control.*;
...
```

You typically need a ***lot*** of imports to use JavaFX - check the API documentation for each class you use

Application

Hi.java

```
...
public class Hi extends Application {
  // No main method !!!
}
```

The program has no `main` method (if you add it, it just calls `launch`)

Extending `Application` is enough for the launcher to work out what to do

Your class must be `public` for the launcher to find it

Threading

The fact that there is no `main` is a signal that you are not in control - the graphics system is

There is a graphics thread, not your `main` thread, that controls everything and calls your methods

Accept it and don't try to add your own threads

Don't mix JavaFX with Swing or AWT (possible, but very messy)

Start

Hi.java

```
...
  @Override
  public void start(Stage stage) {
     ...
  }
...
```

You provide a `start` method, which overrides the method in the `Application` class

You must include `public` and you can add the `@Override` annotation if you like

Stage

Hi.java

```
...
    stage.setTitle("Hello World!");
    stage.setScene(scene);
    stage.show();
...
```

A `Stage` is a window belonging to the platform (called a Frame or JFrame in the other libraries)

The `setTitle` call puts text in the title bar

A `Stage` displays a `Scene`, and appears when `show` is called

Scene graph

Hi.java

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

It is common in graphics libraries, especially 3D ones, to use a scene graph, which is (usually) a tree of nodes

A node is a displayable object (shape, canvas, button...)

Nodes extend the `Node` class

Some extend the `Parent` class, have subnodes, and can be thought of as containers or layout managers

Groups

Hi.java

```
...
    Group root = new Group(hello);
...
```

A `Scene` has to be given a `Parent`, so it knows how to layout the rest of the nodes

A `Group` is the simplest parent, which does not resize its children, like a viewport showing a restricted area

You can give its subnodes as arguments, or add them later with `root.getChildren().add(hello)`

Labels

Hi.java

```
...
    Label hello = new Label("Hello JavaFX world!");
...
```

A label is just a piece of text; it is like a button you can't press, or a text area you can't edit

It is a `Control` which is a self-contained, skinnable, interactive `Node` (called a 'widget' in other systems)

The Drawing program

Drawing.java

```
...
  public void start(Stage stage) {
    Canvas canvas = new Canvas(400, 300);
    Group root = new Group(canvas);
    stage.setScene(new Scene(root));
    GraphicsContext g =
      canvas.getGraphicsContext2D();
    draw(g);
    stage.show();
  }
...
```

A `Canvas` is a flat area for drawing on using a `GraphicsContext` (like a pen or brush)

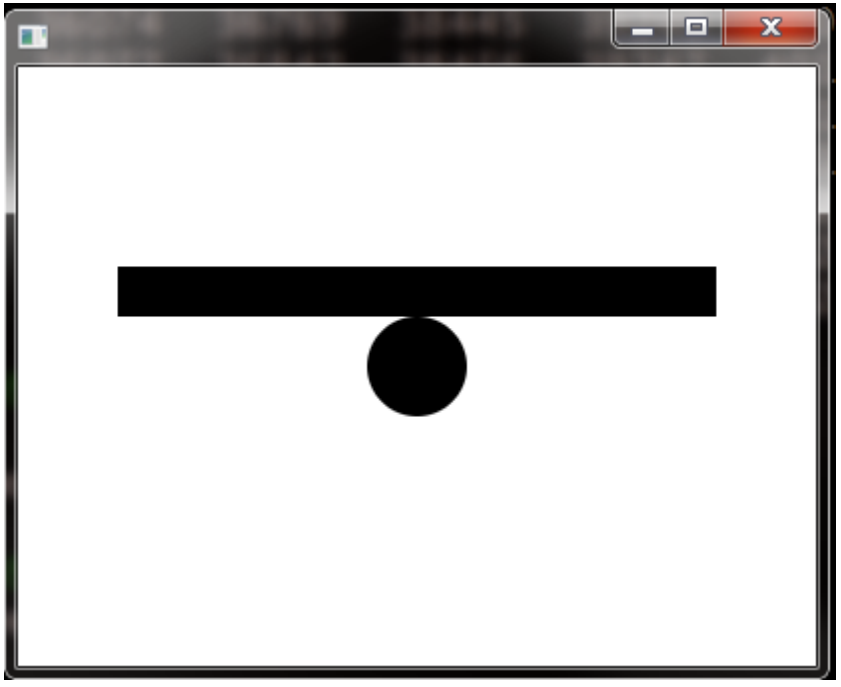
The draw method

Drawing.java

```
...
  private void draw(GraphicsContext g) {
    g.fillRect(50, 100, 300, 25);
    g.fillOval(175, 125, 50, 50);
  }
...
```

The `draw` method, which we provide, draws a rectangle and a circle to form a seesaw

The system remembers how we did the drawing, and automatically re-draws when necessary (e.g. on uncover or maximize)



Images

Let's look at two different ways to use images

One is to create an object and put it into the scene as an independent node

The other is to paint it onto a canvas as a background, so you can then draw other things on top

The Photo program

Photo.java

```
...
  public void start(Stage stage) {
    Image photo = new Image("photo.png");
    ImageView view = new ImageView(photo);
    Group root = new Group(view);
    stage.setScene(new Scene(root));
    stage.show();
  }
...
```

This loads the image from a file (be careful to do this only once!) and puts it into an `ImageView` node

Loading an image


```
1)   Image i = new Image("x.png");
2)   Image i = new Image("/x.png");
3)   Image i = new Image("p/images/x.png");
4)   Image i = new Image("http://www.../x.png");
5)   Image i = new Image("file:x.png");
```

1) loads from the classpath, so it stays OK if you move the project or zip it up into a jar file

2) specifies the 'default package', 3) has a package prefix and a sub-folder, 4) is a URL (won't work offline)

5) loads from a file on your computer (not portable)

The Playground program

Playground.java

