Java Graphics

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

We'll see example programs, plus detailed asides

Hello World

Hi.java

```
import javax.swing.*;
class Hi implements Runnable {
  public static void main(String[] args) {
    Hi program = new Hi();
    SwingUtilities.invokeLater(program);
  }
  public void run() {
    JFrame w = new JFrame();
    w.setDefaultCloseOperation(w.EXIT_ON_CLOSE);
    w.add(new JLabel("Hello graphical world!"));
    w.pack();
    w.setLocationByPlatform(true);
    w.setVisible(true);
  }
}
```

In Java 8, there is a prettier way: Hi8.java

Details

- Swing
- invokeLater
- Runnable
- Frames
- Closing
- Placing
- Labels
- Packing
- Visibility
- Threads
- Variation
- AWT

Swing


```
import javax.swing.*; ...
```

The Swing package (`javax.swing`) provides the main graphics facilities in Java

It was originally an extension (hence `javax`) on top of an older package (`java.awt`) but is now standard

The intention was to rename it from `javax.swing` to `java.swing`, but by the time it was accepted, there was already too much code written using `javax`

invokeLater


```
... SwingUtilities.invokeLater(program); ...
```

This is instead of `program.run()`, because there are two threads, (a) to execute `main`, (b) to handle graphics

This call delays the execution of `run()` until the main program has completed its initialisation

That means, if the graphics thread accesses variables set up by the main program, they will be ready

As well as calling `invokeLater` it is important not to do anything which might start up the graphics thread

Runnable


```
... class Hi implements Runnable ...
```

The Runnable interface just contains the public `run()` method

The invokeLater method accepts any object which implements the Runnable interface, and calls its `run()` method later, when initialisation is over

(Does this follow our inheritance rule? Is a program runnable? Yes! But it isn't really right – see later)

Frames


```
... JFrame w = new JFrame(); ...
```

The `JFrame` class represents a top level window

Things in `javax.swing` start with `J` to show they are new replacements (e.g. `JFrame` extends `Frame`)

The outside of a `JFrame` (border, title, title bar, ...) belongs to the operating system, not to Java, so you have limited access to it from a program

When you create a `JFrame`, it isn't visible (so the user doesn't see it jiggling about while you set it up)

Closing


```
... w.setDefaultCloseOperation(w.EXIT_ON_CLOSE); ...
```

This call makes sure the window closes when you click the button in the top (l or r) corner of the window

Why isn't this the default? – because many programs have multiple windows, and when you close one window, you want the program to continue running with the other windows

Without this call, there is a danger that the Java program continues to run, with no windows!

Placing


```
... w.setLocationByPlatform(true); ...
```

This places the window in the default way for the platform (often variable, in 'cascading' fashion)

Call this after `pack` (so the size is known), but before `setVisible` (so there's no visible movement)

The default placement is the top left corner of the screen, or you can ask for the centre of the screen or you can specify an exact position (but then you had better find out how big the screen is)

Labels


```
... w.add(new JLabel("Hello graphical world")); ...
```

A label of class `JLabel` is just a piece of text

The call to `add()` puts the label into the frame

Warning: if you add something else, it goes on top, which is probably not what you want

By default, in a frame, you can put up to five things: in the centre, or to the north, south, west, east, but it is best to build up a single object and add that

Packing


```
... w.pack(); ...
```

You need to call pack before displaying the window

What it does is to take all the things that you have put into the window, work out how they are laid out, and as a result work out how big the window is

Warning: it is much better to size the window according to what is inside it using pack, than to set its size explicitly (for one thing, sizes and borders etc. are different on different platforms)

Visibility


```
... w.setVisible(true); ...
```

You also need to display the window

After setting up the contents, then packing, then choosing a placement, you need to make the window appear on the screen with this call

Threads

The main method sets up the window, and then exits, so why doesn't the program stop?

The answer is that creating the window also starts up a separate graphics thread, which executes at the same time as the main program thread

The main thread runs your code, and then stops, leaving the graphics thread still running

The graphics thread is responsible for drawing things and handling events such as user mouse clicks

Variation

Lots of books, tutorials, programmers use code like:

```
import javax.swing.*;
class Program extends JFrame {
  ... main(...) {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    ...
```

- Pro: no need for a `w.` prefix on the method calls
- Con: it is not thread safe
- Con: bad inheritance: "Is a Program a JFrame?" No!
- Con: it mixes 'main' and 'frame' responsibilities

AWT

The older GUI library that `javax.swing` is based on is AWT (Abstract Window Toolkit)

```
import java.awt.*;
import java.awt.event.*;
```

The problem was it used facilities provided by each platform (with lots of subtle incompatibilities)

But the AWT package is still used for lots of detailed stuff which didn't need to be reimplemented (colours, events, ...), so you need both packages all the time

The Picture program

Let's adapt Hello World to display a drawing:

Picture.java

```
...
class Picture implements Runnable {
  private Drawing drawing;
    ...
    Picture program = new Picture();
    ...

    ...
    drawing = new Drawing();
    w.add(drawing);
    ...
```

Here's a Java 8 version: Picture8.java

Still life

Let's create a 'still life' drawing of a seesaw:

Drawing.java

```
import javax.swing.*;
import java.awt.*;
class Drawing extends JPanel {
  Drawing() {
    setPreferredSize(new Dimension(400, 300));
  }
  public void paintComponent(Graphics g0) {
    super.paintComponent(g0);
    Graphics2D g = (Graphics2D) g0;
    g.fillRect(50, 100, 300, 25);
    g.fillOval(175, 125, 50, 50);
  }
}
```

Details

- Drawings
- Panels
- Redrawing
- Contexts
- Methods

Drawings

To draw something in Java graphics, you create a ***drawing object*** and ask it to draw itself

When something changes, you ask it to ***redraw*** itself

When the window gets covered up and then uncovered, or minimized and then maximized, the graphics thread will ***automatically*** ask your drawing to redraw itself

So, a drawing must ***always*** have access to the data it needs to draw or redraw itself

