Background

These notes follow on from the previous set but move further into threading & concurrency.

As before examples are accessed through one program which allows you to select which example to run. The structure of this front-end program is different to the last – examine this & be critical we will discuss these issues sometime.

Remember that the descriptions here brief reminders only (not an independent learning resource). [*1] included in these notes or program comments means you will need to turn up to a lecture/seminar for a proper explanation.

ThreadEGs1

(primary program ThreadEGs1.java)

the basics

These notes still use a moving blob & assume you either understand the various issues about (extremely) simple animated graphics in Java or have followed through the previous examples.

The *blob* is now defined in its own class (which is an improvement). There are a few key files which are described first before looking at other issues.

Also note: for my own applications I tend to use the following...
1. *Defs.java* – this contains global *values* which need to be used by multiple classes;
2. *Utils.java* – this is where I keep general purpose functions, some of these are application specific, other functions are more general.

I have combined all of these into *Defs.java* for now.

the AbstractBlob class

This is a wrapper to contain most of the functionality of blobs but not their *startBlob* or *run* methods (the *startBlob* method mostly just starts threads). I decided not to make AbstractBlob implement *Runnable* preferring to leave this up to the subclasses.

class Blob1 run method

```
display();
while( notDeadYet() )
{     snooze();
      erase();
      age();
      if( notDeadYet() )
      {     move();
            display();
      }
}
```

example-1

code activated by the "start ball" button...

```
Blob1 b = new Blob1( ... );
b.setMotionParameters with cords(20,20) & movement(2,3)
b.startBlob( initialAge );
```

issues...
1. it is possible to release multiple blobs – why?
2. releasing lots of blobs messes up the graphics – why?

example-2

code activated by the "start ball" button...

```
if( gBlob == null )      // 1st one hasn't been built yet
{   gBlob = new Blob1( ... );
    gBlob.setMotionParameters with cords(20,20) & movement(2,0)
}
gBlob.startBlob( initialAge );
```

issues (ask yourself why/why not about each of these)...
1. it is not possible to release multiple blobs;
2. re-clicking the "start ball" button makes the blob move faster;
3. sometimes the blob seems to slow down;
4. the graphics still screw up after multiple button clicks.

class Blob2

this class tries to prevent multiple simultaneous blobs from running, significant parts are...

```
boolean isActive = false;           // true when blob is running

public void startBlob( int lifeSpan )
{   if( !isActive )
    {   isActive = true;
        setAge( lifeSpan );
        thread = new Thread(this);
        thread.start();
    }
}
```

single runner Blob2

similar to first example but uses Blob2 in place of Blob1, code activated by the "start ball" button...

```
Blob2 b = new Blob2( ... );
b.setMotionParameters( ... );
b.startBlob( initialAge );
```

issues...
1. it doesn't work – why not?

Blob3 is better constructed than Blob2 because it uses the *Thread.isAlive* method rather than inventing its own mechanism (Blob2 used isActive for this). See below...

```
Thread thread = null;

public void startBlob( int lifeSpan )
{     if( thread == null || !thread.isAlive() )
      {     setAge( lifeSpan );
            thread = new Thread(this);
            thread.start();
      }
}
```

Blob3 is used in the "single runner Blob3" example but it still does not work – same reasons.

synchronization

(still using ThreadEGs1.java)

The notes now move on to issues involving synchronisation. As always – a good resource for general reading is the online Java tutorials. The next couple of examples are using synchronisation to (try to) cure the graphics problem where multiple blobs leave a kind of a trail which does not get erased.

synch eg-1

Uses SynchBlob1, a version of Blob1 which differs only that (by extending a synchronized abstract blob class) the *erase* & *display* methods are synchronized but everything else is the same and... the problem still exists.

I have included this example because it seems like a possible solution worth investigating. Check it out & understand why it does not solve the *graphics trail* problem.

synch eg-2

This version uses a version of Blob1 which has an additional static variable & a modified *run* method (its *erase* & *display* methods are not synchronized). The declaration & run method looks like this...

```
static Object lock = new Object();

public void run()
{     display();
      while( notDeadYet() )
      {     snooze();
            synchronized(lock)
            {     erase();
                  age();
                  if( notDeadYet() )
                  {     move();
                        display();
                  }
            }
      }
}
```

Issues...
1. this finally solves the display problem – for further explanation see [*1];
2. what would happen if (i) lock was not static or (ii) is was left uninitialised?

a more complex example

files currently in use...
dir: multiframe2
SCL014.java, Defs.java, PrepRunner.java, PrepRun.java

the general concept

The program animates by displaying changing images one after another. It maintains an array of background images and uses different threads to manage them in particular...
1. the standard main thread that the program runs in – this allows the GUI to operate even when other work is going on;
2. a thread to regularly (based on a timer) triggers the displaying of the (logically) next image in the array;
3. a thread to prepare (in advance) other images in the array.

classes

The link/contract between the main program and the triggering & preparing thread classes is defined by an interface class. The various classes are arranged as follows...