```
...
  public void start(Stage stage) {
    Image bg = new Image("background.png");
    Canvas canvas = new Canvas(400, 300);
    Group root = new Group(canvas);
    stage.setScene(new Scene(root));
    GraphicsContext g =
      canvas.getGraphicsContext2D();
    draw(g, bg);
    stage.show();
  }
...
```

This loads the image and passes it to `draw`

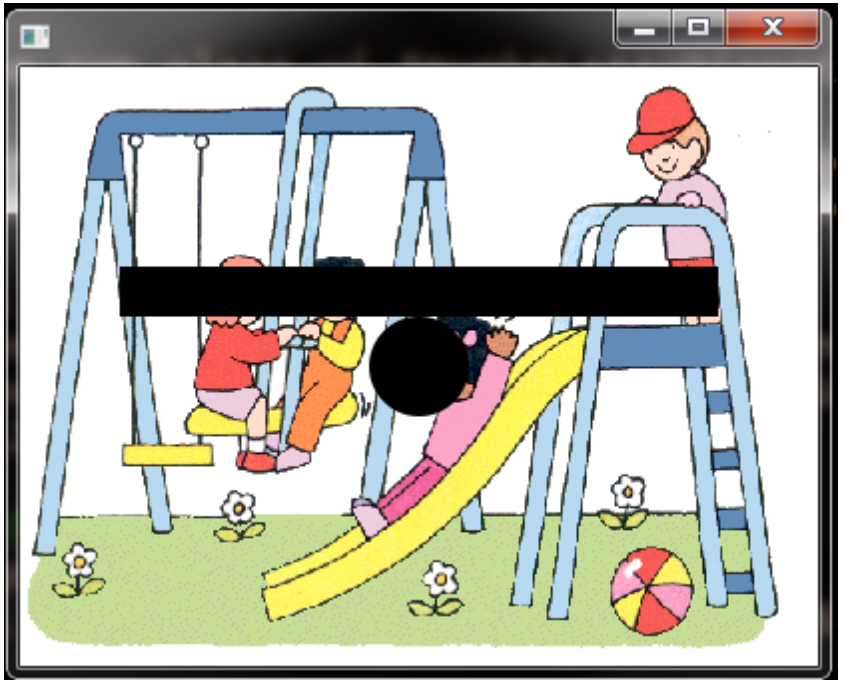
Painting an image

Playground.java

```
...
private void draw(GraphicsContext g, Image b) {
   g.drawImage(bg, 0, 0);
   g.fillRect(50, 100, 300, 25);
   g.fillOval(175, 125, 50, 50);
}
...
```

The `drawImage` method paints the image as a background, before drawing on top

The image is 300x300, the same as the canvas



Events

Input comes into a program via the graphics system in the form of ***events*** such as key presses, mouse clicks, mouse movements

You don't "ask for the next event", the events themselves drive the program

Each event causes a method to be called (a callback), which you write and "register" with the event system

The Press program

Press.java

```
...
  public void start(Stage stage) {
    Button b = new Button("Press me");
    b.setOnAction(this::press);
    Group root = new Group(b);
    stage.setTitle("Press");
    stage.setScene(new Scene(root));
    stage.show();
  }
...
```

The `setOnAction` call attaches the `press` method to the button

Then `press` is called when the button is pressed

The press method

Press.java

```
...
  private void press(ActionEvent e) {
    System.out.println("Button pressed");
  }
...
```

The system passes an `ActionEvent` object to your method, with various info in it

Method passing


```
b.setOnAction(this::press);
```

The argument `this::press` means "the method `press` attached to the object `this` as its closure"

The `::` notation is used to pass a method, and the argument type must be a one-method interface

The method must match the signature in the interface, but doesn't need to be public, or have the same name, or to be declared as implementing the interface

Callbacks


```
b.setOnAction(this::press);
```

The method being passed is a *callback*

The graphics thread is not under your control – it calls methods that you provide

There are ***no*** calls to `press` in your own code – the system decides when to call it (when the button is pressed)

The callback rule

A graphical interface must be responsive at all times

So, a callback function must ***never*** take a long time

It must not call methods like `sleep` or `wait` or do much I/O or networking, or do any long calculations

To get round these restrictions, without threading problems, use something like `Timeline` or `Platform.runLater` or `Worker` and `Task` from `javax.concurrent.*`

A bad alternative


```
btn.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent event) {
    System.out.println("Hello World!");
  }
});
```

You see this bad style in lots of documentation

It creates an object of an anonymous class which overrides the `handle` method

This is ugly, and unnecessary in Java 8

Why bad

The notation has a vertical thing (class body) inside a horizontal thing (method call)

And one statement is doing too much, with a mess of brackets (long statements ought to be broken up)

And it has a method inside another method (methods ought to be short and separate)

And it creates an extra class file

And there is something so much better

An ugly alternative


```
btn.setOnAction(new Handler());
...
private static class Handler
implements EventHandler<ActionEvent>() {
  public void handle(ActionEvent event) {
    System.out.println("Hello World!");
  }
}
```

This is not quite so bad, because it is not so heavily nested, and it is reasonable in Java 7 or earlier

The inner class is no longer anonymous – it needn't be static if needs access to fields in the outer class

Inner classes

An *inner class* is one class defined inside another class

As long as you don't use a bad notational style, inner classes can sometimes be very useful

A `static` inner class is a 'normal' class, e.g. `class Y` inside `class X` produced the class `X.Y` with class file `X$Y.class`

An object of a non-static inner class has a hidden pointer to its outer object in which it was created, and so has implicit access to the fields of the outer object

Unusual notation

Suppose class X has an inner class Y which is not static or private

From the outside, is it possible to create an object x of class X, and an object y of class Y with x as its parent? Yes:

```
X x = new X();
X.Y y = x.new Y();
```

A constructor can be called as a method on an object, even though it includes the new keyword

The Click program

Click.java

```
...
public class Click extends Application {
  private Label counter;
  private int n = 0;
...
```

The program keeps a count `n` of clicks, and displays it in a `counter` label

These two variables need to survive between method calls, so they are defined as fields

The start method

Click.java

```
...
  public void start(Stage stage) {
    counter = new Label("0");
    Group root = new Group(counter);
    Scene scene = new Scene(root);
    scene.setOnMousePressed(this::click);
    stage.setTitle("Click");
    stage.setScene(scene);
    stage.show();
  }
...
```

The `setOnMousePressed` call sets up the `click` method as a callback

The click method

Click.java

```
...
  private void click(MouseEvent event) {
    n++;
    counter.setText("" + n);
  }
...
```

The system detects that the `counter` object has changed and needs to be redisplayed

Mouse clicks

Don't use `setOnMouseClicked`

A 'mouse clicked' event is, technically, a mouse press then a mouse release ***with no movement in between***

In a fast-paced game or similar, the mouse has often moved a bit between the press and release, so it will seem as if the program is missing some clicks

So it is usually better to detect mouse press and/or release events

The Counter program

Counter.java

```
...
public class Counter extends Application {
  private IntegerProperty n;
...
```

This is the same as the `Click` program, but using a binding technique where a graphical object automatically tracks changes in a non-graphical object

An `IntegerProperty` is like an `int`, except that changes to it can be tracked

The start method

Counter.java