Panels


```
class Drawing extends JPanel
```

A Drawing is a graphics object, extending a JPanel which is a blank area to draw on

```
setPreferredSize(new Dimension(400, 300));
```

Unlike built-in graphics objects (buttons, labels, or containers with graphics objects inside) our panel has no natural size, so we must set the size

As well as the preferred size, we could set the minimum and maximum size to allow for resizing

Redrawing


```
public void paintComponent(Graphics g) {
```

A drawing object is asked to redraw itself by calling `paintComponent`

This is ***never*** called from anywhere in your own code

It is called by the graphics thread whenever it wants to, e.g. when your window has been minimized and then maximized, or covered up and then uncovered

For a still picture, you don't need access to any data, you can just use drawing methods

Contexts


```
super.paintComponent(g0);
Graphics2D g = (Graphics2D) g0;
```

`paintComponent` should always begin like this

The first line is a convention to make sure that backgrounds are drawn before foregrounds

The second line is historical - the `Graphics` (context) class has been upgraded to `Graphics2D`, but the type of the argument to `paintComponent` can't be changed for backwards compatibility reasons, so you need to cast the object

Methods


```
g.fillRect(50, 100, 300, 25);
g.fillOval(175, 125, 50, 50);
```

These draw a filled in rectangle and filled in circle

Other methods deal with different filled shapes, shape outlines, text, images, colours, fonts, clipping, collision detection (`hit`), transforms (`translate`, `rotate`, `scale`, `shear`, `transform`)

See the documentation for `Graphics2D`, including the methods inherited from `Graphics`

An animated seesaw

Play.java

```
import javax.swing.*;
class Play implements Runnable {
  private Seesaw seesaw;
  public static void main(String[] args) {
    ...
    program.animate();
  }
  void animate() {
    while (true) {
      try { Thread.sleep(1000); }
      catch (InterruptedException e) { }
      seesaw.tick();
    }
  }
  ...
}
```

Here's a Java 8 version: Play8.java

The Seesaw

Seesaw.java

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

class Seesaw extends JPanel {
  private double angle = 0.0;
  private boolean clockwise = false;

  Seesaw() {
    setPreferredSize(new Dimension(400, 300));
  }

  void tick() {
    if (clockwise) angle = angle + 0.1;
    else angle = angle - 0.1;
    if (angle >= 0.4) clockwise = false;
    else if (angle <= -0.4) clockwise = true;
    repaint();
  }
```

Details

- Animation
- Animation loop
- State
- Update
- Transforms
- Thread safety

Animation


```
... main() {
  ...
  program.animate();
}
```

An animation loop contains a `sleep` call, so it ***cannot*** be executed by the graphics thread, which must stay responsive

Instead, the loop can be executed in the main thread, by executing the animation code from the `main` method

Animation loop


```
while (true) {
   try { Thread.sleep(1000); }
   catch (InterruptedException e) { }
   seesaw.tick();
}
```

The loop runs until the program is explicitly shut down, and tells the seesaw object to update itself on each tick

The `Thread.sleep` method takes a number of milliseconds

The `try-catch` ignores interruptions

State


```
private double angle = 0.0;
private boolean clockwise = false;
```

This is the data from which the picture can be drawn at any moment

All that changes is the slope and direction

Update


```
void tick() {
  if (clockwise) angle = angle + 0.1;
  else angle = angle - 0.1;
  if (angle >= 0.4) clockwise = false;
  else if (angle <= -0.4) clockwise = true;
  repaint();
}
```

This changes the data on each 'tick'

You ***can't*** call `paintComponent` (the graphics thread must do that)

You call `repaint` to tell the graphics thread to call `paintComponent` when it is ready

Transforms