MainPanel
: the class that (i) initializes everything (ii) builds the main GUI/visuals (iii) implements the contractual methods defined by the PrepRunner interface class;

PrepRun
: a wrapper class – owns the following 2 classes, coordinates them and provides accessor/muator methods for key variables, etc

PrepRun.TriggeringThread
: this periodically (on a delay type timer) triggers the MainPanel.useData method which, in this program, display the next image in the animation sequence;

PrepRun.PreparingThread
this periodically (on a delay type timer) triggers the MainPanel. prepareData method which prepares off-screen images for later display;

interface methods

public void useData(int n)
this is the trigger to use the n^{th} item in the data array. In this program where the data array is a collection of images this means making the n^{th} image visible;

public void prepareData(int n, int f)
this is the trigger to prepare the n^{th} item in the data array. Where "n" refers to an array index "f" is the number of the image since the start of the sequence (so n wraps around but f does not). In this program, where the data array is a collection of images, this means doing the visuals for the f^{th} image & placing it in the n^{th} slot of the image array;

features of main class

key points...

- the "data" is an array of BufferedImage – these can be manipulated as off-screen images & then drawn on the visible screen when required. If you are doing much with graphics BufferedImages are worth considering (check the Java tutorials, etc);
- the images (here) are built up in a couple of layers (i) the background image and (ii) the *layer1* image which is drawn on top of it;
- the "runner" variable is set up as the PrepRun. It is initialised as...
  ```
  runner = new PrepRun( this, numFrames )
  ```
  "this" is provided as the link back for PrepRun to call the interface methods & "numFrames" is the number of elements in the data array – the PrepRun class needs to know about this;
- the Mainpanel class handles all the normal initialisation & sets up the (minimal) GUI;
- there is a note about the line...
  ```
  MainPanel main = this;
  ```
  ...this is to make it easier to retrospectively rewrap the main classes.

notes re: methods & listener code

- "start" button calls runner.start() – this starts the two PrepRun threads to trigger preparation & display of images;

- "stop" button calls runner.stop() – causes the PrepRun threads to halt (but in a way they can be restarted);

- the useData drawing process – this is triggered by one of the PrepRun threads (& will execute within that thread). useData just keeps the number of the frame no. its passed in a global variable (so it can be used by the paintComponent method of the graphics panel) then calls repaint on the graphics panel. The graphics panel paintComponent method checks that the image has been prepared and, if so, draws it. NB: the first time(s) this paintComponent method will be called is during the construction process for the main panel – no image will have been prepared at this stage.
- prepareData – this is triggered by one of the PrepRun threads (& will execute within that thread). prepData takes two arguments... **i** the index into the BufferedImage array indicating the image it should prepare and **f** the frame no. in the sequence of

frames it is preparing. prepData does a check on the value of **f** – this is only a demo program so it pauses after it has run for a while. Assuming the value of **f** is ok (the else clause) the method draws the background & layer1 images into the i^{th} slot of the images array. Note the need for getGraphics on the BufferedImage.

threading issues

The PrepRun class has two subclasses: TriggeringThread & PreparingThread. Both subclasses run threads by implementing Runnable rather than overriding Thread – this gives a bit more control. They use a variable stopSignalled to indicate stop/pause conditions – check in the code how this is used.

TriggeringThread.run cycles around as long as stopSignalled has not been set doing the following...

1. calling.useData(elementInUse) on the MainPanel – the class that implements the PrepRunner interface;
2. it increments the elementInUse variable (allowing for wrap-around);
3. it sleeps for a given delay period.

PreparingThread.run cycles around as long as stopSignalled has not been set. Remember that this thread is causing images to be prepared in advance of their use so it is calling the MainPanel.prepData method on slots of its BufferedImage array that have been used but not yet prepared. It uses the "elementInPrep" variable to keep track of which images have already been prepared.

In some PreparingThread.run cycles there will be no work to do (all images will already be prepared) so the thread will just sleep (NB: by default the prep delay is considerably longer than the use delay). When there is work to be done the thread will call the MainPanel.prepareData method with appropriate arguments.

things to consider

This example can be considerably better organised in various ways, once you have read through the code (& followed the lectures) think about the advantages/disadvantages of reorganising it...

1. by passing the data buffers (images or whatever) to the subclasses/thread objects;
2. splitting PrepRun into two classes (i) doing the use triggering (ii) doing the data prep triggering;
3. explicitly coding preparation activity into the places where delays are placed.