```
...
  public void start(Stage stage) {
    n = new SimpleIntegerProperty(0);
    Label counter = new Label();
    counter.textProperty().bind(n.asString());
    Group root = new Group(counter);
    Scene scene = new Scene(root);
    scene.setOnMousePressed(this::click);
    stage.setTitle("Counter");
    stage.setScene(scene);
    stage.show();
  }
...
```

The call `...bind(n.asString())` makes the text in the label track the text value of the counter

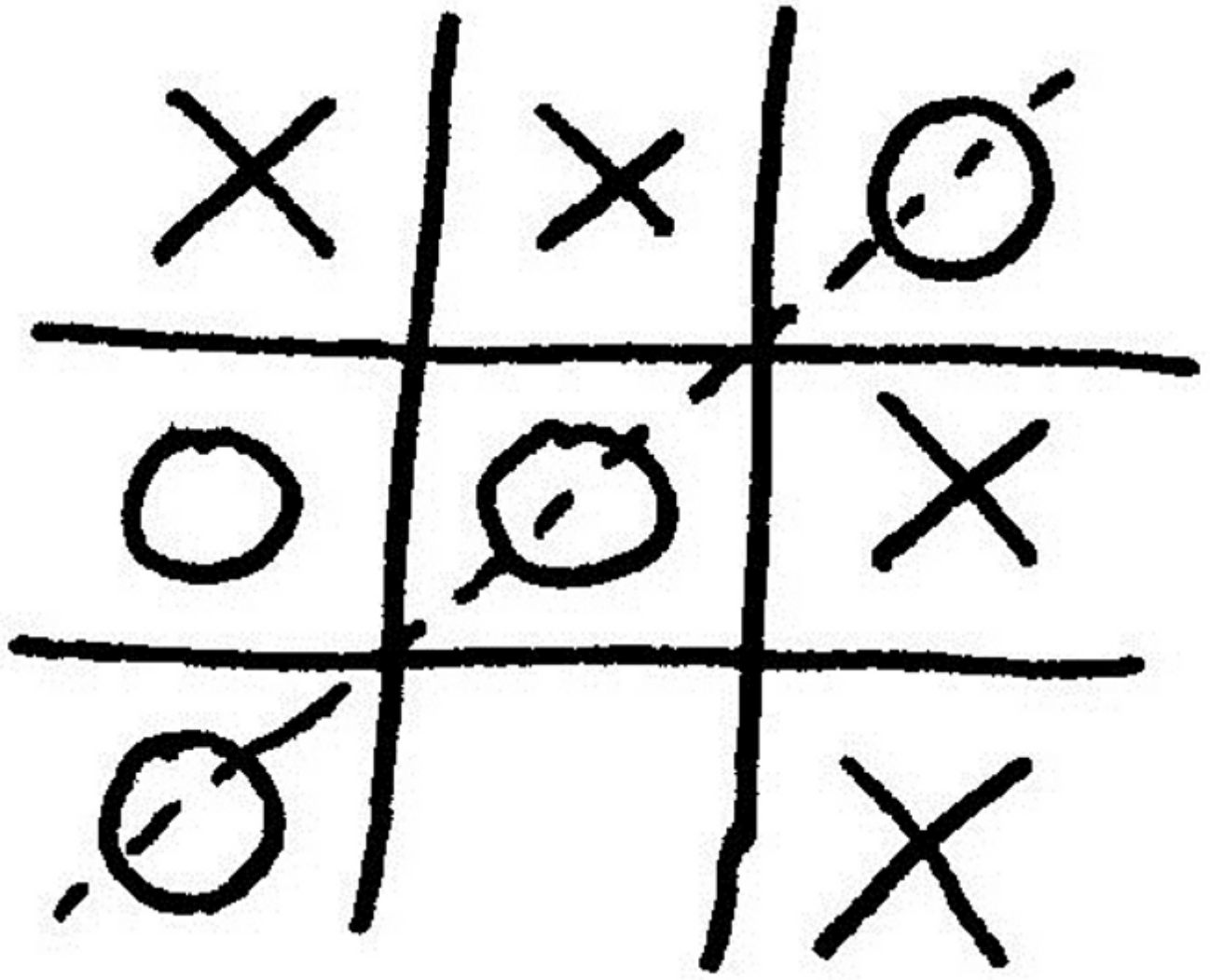
The click method

Counter.java

```
...
  private void click(MouseEvent event) {
    n.set(n.get()+1);
  }
...
```

Now the label doesn't have to be updated separately, just the integer

There isn't much gain in this program, but you can see that it could be a powerful design aid in some programs



Noughts and crosses

As a slightly larger example, let's put together a graphical interface for noughts and crosses

The program is complex enough to need splitting into two classes

The `Cross` class provides the user interface, and the `Grid` class separates out the logic

`Cross` depends on `Grid` but not vice versa

That means `Grid` can be developed separately, and tested

The Grid class

Grid.java

```
class Grid {
    private char X='X', O='O', S=' ';
    private char[][] cells = {{S,S,S},{S,S,S},{S,S,S}};
    private char whoseTurn = X;

    char get(int x, int y) { return cells[x][y]; }

    void move(int x, int y) {
        cells[x][y] = whoseTurn;
        whoseTurn = (whoseTurn == X) ? O : X;
    }
}
```

There is no validation, no win/draw checking, no testing

You can see that there is no dependency on graphics, and that it would be easy to add those things

The Cross program

Cross.java

```
...
public class Cross extends Application {
  private Grid grid;
  private GraphicsContext g;
...
```

The `Cross` class is the main program and the GUI

The grid, and the graphics context for drawing on the canvas, are made fields so they are accessible in every method call

The start method


```
...public void start(Stage stage) {
      grid = new Grid();
      Canvas canvas = new Canvas(300, 300);
      Group root = new Group(canvas);
      Scene scene = new Scene(root);
      scene.setOnMousePressed(this::move);
      stage.setTitle("Cross");
      stage.setScene(scene);
      g = canvas.getGraphicsContext2D();
      draw();
      stage.show();
...}
```

Cross.java

The grid and canvas are initialised, a move method is set up as a callback for mouse clicks, and a draw method is called to draw the initial grid

The move method

Cross.java

```
...
  private void move(MouseEvent e) {
    int x = (int) e.getSceneX() / 100;
    int y = (int) e.getSceneY() / 100;
    grid.move(x, y);
    draw();
  }
...
```

This gets the mouse position from the event object, converts the floats to integer pixel values, divides by 100 to get grid coordinates from 0 to 2, makes a move in the grid, and redraws the grid onto the canvas

The draw method

Cross.java