```
g.rotate(angle, 200, 150);
g.fillRect(50, 100, 300, 25);
g.fillOval(175, 125, 50, 50);
```

The rectangle and circle drawing code is the same

Instead, the entire coordinate system has been transformed by rotating it about the centre of the circle

`rotate` is one of several methods for applying a transformation to the coordinates

Thread safety

Technically speaking, the program is not thread safe

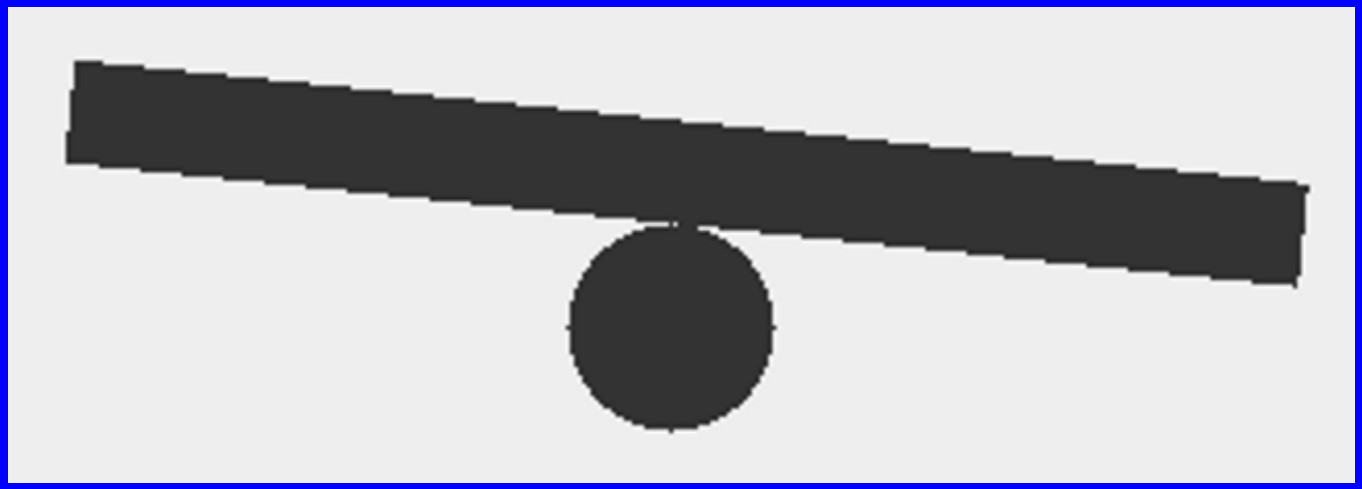
The main thread alters the fields of the `Seesaw` class, and the graphics thread reads from them

In a more complex program, the fields could be in an *inconsistent state* when the graphics thread reads them

One solution is to use `invokeLater` to get the graphics thread to do the updating, another is to use Swing timers for animating

Aliasing

You may notice jagged edges:



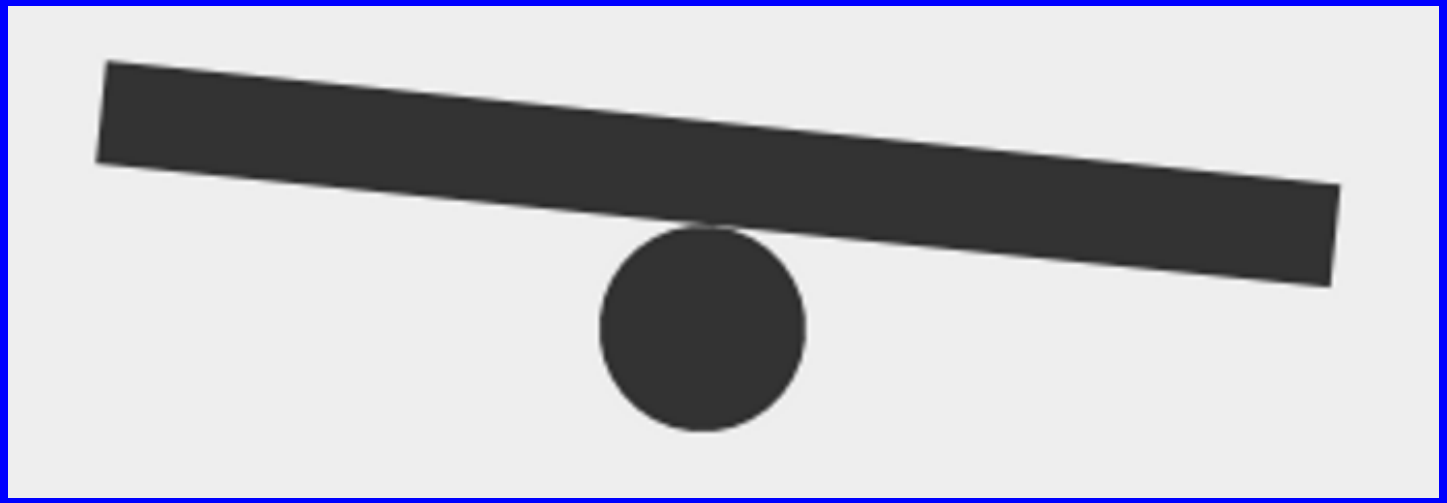
This is *aliasing* due to the pixel resolution of the screen

Anti-aliasing involves replacing each pixel with the 'average' colour of the theoretical exact square that it represents

Anti-aliasing


```
RenderingHints rh = new RenderingHints(
   RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
g.setRenderingHints(rh);
```

The quality of Java's rendering can be improved by adding this code which gives Java a hint that you would like anti-aliasing switched on if available



An image as an icon

Photo.java

```
import javax.swing.*;
import java.net.*;

class Photo implements Runnable {

  public static void main(String[] args) {
    Photo program = new Photo();
    SwingUtilities.invokeLater(program);
  }

  public void run() {
    JFrame w = new JFrame();
    w.setDefaultCloseOperation(w.EXIT_ON_CLOSE);
    URL u = this.getClass().getResource("photo.png");
    ImageIcon icon = new ImageIcon(u);
    w.add(new JLabel(icon));
    w.pack();
    w.setLocationByPlatform(true);
    w.setVisible(true);
```

Here's a Java 8 version: Photo8.java

An image as a background

Seesaw2.java

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

class Seesaw2 extends JPanel {
  private Image image;
  private double angle = 0.0;
  private boolean clockwise = false;

  Seesaw2() {
    setPreferredSize(new Dimension(400, 300));
    URL u = this.getClass().getResource("background.png");
    ImageIcon icon = new ImageIcon(u);
    image = icon.getImage();
  }

  void tick() {
    if (clockwise) angle = angle + 0.1;
```

There's a Playground.java class which uses this

Details

- Absolute images
- Relative images
- Resource images
- Images as icons
- Images as backgrounds

Absolute images

You can read an image with an absolute path

```
ImageIcon image = new ImageIcon("C:\Pics\pic.png");
```

This won't work if you move the program to a different platform

It won't even work if you move the program onto a different computer with the same platform if its files are laid out differently

Relative images

You can read an image with a relative path

```
ImageIcon image = new ImageIcon("pic.png");
```

This is ***not*** relative to the directory the program is in, it is relative to the directory you happen to be in when you ***run*** the program, so it is not reliable

In any case, it ***fails*** when you publish the program, e.g. as an executable jar file

Resource images

For *resources*, files belonging to the program, do this:

```
import java.net.*;
...
URL u = this.getClass().getResource("images/pic.png");
ImageIcon image = new ImageIcon(u);
```

This is relative to the directory where the current class is stored

This works while developing the program, no matter what platform you are working on

It works when the program is zipped up into a jar file

Images as icons

You can use an image as a graphics object by wrapping it in a label and placing the label into the GUI

```
ImageIcon icon = ...
JLabel label = new JLabel(icon);
frame.add(label);
```

Images as backgrounds

Or you can paint an image as a background within a `paintComponent` method

```
ImageIcon icon = ...
Image img = icon.getImage();
...
public void paintComponent(Graphics g)
{
    g.drawImage(img, 0, 0, null);
    ...
}
```

Counting clicks

The program `Click.java` looks like this:

