Lecture 1

So far our programs have been *sequential*: only one thing has been happening at a time. In particular, if object A sends a message to object B, then A 'does nothing' (it loses control) while B is executing the appropriate method. When B finishes, control returns to A, so A resumes.

In this lecture we see how to write a program in which methods can execute concurrently. E.g. A can be getting on with something else while B is responding to the message.

As an illustration of the use of very simple concurrency, we write some programs that perform animations. (These are simple because they do not raise any issues concerning synchronised access to shared data.)

They are easily written by using the `Timer` class that is part of Swing.

Java threads

- The Java libraray has a class `Thread`, having, e.g., a zero-parameter constructor
- A `Thread` object will have a `run()` method, which tells the system what to do when it runs this thread
- Start a `Thread`, `t`, using `t.start();` — this invokes the `run` method
- By default, `run` does nothing. We must define our own useful `run` methods.

A pinging and ponging program

```
public class Pinger implements Runnable
{
   public void run()
   {  while (true)
      {  System.out.print("ping ");
      }
   }
}
---------------------------------------

public class Ponger implements Runnable
{
   public void run()
   {  while (true)
      {  System.out.print("pong ");
      }
   }
}
```


Creating and starting two threads

```
public class PingPong
{
   public static void main(String[] args)
   {  Pinger p1 = new Pinger();
      Thread t1 = new Thread(p1);
      Ponger p2 = new Ponger();
      Thread t2 = new Thread(p2);
      t1.start();
      t2.start();
   }
}
```

`start` returns to its caller immediately; the caller then executes concurrently with the launched thread

Questions. When we run `PingPong`*, how many threads are launched? And what is the program's output?*

Remarks

- Actual behaviour depends on the scheduling policy, which may vary from platform to platform.
- To 'guarantee' concurrency, write programs that encourage unfair schedulers to be fairer, e.g. by putting one thread to sleep
- There is much more to be said particularly if you want threads to communicate with each other or share some resource
- Their access to shared data/objects must be synchronised so that, e.g., one thread waits if another is accessing. But, waiting brings risks of deadlock
- Java provides built-in facilities and further methods in the `Thread` class definitions for this kind of advanced concurrent programming — see courses on, e.g., distributed systems

Animation

- Motion will be perceived if we show successive 'frames' (images) at a rate of 8-24 frames per second
- The animation loop should be in its own thread, otherwise nothing else could go on at the same time (e.g. button clicks wouldn't be responded to)
- Swing provides the `Timer` class, which makes it easy to write animations
  - Constructor: `public Timer(int delay, ActionListener)`
    Creates a thread object; every `delay` milliseconds, it creates an `ActionEvent`; the `ActionListener` responds to the event
  - Methods include: `start()`, `stop()`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class AnimatedLabel extends JLabel
{  public AnimatedLabel()
   {  super();
      timer = new Timer(1000,
         new ActionListener()
         {  public void
             actionPerformed(ActionEvent ae)
            {  frameNumber++;
               setText("" + frameNumber);
            }
         });
   }
   public synchronized void startAnimation()
   {  timer.start();
   }
   public synchronized void stopAnimation()
   {  timer.stop();
   }
   private Timer timer;
   private int frameNumber = -1;
```



```
public static void main(String[] args)
{ JFrame frame = new JFrame("Timer Demo");
  final AnimatedLabel lbl =
    new AnimatedLabel();
  Container contentPane =
    frame.getContentPane();
  contentPane.add(lbl);
  frame.addWindowListener(
    new WindowAdapter()
  { public void windowClosing(WindowEvent we
    { System.exit(0);
    }
  });
  frame.pack();
  frame.setVisible(true);
  lbl.startAnimation();
}
```


Displaying an image repeatedly

We load 10 images into an array. In response to `Timer` events, we update the image on the `JLabel`:

```
public AnimatedLabel()
{  super();
   images = new ImageIcon[NUM_OF_IMAGES];
   for (int i = 0; i < NUM_OF_IMAGES; i++)
   {  images[i] =
         new ImageIcon("duke" + i + ".gif");
   }
   timer = new Timer(50,
      new ActionListener()
   {  public void
        actionPerformed(ActionEvent ae)
      {  frameNumber++;
         setIcon(images[frameNumber %
                          NUM_OF_IMAGES]);
      }
   });
}
private ImageIcon[] images;
private static final int NUM_OF_IMAGES = 10;
```


Polite use of resources

If there is any way the animation can become invisible, then, rather than allow it to continue to consume resources, we should suspend it. E.g. we can respond to window iconification and deiconification:

```
frame.addWindowListener(new WindowAdapter()
{ public void windowIconified(WindowEvent we
  {  lbl.stopAnimation();
  }

  public void windowDeiconified(WindowEvent
  {  lbl.startAnimation();
  }

  public void windowClosing(WindowEvent we)
  {  System.exit(0);
  }
});
```


Politeness to the user

Users should be able to switch off distracting animations, e.g., by clicking on them.

```
// Add this to the constructor:
addMouseListener(new MouseAdapter()
{  public void mousePressed(MouseEvent me)
   {  if (frozen)
      {  frozen = false;
         startAnimation();
      }
      else
      {  frozen = true;
         stopAnimation();
      }
   }
});
// Change this method and add this variable:
public synchronized void startAnimation()
{  if (! frozen)
   {  timer.start(); }
}
private boolean frozen;
```


Custom painting & animation

The handler for the `Timer` events now calls `repaint`, and we override `paintComponent`, e.g.:

```
timer = new Timer(40, new ActionListener()
{ public void actionPerformed(ActionEvent ae
  { frameNumber++;
    repaint();
  }
});

public void paintComponent(Graphics g)
{ super.paintComponent(g);
  // Compute margin sizes (insets) &
  // availWidth & availHeigth as we
  // did in lecture 37
  g.drawOval(
    insets.left + frameNumber % availWidth,
    insets.top + availHeight / 2, 20, 20);
}
```