```
...
  private void draw() {
    g.clearRect(0, 0, 300, 300);
    g.setLineWidth(1);
    drawLines();
    g.setLineWidth(3);
    for (int x=0; x<3; x++) {
      for (int y=0; y<3; y++) {
        char c = grid.get(x,y);
        if (c == 'O') drawO(100*x, 100*y);
        else if (c == 'X') drawX(100*x, 100*y);
      }
    }
  }
...
```

This redraws the current state of the grid from scratch

The drawLines method

Cross.java

```
...
  private void drawLines() {
    g.strokeLine(100.5, 10, 100.5, 290);
    g.strokeLine(200.5, 10, 200.5, 290);
    g.strokeLine(10, 100.5, 290, 100.5);
    g.strokeLine(10, 200.5, 290, 200.5);
  }
...
```

This draws the four one-pixel-width grid lines

To draw a crisp orthogonal line with an odd pixel-width, draw it at a half-pixel position

An alternative


```
g.strokeLine(100.5, 10, 100.5, 290);
```

```
g.beginPath();
g.moveTo(100.5, 10);
g.lineTo(100.5, 290);
g.closePath();
g.stroke();
```

Instead of `strokeLine`, you can use a path

The `beginPath`, `closePath`, `stroke` calls can be shared between all four lines

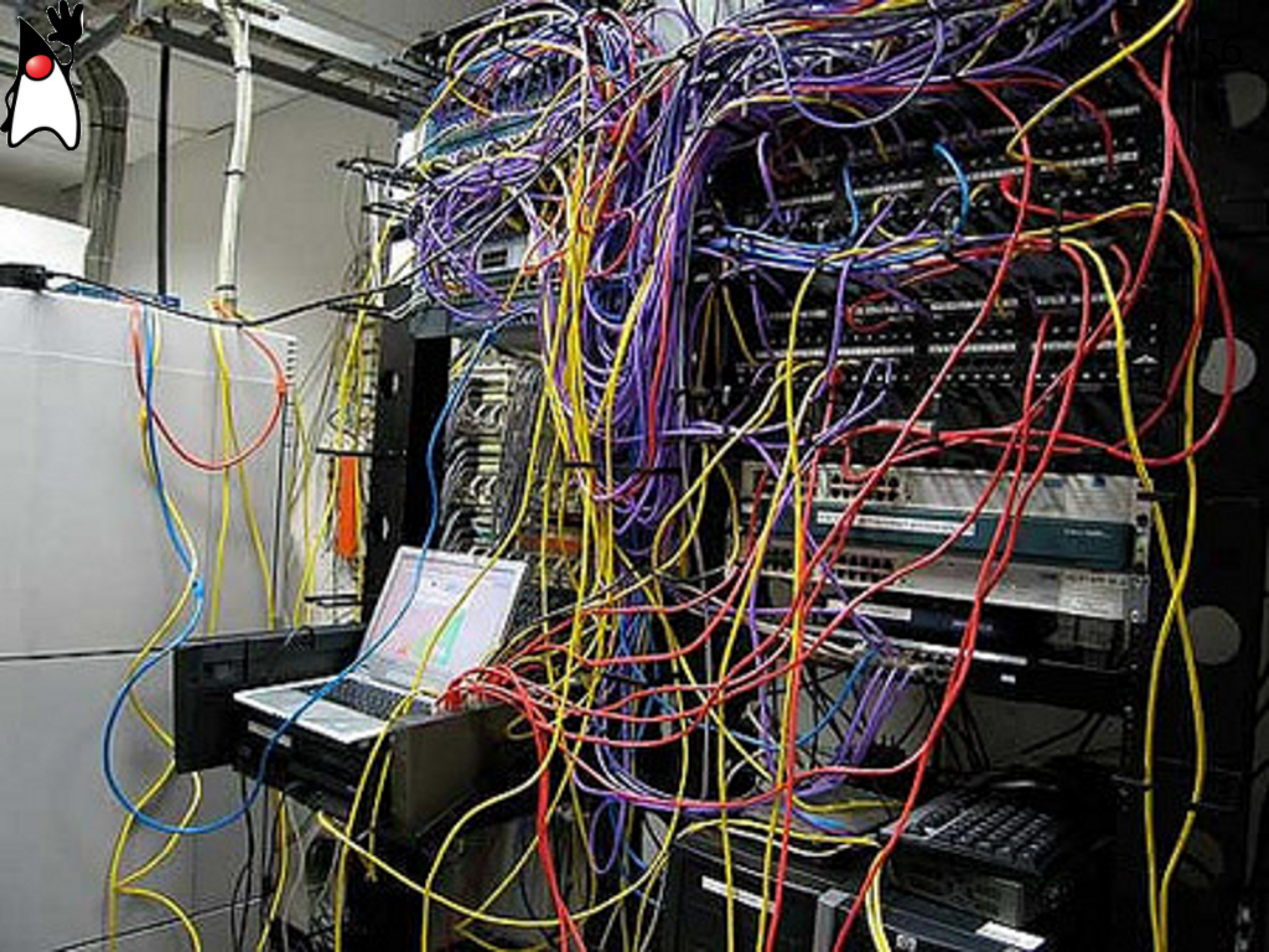
The drawO, drawX methods

Cross.java

```
...
  private void drawO(double x, double y) {
    g.strokeOval(12+x, 12+y, 75, 75);
  }

  private void drawX(double x, double y) {
    g.strokeLine(12+x, 12+y, 88+x, 88+y);
    g.strokeLine(12+x, 88+y, 88+x, 12+y);
  }
```

The O is drawn as a circle, and the X as two lines



Wiring up

Suppose we have a calculator with ten buttons for digits

What is the best design?

To create the buttons, we could use ten lines:

```
Button b0 = new Button("0");
Button b1 = new Button("1");
...
```

That's not very DRY, so can we do better?

Array

We can have an array of buttons:

```
Button[] bs = new Button[10];
for (int i=0; i<10; i++) bs[i] = new Button("" + i);
```

That has reduced ten lines to two

Order

Now we want to add them to the display, e.g. in a 3-column `TilePane`

The problem is, for a conventional calculator, we want to add them in the order 7,8,9, 4,5,6, 1,2,3, 0

We could do it in ten lines:

```
...add(bs[7]);
...add(bs[8]);
...add(bs[9]);
...
```

That's not very DRY, so can we do better?

Formula

Maybe we can find a formula so that we can write:

```
for (int i=0; i<10; i++) {
  ...add(bs[formula_in_i]);
}
```

The formula should map 0 to 7, 1 to 8 and so on

Can we find one? Yes, roughly `(2-i/3)*3+i%3+1`

Can we write it on one line? Yes

Should we use it? No

It is messy and unreadable (cryptic or 'magic')

Array

A better way is to use an array to define the order::

```
int[] order = {7,8,9, 4,5,6, 1,2,3, 0};
for (int i=0; i<10; i++) {
  pane.getChildren().add(bs[order[i]]);
}
```

That's not bad

Handlers

The next problem is handling button presses

One approach is to attach each button to a separate handler method:

```
b0.setOnAction(this::click0);
b1.setOnAction(this::click1);
...
```

That's not very DRY, because each method does nearly the same thing, so can we do better?

Single handler

Another possibility is to attach all the buttons to a single handler, and check which button has been pressed:

```
for ... bs[i].setOnAction(this::click);
...
private void click(ActionEvent e) {
  Object b = e.getSource();
  if (b == bs[0]) ...
  else if (b == bs[1]) ...
  else ...
}
```

That's not very DRY, so can we do better?

A switch would be more efficient, but still not very DRY

Button labels

What we can do is look at the text label on the button:

```
for ... bs[i].setOnAction(this::click);
...
private void click(ActionEvent e) {
  Button b = (Button) e.getSource();
  String s = b.getText();
  System.out.println("Button " + s + " pressed");
}
```

A cast is needed because the result from `getSource` is an `Object`, but we know it is a `Button`

The resulting program is Buttons.java

Design

The moral of this example is: use all your normal design skills to make your graphics code neat, compact, readable and non-repetitive

This is all the more important because a graphics class cannot normally be directly unit-tested

Too many programmers, including most tutorial writers, seem to forget the usual skills and techniques when writing graphics programs

The Seesaw program

Seesaw.java

```
...
public class Seesaw extends Application {
  private GraphicsContext g;
  private double angle;
...
```

Let's implement an animated seesaw, using a canvas

We need to keep track of the canvas to draw on it, and the angle of the seesaw to work out how to draw it

The start method

Seesaw.java

```
...
  public void start(Stage stage) {
    Canvas canvas = new Canvas(400, 300);
    stage.setScene(new Scene(new Group(canvas)));
    g = canvas.getGraphicsContext2D();
    stage.show();
    timer.start();
  }
...
```

We are going to define a timer, which is going to draw on the canvas, once per frame

The timer

Seesaw.java

```
...
  AnimationTimer timer = new AnimationTimer() {
    public void handle(long now) {
      updateAngle(now);
      g.save();
      g.clearRect(0, 0, 400, 300);
      g.translate(200, 150);
      g.rotate(angle);
      g.fillRect(-150, -50, 300, 25);
      g.fillOval(-25, -25, 50, 50);
      g.restore();
    }
  };        // note semicolon to end assignment
...
```

We are going to define a timer, which is going to draw on the canvas, once per frame

The updateAngle method

Seesaw.java

```
...
private void updateAngle(long now) {
  long bn = 1000000000;
  long seconds = now / bn;
  double fraction = (double) (now % bn) / bn;
  if ((seconds & 0x01) != 0)
    fraction = 1 - fraction;
  angle = -23 + fraction * 46;
}
```

This calculates the angle from the current time, in nanoseconds; in the `fraction` line, the cast ensures the division isn't an integer one

The angle increases/decreases in even/odd seconds

Detail: inner class


```
...
  AnimationTimer timer = new AnimationTimer() {
    public void handle(long now) {
      ...
    }
  };      // note semicolon to end assignment
...
```

This creates an anonymous class extending `AnimationTimer` and overriding `handle`, and creates an object of that class, in one go

(It would be nicer if you could pass a method in a call, as with other callbacks, using `::`, but you can't)

Detail: frames


```
...
    public void handle(long now) {
       ...
    }
...
```

The animation `handle` method is called once per frame

The time between frames may vary, even in one program run

So, the time (in nanoseconds) is passed in, and everything is calculated from that

Detail: transformations

Seesaw.java

```
...
      g.save();
      ...translate...
      ...rotate...
      g.restore();
...
```

Transformations affect the whole coordinate system

The `save` and `restore` calls make sure that the transformation is reset on each call, instead of accumulating

Detail: rotation

Seesaw.java

```
...
        g.translate(200, 150);
        g.rotate(angle);
        g.fillRect(-150, -50, 300, 25);
        g.fillOval(-25, -25, 50, 50);
...
```

There is no method to rotate about a particular point, only about the origin

So, we move the origin to the centre of the canvas, and draw the picture with the circle centred on the origin

Objects and vectors

Canvases are a recent addition to JavaFX, for compatibility with other graphics libraries (especially HTML5+JS)

They are bitmap based, whereas JavaFX is mainly object and vector based

The advantages of the object/vector approach are (a) scalability (b) GPU efficiency

So let's redo the seesaw in 'standard' JavaFX style

Design

With the canvas approach, we saw two stages

1. calculate variables from the current time
2. use the variables to draw something

Animation classes are provided to help with 1

By having self-drawable objects, 2 is fully automated

The simplest case is a `Transition` to calculate a single variable which ***is*** a property of an object

The Teeter program

Teeter.java

```
...
public class Teeter extends Application {
  public void start(Stage stage) {
    Circle hub = new Circle(200, 150, 25);
    Rectangle board = new Rectangle(50, 100, 300, 25);
    Group root = new Group(hub, board);
    Scene scene = new Scene(root, 400, 300);
    stage.setScene(scene);
    stage.show();
    animate(board);
  }
...
```

The seesaw is defined as circle and rectangle objects

Only the rectangle needs to be rotated

The animate method

Teeter.java

```
...
  private void animate(Rectangle board) {
    board.setTranslateY(25);
    board.getTransforms().add(new Translate(0, -25));
    Duration d = Duration.millis(1000);
    RotateTransition rt = new RotateTransition(d, board);
    rt.setFromAngle(-23);
    rt.setToAngle(23);
    rt.setAxis(Rotate.Z_AXIS);
    rt.setAutoReverse(true);
    rt.setCycleCount(Animation.INDEFINITE);
    rt.play();
  }
...
```

A transition is used to calculate the angle

Detail: pivot point

Teeter.java