When you click in the window, the number of clicks is counted

The program has two classes, `Click.java` and `Counter.java`

The counter

Counter.java

```
import javax.swing.*;
import java.awt.*;
class Counter extends JTextField {
    private int n = 0;
    Counter() {
        setText("" + n);
        setEditable(false);
        setHorizontalAlignment(CENTER);
        setColumns(15);
    }
    void count() {
        n++;
        setText("" + n);
    }
}
```

The counter is a drawing object, based on a text widget

The Click program

Click.java

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

class Click implements Runnable, MouseListener {
  private Counter counter;

  public static void main(String[] args) {
    Click program = new Click();
    SwingUtilities.invokeLater(program);
  }

  public void run() {
    JFrame w = new JFrame();
    w.setDefaultCloseOperation(w.EXIT_ON_CLOSE);
    w.setTitle("Click Demo");
    counter = new Counter();
    counter.addMouseListener(this);
```

Details

- Events
- Event types
- MouseListener
- Listeners
- Mouse events

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

Event types

A JFrame generates whole-window events:
`windowOpened`, `windowIconified`, `windowClosing`, ...

A JButton or JMenuItem generates action events:
`actionPerformed`

A JPanel (etc) generates mouse events:
`mouseClicked`, `mouseEntered`, `mouseExited`, ...

A program needs to provide a ***listener object*** with suitable methods to be called

MouseListener


```
... class Click implements Runnable, MouseListener { ...
```

The Click program implements two interfaces

As well as providing `run`, it also promises to provide the methods in the `MouseListener` interface

They are `mouseClicked`, `mousePressed`, `mouseReleased`, `mouseEntered`, `mouseExited`

Listeners


```
... counter = new Counter();
    counter.addMouseListener(this);
    w.add(counter); ...
```

The counter has a mouse *listener* registered on it

The listener object is the main program object (`this`)

Whenever a mouse event occurs, each listener is told

Each listener implements `MouseListener` and provides the event methods to be called

Mouse events


```
... public void mouseClicked(MouseEvent e) {
        counter.count();
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
```

Five methods have to be provided: four of them are ignored, and one causes a call to `counter.count`

There are separate `MouseMotionListeners` for the more costly job of monitoring mouse movements

Pollution


```
... class Click implements Runnable, MouseListener { ...
```

The two interfaces here are *pollutants*

That's because the methods `run` and `mouseClicked` are concerned with the how the class is ***implemented***, not how it's ***used***, so they don't belong in its API and shouldn't be called by other classes

If this was a support class or a library class, this could become an important issue, reducing robustness

Are there other ways to achieve the same result?

Details

- Run1
- Run2
- Inner classes
- Run3
- Anonymous inner classes
- Run4
- Lambdas
- Run5

Run 1

Run1.java

```
class Run1 implements Runnable {

    public void run() { System.out.println("run"); }

    public static void main(String[] args) {
        Runnable program = new Run1();
        program.run();
    }
}
```

This illustrates the main technique we have been using with the `Runnable` interface

The `program` variable has type `Runnable`, as it would have if passed as an argument to a library method

Run 2

Run2.java

```
class Run2 {

    class Proxy implements Runnable {
        public void run() { System.out.println("run"); }
    }

    public static void main(String[] args) {
        Run2 program = new Run2();
        Runnable proxy = program.new Proxy();
        proxy.run();
    }
}
```

The call `program.new Proxy()` means call `new Proxy()` as if from within the `program` object

Inner classes

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

The full name of the inner class is `...Run2.Proxy`

Compiling `Run2.java` creates files `Run2.class` and `Run2$Proxy.class`

A `static` inner class is normal

An object of a non-static inner class has implicit access to the fields of the outer object in which it was created

Run 3

Run3.java

```
class Run3 {

    Runnable proxy = new Runnable() {
        public void run() { System.out.println("run"); }
    };

    public static void main(String[] args) {
        Run3 program = new Run3();
        program.proxy.run();
    }
}
```

The class which implements `run` is only needed for the creation of a single object

The definition of the class can be combined with the construction of the object

Anonymous inner classes

An *anonymous inner class* is one without a name

Compiling `Run3.java` creates files `Run3.class` and `Run3$1.class`

The notation is abominable, because a vertical block appears within a horizontal expression

```
Runnable proxy = new Runnable() { ... };
```

Even in this code, ***the semicolon at the end is essential to complete the assignment***

Run 4

Run4.java

```
class Run4 {

    Runnable proxy = () -> System.out.println("run");

    public static void main(String[] args) {
        Run4 program = new Run4();
        program.proxy.run();
    }
}
```

This version only works in Java 8

An interface contaning one function is treated the same as a function

A function can be specified using a `(args) -> body` notation

Lambdas

`() -> System.out.println("run")` is a *lambda* expression, which is an anonymous function

Find a good tutorial and discover all the goodies in Java 8, of which lambda expressions are probably the most important

For the `MouseListener` interface, all the approaches work, except lambdas because the interface has more than one method in it

A Google search gives ideas about other possibilities

Run 5

Run5.java

```
class Run5 {

    private void run() { System.out.println("run"); }

    Runnable proxy = this::run;