```
...
    board.setTranslateY(25);
    board.getTransforms().add(new Translate(0, -25));
...
```

This is a trick used because a `RotateTransition` only rotates an object around its centre

First the rectangle is moved down so that its centre is the same as the centre of the circle (which you could also do by changing the original rectangle coordinates)

Second, a translation is applied to move it up, leaving its centre of rotation behind

Detail: transitions

Teeter.java

```
...
    Duration d = Duration.millis(1000);
    RotateTransition rt = new RotateTransition(d, board);
    rt.setFromAngle(-23);
    rt.setToAngle(23);
    rt.setAxis(Rotate.Z_AXIS);
    rt.setAutoReverse(true);
    rt.setCycleCount(Animation.INDEFINITE);
    rt.play();
...
```

The start, end and time of the transition allow it to interpolate

Rotating about the Z axis means rotating in the X-Y plane

Detail: alternative

The trick we used to move the pivot point was ugly

To avoid, use the `Timeline` class instead of transitions

```
private void animate(Rectangle board) {
  Rotate rotate = new Rotate(-23, 200, 150);
  board.getTransforms().add(rotate);
  DoubleProperty angle = rotate.angleProperty();
  Duration d = Duration.seconds(1);
  KeyValue key = new KeyValue(angle, 23);
  KeyFrame frame = new KeyFrame(d, key);
  Timeline rt = new Timeline();
  rt.getKeyFrames().add(frame);
  rt.setAutoReverse(true);
  rt.setCycleCount(Animation.INDEFINITE);
  rt.play();
}
```


82
BLOND
BRUNETTE
TYPE IN TWO THINGS TO COMPARE
WORLD ENERGY USED NOW
COFFEE CUPS BEING PRODUCED
911 CALLS BEING MADE
CURRENT WORLD POPULATION
6707
Sprint
This is Now.
Take it to go with Mobile Broadband
YOU NOW
CLICK TO ADD YOURSELF
EGGS BEING PRODUCED
1084
CURRENT FLIGHTS
STICKY NOTES BEING PRODUCED
TOP WORDS BEING USED ONLINE
information
PLAY NOW
FORESTS CUT NOW
194
SHOPPING DAYS 'TIL CHRISTMAS
41
Get the Now widget
MONEY BEING SPENT
12100
Google
TOP SEARCHES TODAY
PEOPLE STUCK IN ELEVATORS
BABIES BEING BORN
WORLD MOOD
DOGGY DAYCARE CAM
HOUSES BUILT NOW
42.1
846
LISTEN NOW
U.S. MOON
PUSH NOW

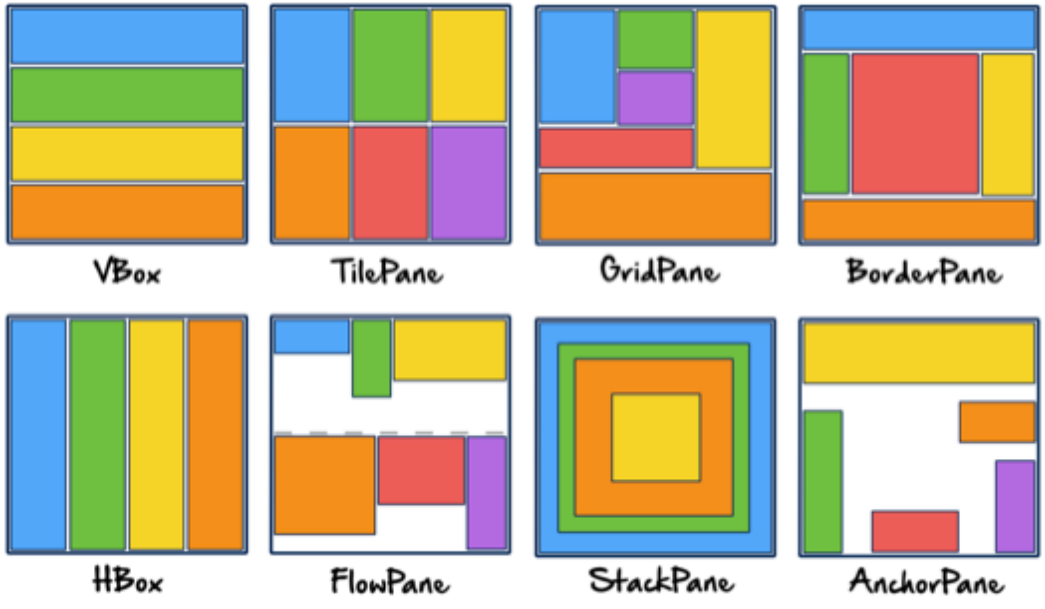
Controls

JavaFX provides lots of ready-made, off-the-shelf graphics classes, subclasses of `Control` (also called widgets) - here are a few

- Label
- Button
- CheckBox
- TextField
- TextArea
- ListView
- TableView
- ScrollBar
- Slider
- Spinner

Layout

To make these controls into a GUI, they need to be laid out using `Pane` classes (instead of `Group`)



(Image from dzone.com/refcardz/javafx-8-1)

For medium-complex cases, `GridPane` is often enough

Layout alternatives

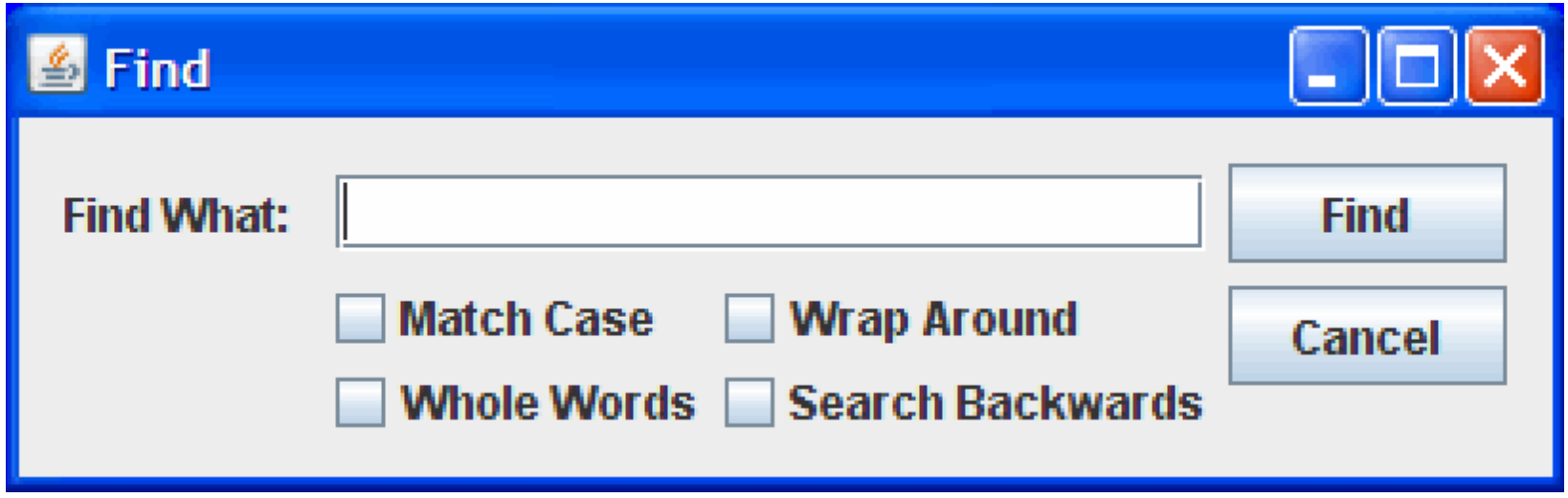
Layout of controls can be tough - some difficult issues are independence of platform measurements, pixel and screen resolution, window size

Instead of using the standard library classes, you could use the non-standard but easily available MigPane class

Or you could use an IDE with graphical layout assistance, though this is often poor (e.g. producing non-window-size-independent code or poor code, or destroying your added code if you regenerate)

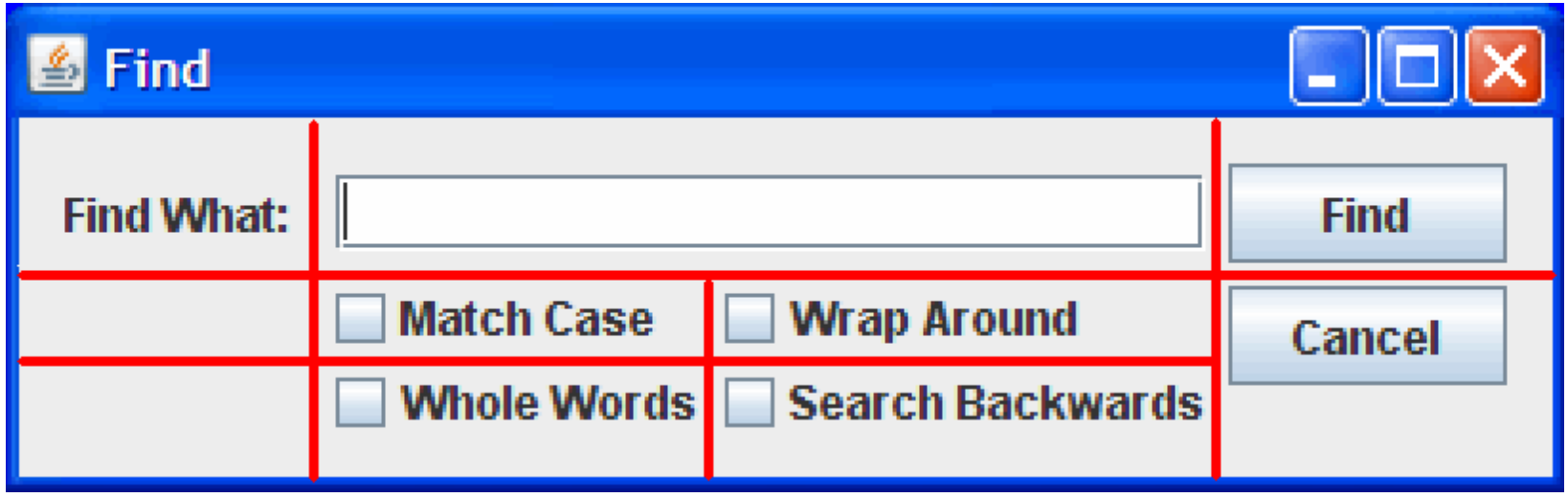
Form Layout

Suppose you want a form to look like this



Grid

It is fairly easy to see that the form can be thought of as a grid with three rows and four columns



We need the text field to span two columns, and the cancel button to span two rows

The Form program

Form.java

```
...
public class Form extends Application {
  private Label what;
  private TextField text;
  private CheckBox match, wrap, words, back;
  private Button find, cancel;
...
```

The controls are defined as fields, so we can split the setup into separate small methods

The start method

Form.java

```
...
  public void start(Stage stage) {
    GridPane pane = new GridPane();
    create();
    layout(pane);
    adjust(pane);
    Scene scene = new Scene(pane);
    stage.setScene(scene);
    stage.setTitle("Form");
    stage.show();
  }
...
```

A `GridPane` is created, and the setup is split into `create`, `layout` and `adjust` methods

The create method

Form.java

```
...
  private void create() {
    what = new Label("Find what:");
    text = new TextField("");
    match = new CheckBox("Match Case");
    wrap = new CheckBox("Wrap Around");
    words = new CheckBox("Whole words");
    back = new CheckBox("Search Backwards");
    find = new Button("Find");
    cancel = new Button("Cancel");
  }
...
```

The layout method

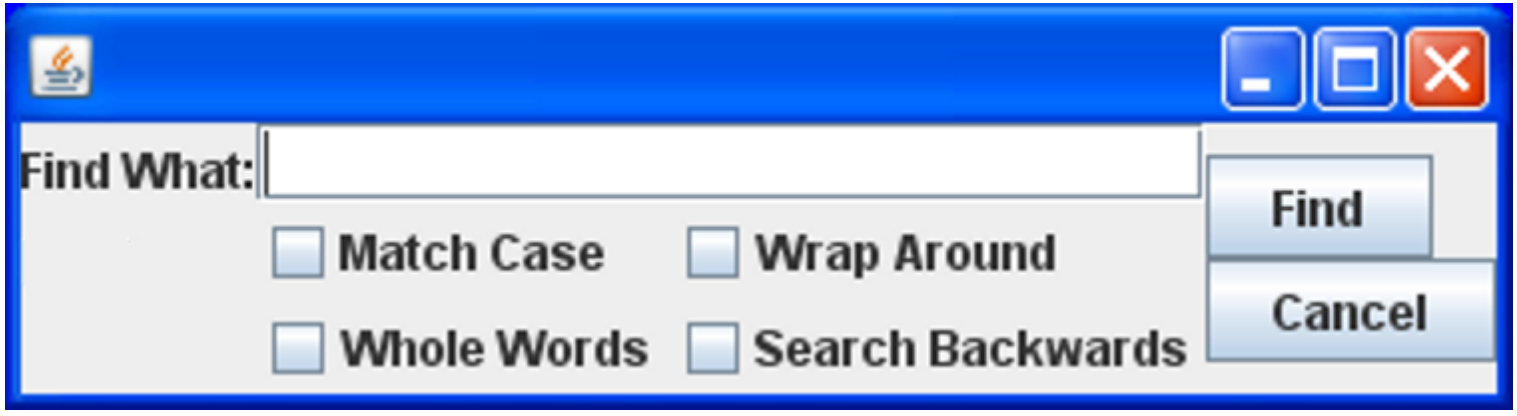
Form.java