    public static void main(String[] args) {
        Run5 program = new Run5();
        program.proxy.run();
    }
}
```

This version also only works in Java 8

The notation `object::method` allows an existing method to be passed around as a function – it is part of the lambda expression extensions of Java 8


28
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
U.S. MOON
LISTEN NOW

Widgets

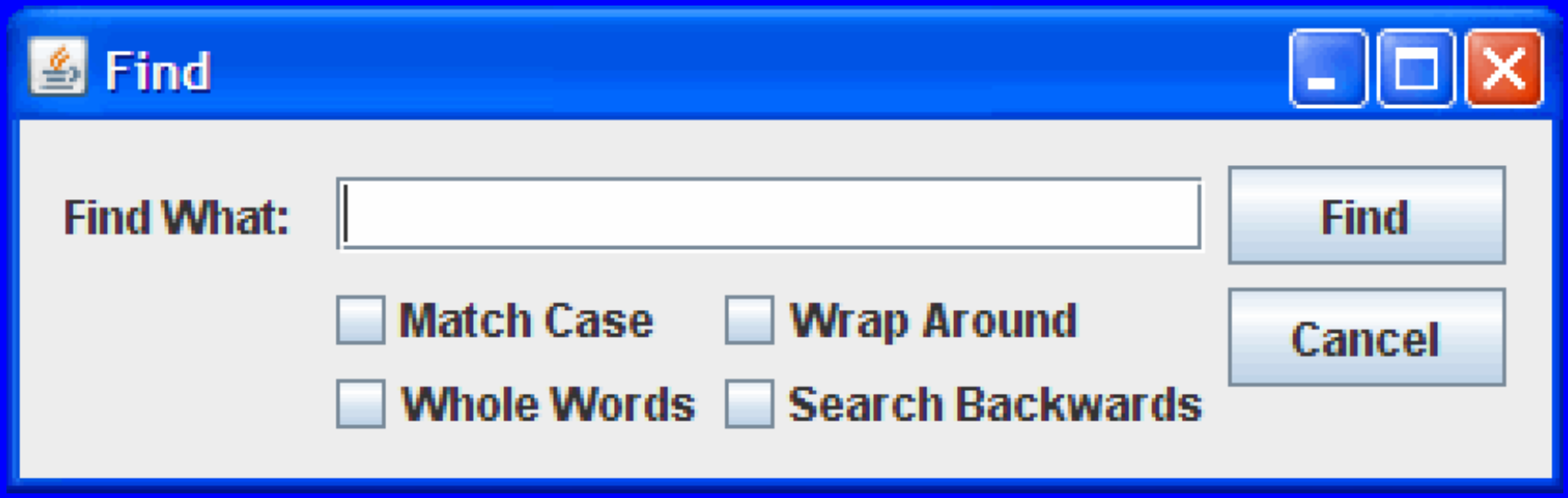
Java provides lots of ready-made, off-the-shelf graphics classes, subclasses of `JComponent` - let's call them widgets

- Box
- JPanel
- JLabel
- JTextField
- JTextArea
- JButton
- JCheckBox
- JScrollBar
- JScrollPane

Layout

You can use widgets to produce complex user interfaces

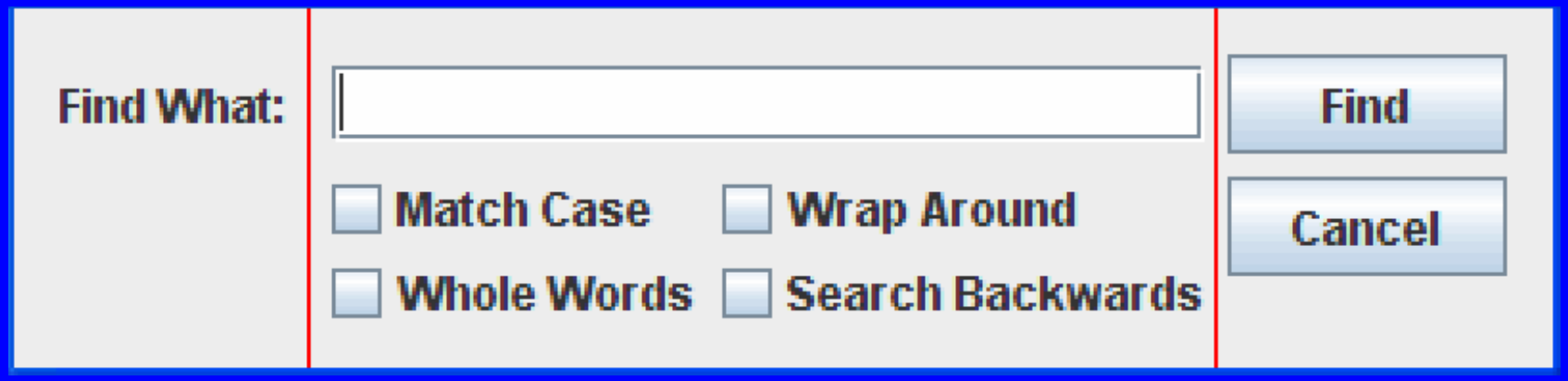
Start with a sketch of the user interface you want:



Split it as a row or column (horizontal or vertical box) of reasonably independent items

The first split

Split into a row of label | form | buttons

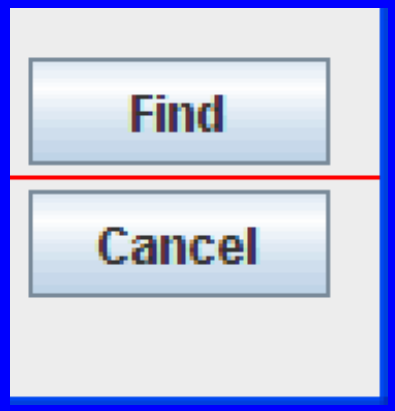


The whole display is a horizontal box containing the label then the form and then the buttons

```
Box display = Box.createHorizontalBox();
display.add(label);
display.add(form);
display.add(buttons);
```

The second split

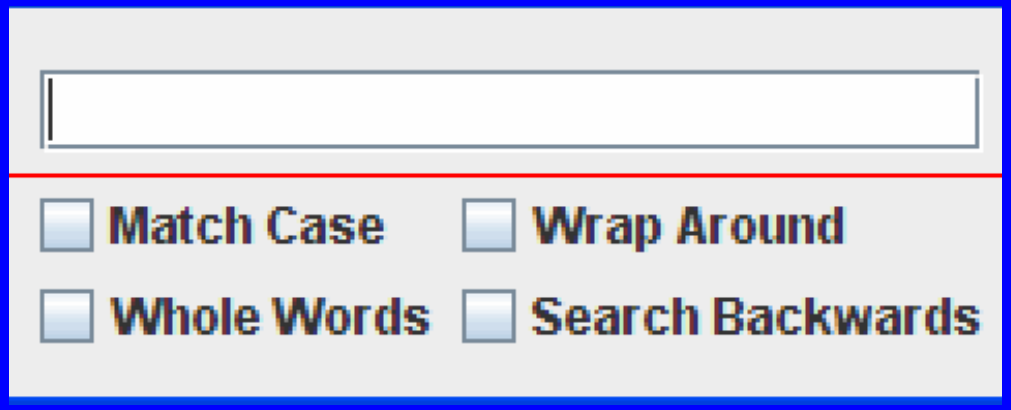
The label is easy, and the buttons form a vertical box