```
...
  private void layout(GridPane pane) {
    pane.add(what, 0, 0);
    pane.add(text, 1, 0, 2, 1);
    pane.add(match, 1, 1);
    pane.add(wrap, 2, 1);
    pane.add(words, 1, 2);
    pane.add(back, 2, 2);
    pane.add(find, 3, 0);
    pane.add(cancel, 3, 1, 1, 2);
  }
...
```

This adds each control to the relevant (x,y) cell, i.e. (col,row), with a row span for `text` and a column span for `cancel`

The need for adjustment

At this point, without the `adjust` method, it looks something like this



We need gaps to space things out, and we need to widen the buttons so they have the same width

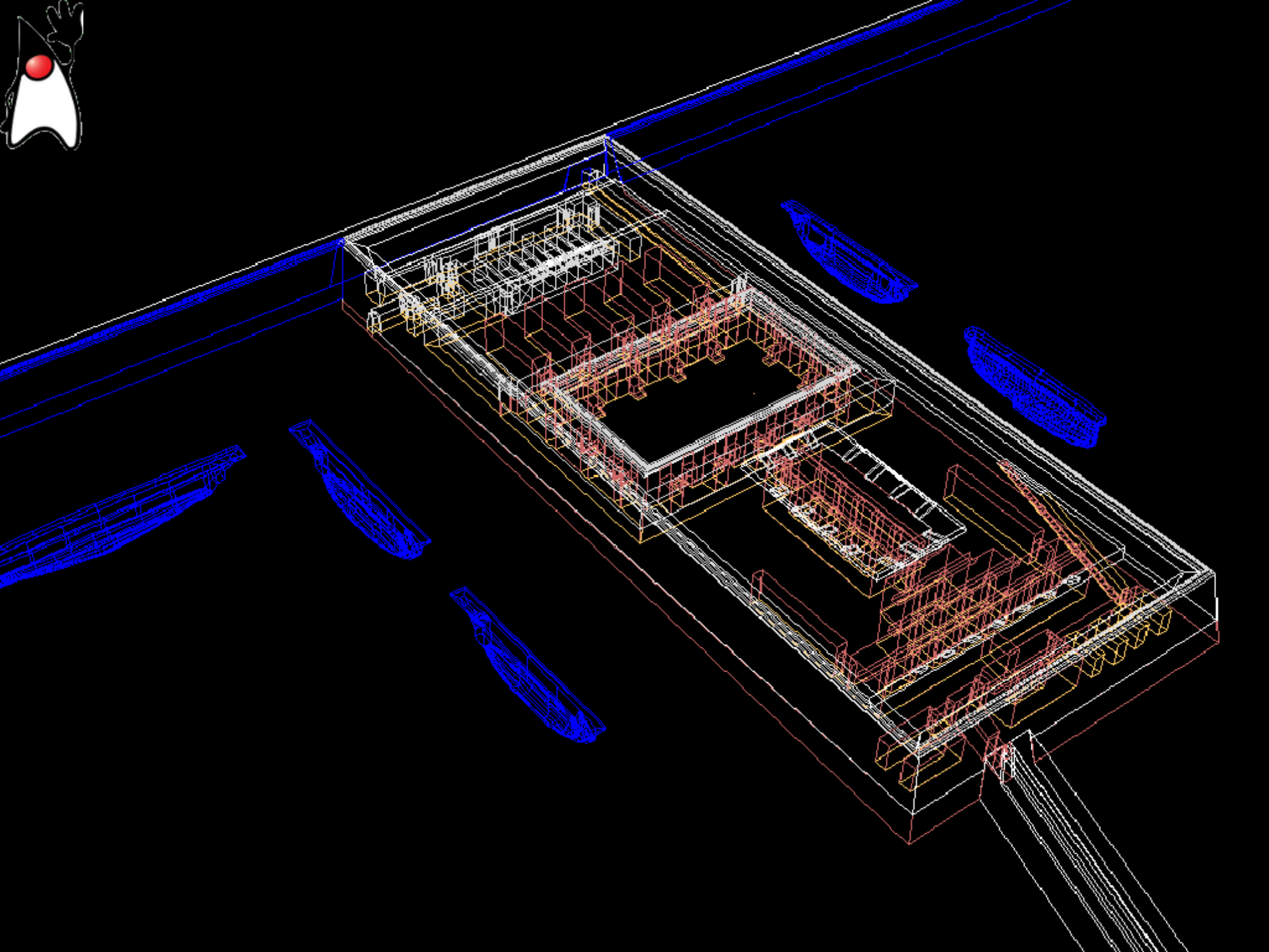
The adjust method

Form.java

```
...
  private void adjust(GridPane pane) {
    pane.setPadding(new Insets(10));
    pane.setHgap(10);
    pane.setVgap(10);
    find.setMaxWidth(300);
    cancel.setMaxWidth(300);
  }
```

The padding provides a gap round the edges, the two gap calls provide gaps between the grid cells, and increasing the maximum width of the buttons allows the grid pane to stretch them

More adjustment is needed to sort out what happens if the user resizes the window



Model-View

There is a design pattern which refers to GUI programming, with many variations such as Model-View-Controller, Model-View-ViewModel and Model-View-Presenter

What they have in common is the most important part: Model-View

That means carefully separating the graphics (View) from the logic (Model) parts of a program

Go-between

The Controller or ViewModel or Presenter mediates between the Model and the View

The differences are to do with whether the emphasis is on input or output, and on whether information is sent actively or asked for passively

Nobody seems able to agree on exactly what each pattern consists of - depending on differences in how graphics and interaction are handled in languages

The MVP pattern perhaps most closely matches the Java situation

Model-View-Presenter

The idea is that, somehow, input events from the view are converted into update calls on the model, and information about changes to the model are converted into updates to the view

The idea is that these components can be developed independently, and auto-tested more easily