```
JLabel label = new JLabel("Find What:");
Box buttons = Box.createVerticalBox();
JButton find = new JButton("Find");
JButton cancel = new JButton("Cancel");
buttons.add(find);
buttons.add(cancel);
```

The third split

The form is a vertical box of a field and options

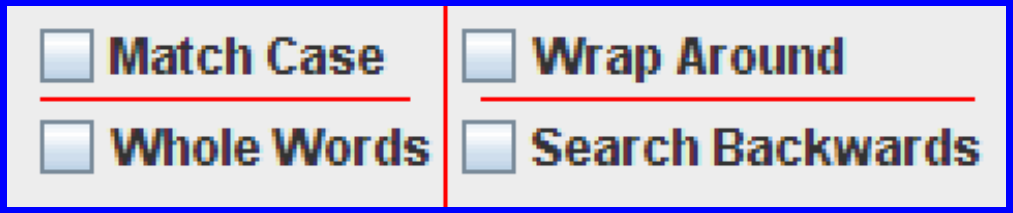



```
JTextField field = new JTextField("");
... options = ...

Box form = Box.createVerticalBox();
form.add(field);
form.add(options);
```

The fourth and fifth splits

The options can be a horizontal box of vertical boxes



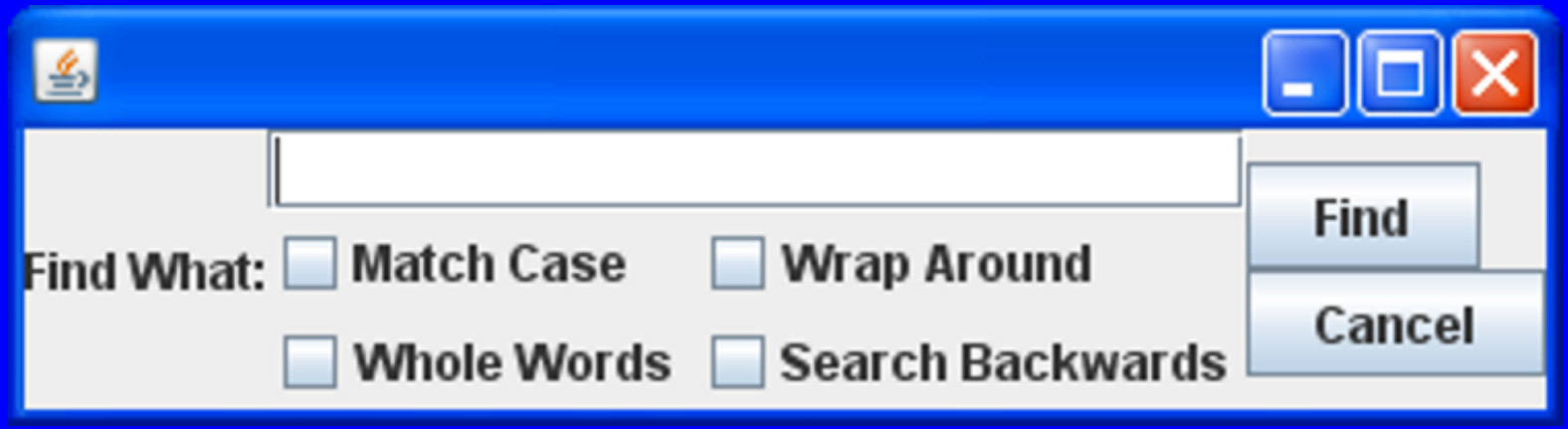

```
JCheckBox match = new JCheckBox("Match Case");
JCheckBox whole = new JCheckBox("Whole Words");
Box leftColumn = Box.createVerticalBox();
leftColumn.add(match);
leftColumn.add(whole);
...
Box options = Box.createHorizontalBox();
options.add(leftColumn);
options.add(rightColumn);
```

Putting it all together

If you stuff all the code so far into a single method, it will make a mess that is impossible to understand

Use little methods: see Boxes.java

You get something like this, which is correct except for detailed alignment, spacing and border issues



Fixing buttons

In the example program, the buttons are different sizes

One way to make them the same size is to use a grid layout with 2 rows and 1 column

```
GridLayout grid = new GridLayout(2,1);
JPanel buttons = new JPanel();
buttons.setLayout(grid);
buttons.add(find);
buttons.add(cancel);
```

The main alternative to using the Box class is to use *layout managers* (use Google to find out more)

Spacing

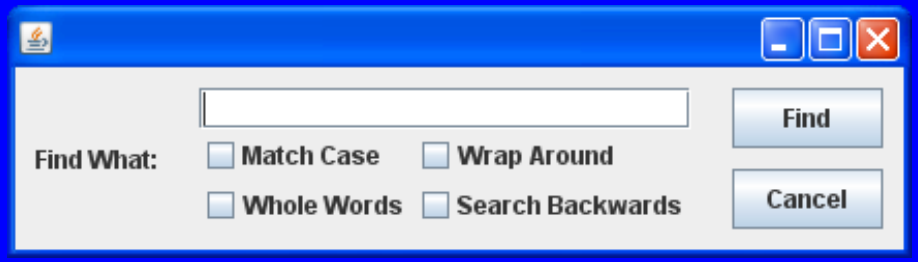
To put space between the buttons, specify horizontal and vertical spacing when creating the grid

To fix up other spacing problems, add borders to all the relevant components

```
import javax.swing.border.*;
...
GridLayout grid = new GridLayout(2, 1, 10, 10);
Border border =
  BorderFactory.createEmptyBorder(10, 10, 10, 10);
JPanel buttons = new JPanel();
buttons.setLayout(grid);
buttons.setBorder(border);
```

Alignment

The result of all the work so far is something like this, where the left label doesn't line up with the text field



One solution is to line up the items by their tops:

```
label.setAlignmentY(JComponent.TOP_ALIGNMENT);
form.setAlignmentY(JComponent.TOP_ALIGNMENT);
buttons.setAlignmentY(JComponent.TOP_ALIGNMENT);
Box box = Box.createHorizontalBox();
box.add(label);
box.add(form);
box.add(buttons);
```

Final result

The final result is Layout.java

It also adds a title in the title bar

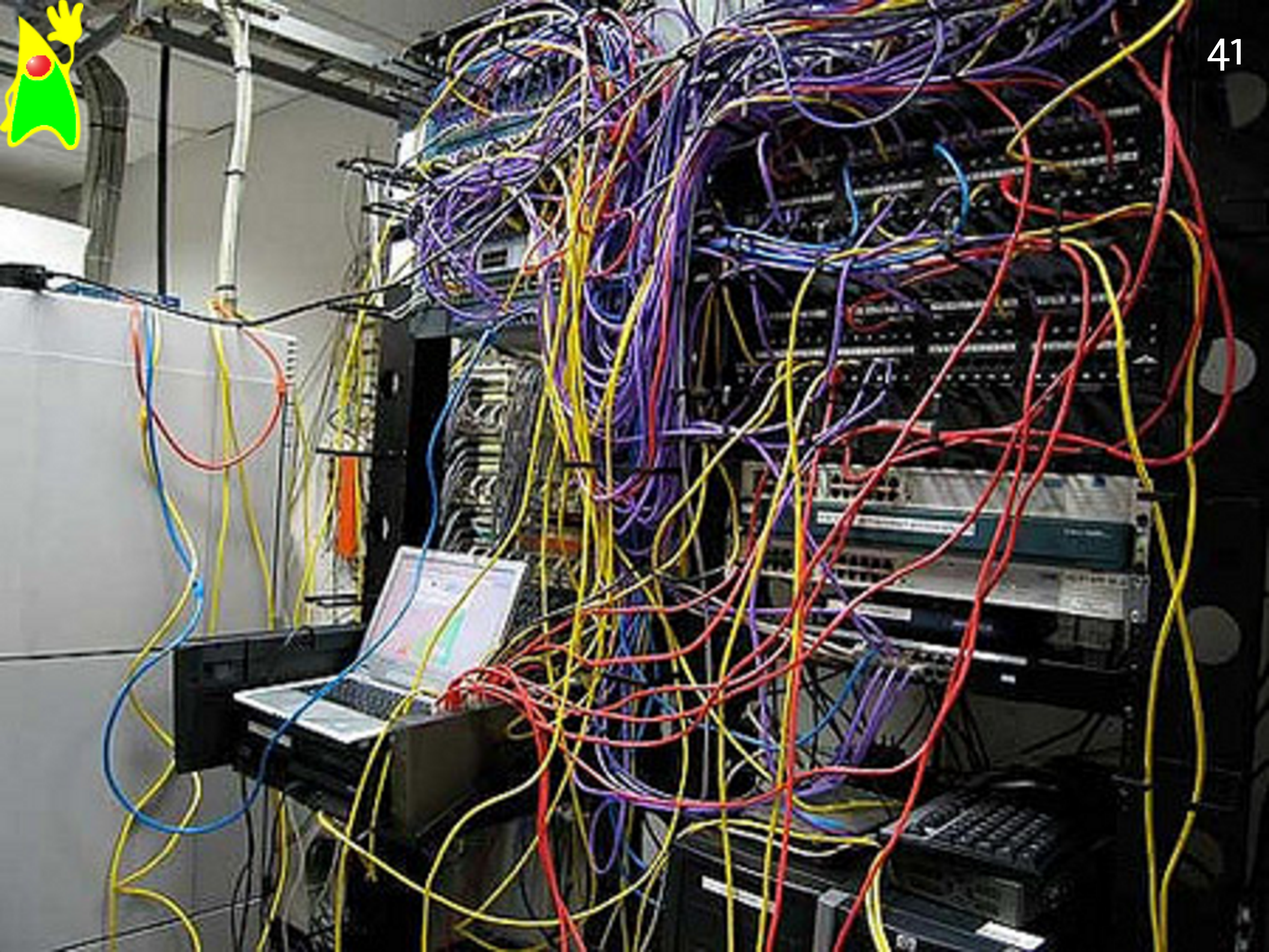
It turns out to be absolutely vital that we started from a lot of small methods, because each method has grown and become more complicated

- Graphical IDEs?

Graphical IDE tool?

It is possible to use a graphical layout tool, as in some IDEs, but there are considerable dangers:

- there is a difficult learning curve to use the tools
- the tools produce bad un-maintainable code
- after you have customised the code produced, it is difficult to change your mind about the layout
- the tools often use absolute pixel coordinates, so the result doesn't work with different platforms, or fonts, or resolutions, or when users resize windows



Wiring up

Most widgets generate action events:

```
class Program implements ActionListener
...widget.addActionListener(this);
...public void actionPerformed (ActionEvent event)
```

One listener can handle many widgets:

```
button1.setActionCommand("Up");
button1.addActionListener(this);
button2.setActionCommand("Down");
button2.addActionListener(this);
...
String action = event.getActionCommand();
if (action.equals("Up")) ...
```

Noughts and crosses

This program has 4 classes – 2 logic and 2 graphics:

- `Cross` main program, takes in a queue of moves from the user interface and passes on the moves
- `Grid` state, executes the logic of moves (currently doesn't check for reusing cells, or winning)
- `Window` provides the frame within which the program is displayed
- `Display` displays the grid, and queues up moves from the players

The Cross program

Cross.java

```
/* Enable two human players to play
noughts and crosses.  The program
concentrates on graphics and concurrency,
not game logic. */

import java.util.concurrent.*;

class Cross {
  private Grid grid;
  private BlockingQueue<Integer> queue;

  public static void main(String[] args) {
    Cross program = new Cross();
    program.run();
  }

  void run() {
    queue = new LinkedBlockingQueue<Integer>();
```

The Grid class

Grid.java

```
/* Hold the current state of the game.
That's the contents of the 3x3 grid
and whose turn it is. */

import java.util.concurrent.*;

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

  Grid(BlockingQueue<Integer> queue) {
    if (queue != null) window = new Window(queue);
    cells = new char[][] {{S,S,S},{S,S,S},{S,S,S}};
  }

  void move(int x, int y) {
    cells[x][y] = whoseTurn;
```

The Window class


```
/* Provide a top-level window for the game. */  Window.java

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;

class Window implements Runnable {
    private int task, START=0, UPDATE=1;
    private Display drawing;
    private BlockingQueue<Integer> queue;
    private int x, y;
    private char c;

    Window(BlockingQueue<Integer> q) {
        queue = q;
        task = START;
        SwingUtilities.invokeLater(this);
    }
```

The Display class

Display.java

```
/* Display the current state of the grid
on screen.  Catch mouse clicks, convert to
grid coordinates, and queue them up
for the main program. */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;

class Display extends JPanel implements MouseListener {
  private char[][] cells;
  private BlockingQueue<Integer> queue;

  Display(BlockingQueue<Integer> q) {
    cells = new char[3][3];
    queue = q;
    setPreferredSize(new Dimension(300,300));
    setBackground(Color.WHITE);
```

Details

- Concurrency
- Startup
- Shutdown
- Graphics code
- Graphics thread
- Shared data
- Low level cooperation
- High level cooperation
- Drawing objects
- Output
- Input
- Queues

Concurrency

Using graphics for the examples, how does Java's concurrency work?

A Java program has a main thread, and a graphics thread (also called the Event Dispatch Thread)

It also has logic code and graphics code, and the graphics code must run on the graphics thread

The logic code could run on either thread, but must not run on the graphics thread if causes any delay

Graphics startup

The Java graphics system and graphics thread start up the moment you 'touch' a graphics class

Suppose you have a non-GUI program which uses graphics, for example an image manipulation program

or you want to do automated testing with graphics switched off but you can't avoid graphics classes

and you work over a network with no graphics available:

```
System.setProperty("java.awt.headless", "true");
```

Graphics shutdown

A program ends when ***all*** the threads stop

It is easy to create a GUI program which works, but refuses to shut down

The graphics thread is still running, waiting for events (even with no windows left)

You may be able to use `EXIT_ON_CLOSE` or similar

Otherwise, try `System.exit(0)`

Graphics code

"All graphics code must run on the graphics thread", but what is graphics code?

```
1: SwingUtilities.invokeLater(...);
2: p = new JPanel();
3: p.setPreferredSize(...);
4: drawing.repaint();
5: drawing.update();
6: class Program implements MouseListener
```

1: no, *2: yes*, *3: yes* (graphics method), *4: no* (`repaint`, `invalidate`, `revalidate` are OK) *5: no* (if `update` doesn't call any graphics methods other than `repaint`) *6: no* (an interface contains no code)

Graphics thread

Which code (of yours) runs on the graphics thread?

The `run` method triggered by `invokeLater`, `paintComponent`, all callbacks (`mouseClicked`, `actionPerformed`, ...) and anything they call, directly or indirectly

Every method called by the graphics thread must be *quick*: it can't sleep, print, load, do long calculations, ...

For example, load images before making the window visible, or create a worker thread or queue up the work for the main thread

Shared data

Java uses ***shared-memory*** concurrency

Many threads wander around your collection of objects, independently executing code

If you think it is simple, read Java language spec, 17.4

Danger: whenever an object or field can be accessed by multiple threads, things might go wrong

Low level cooperation

To get round the problems, there are many low level language and library features:

The `synchronized` and `volatile` keywords, `wait`, `notify`, `notifyAll`, `yield`, `join`, `Semaphor`, `Exchanger`, `Executor`, `ForkJoinPool`, `Future`, `CountDownLatch`, `CyclicBarrier`, `Phaser`, `ReentrantLock`, `AtomicInteger` etc

Advice: they are not debuggable: if you use them, your program is probably too complex

High level cooperation

A different approach is to make sure that every shared object or method is from the libraries and is ***thread-safe***

Check the documentation of classes for thread-safety

There is still an initalization problem: if one thread creates the shared data structure in a variable, how does the other know the variable has been initialized by the time of its first access?

Drawing objects

One non-thread-safe data structure is a drawing object:

```
class Drawing extends JPanel {
   fields
   void update() { update fields; repaint(); }
   public void paintComponent() { read fields }
}
```

The main thread may be half way through updating the fields, with the field values in an inconsistent state, when the graphics thread reads the fields to draw with

If the fields are simple enough, you can get away with it

Output

Code on the main thread, for example an animation loop, ought not to access drawing objects directly

It can call `invokeLater` to pass the responsibility to the graphics thread

If the object which has the `run` method is shared, e.g. the main thread fills in fields which are picked up by the graphics thread, `invokeAndWait` can be used to make sure the main thread doesn't change the fields again before they are used

Input

How can the main thread deal with events? It must wait somehow until one is available

A useful thread-safe structure is the `BlockingQueue`, (using the `Linked` or `Array` implementation)

The graphics thread can add items to the queue, which makes sure that none get lost

The main thread can call `take`, which waits if the queue is empty instead of returning null or throwing an exception

Queues

Suppose there is a queue shared by the two threads: one thread must own and create it, but how does the other thread know when it has been initialised?

The best bet is for the main thread to create it, and then pass it to the graphics thread through an `invoke` call

The queue should not contain graphics objects, partly because the main thread mustn't touch graphics objects

Also, e.g., mouse clicks contain pixel coordinates, which only graphics code should know about

The end