CSSE 220 Day 13

Threads
More algorithm efficiency analysis, Big-Oh
Work on Minesweeper

CSSE 220 Day 13

- See the nine announcements in the email message that I sent you yesterday afternoon.
- By now, everyone should know how to submit files in AFS and to SVN repositories.
  - I have been rather lenient in the past if you didn't get it submitted correctly. By now you should be able to submit it to the right place on time.

Key Concepts quiz tomorrow

- It is important that we not only be able to write object-oriented programs, but that we build a vocabulary that enables us to communicate with each other about them.
- That is why I asked you to spend four weeks learning the "lingo" of OOP in Java.
- Tomorrow is the check-up on that.
- This ANGEL-based quiz is closed book and notes.
- It consists of matching questions, and you will only have about 30 seconds per term to complete it. So know your terms well!

Class this week

- Each class day this week.
  - Discuss a Java feature (threads, function objects)
  - A little bit on algorithm analysis
  - Some time to work on Minesweeper (typically 30+ minutes).
- A progress report is due at the end of each class.
  - It is basically an updated version of your IEP, showing your progress on the phases that you outlined.
  - Name today's report **Day 13 progress Report.xlsx** (You should be able to use "Save as" in Excel to do this.)
  - Commit it to your Minesweeper repository.

Minesweeper Note

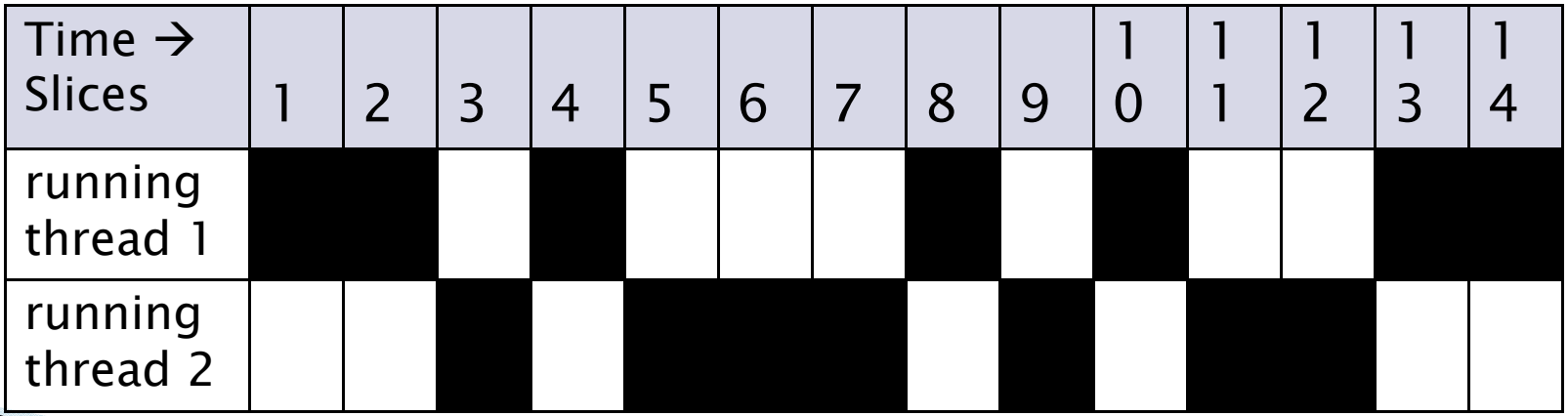
- Additional requirement for your project: You should add a "cheat" feature
  - to help you debug your code
  - to help me test your code more easily
- Details are on the **Assignments discussion forum**

Today's agenda

- Multithreaded Programs
- More on Algorithm analysis – Big Oh
- Work on Minesweeper

Multithreaded programs

- Often we want our program to do multiple (semi) independent tasks at the same time
- Each thread of execution can be assigned to a different processor, or one processor can simulate simultaneous execution through "time slices" (each probably a large fraction of a millisecond)



A Java Program's Threads

- There is always one default thread; you can create others.
- Uses for additional threads
  - Animation that runs while still allowing user interaction.
  - A server (such as a web server) communicates with multiple clients.
  - Animate multiple objects (such as the timers in the CounterThreads example – in a few minutes).
- A thread may sleep for a specified amount of time by calling

```
Thread.sleep(numberOfMilliseconds);
```

The Emperor's New Threads

- How to create and run a new thread
  - Define a new class that implements the **Runnable** interface. (it has one method: **`public void run();`**)
  - Place the code for the threaded task in the run() method:
    - ```
      class MyRunnable implements Runnable {
          public void run () {
              // task statements go here
          }
      }
      ```
  - Create an object of this class:
    - **`Runnable r = new MyRunnable();`**
  - Construct a Thread object from this Runnable object
    - **`Thread t = new Thread(r);`**
  - Call the **start** method to start the thread
    - **`t.start();`**
      ```

Threads examples (in your SVN repos.)

- **Greetings** –simple threads, different wait times
- **AnimatedBall** – move balls, stop with click
- **CounterThreads** – multiple independent counters
- **CounterThreadsRadioButtons** – same as above, but with radio buttons.

The remaining two are more advanced than we will use in this course, dealing with race conditions and synchronization. Detailed descriptions are in *Big Java*.

- **BankAccount**
- **SelectionSorter**

Simple example (1) – Output

One thread prints the **Hello** messages; the other Thread prints the **Goodbye** messages.

Each thread sleeps for a random amount of time after printing each line.

```
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Goodbye, World!
Thu Jan 03 16:09:36 EST 2008 Hello, World!
Thu Jan 03 16:09:37 EST 2008 Goodbye, World!
Thu Jan 03 16:09:37 EST 2008 Hello, World!
Thu Jan 03 16:09:38 EST 2008 Hello, World!
Thu Jan 03 16:09:38 EST 2008 Goodbye, World!
Thu Jan 03 16:09:38 EST 2008 Goodbye, World!
Thu Jan 03 16:09:38 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:39 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Hello, World!
Thu Jan 03 16:09:39 EST 2008 Goodbye, World!
Thu Jan 03 16:09:40 EST 2008 Hello, World!
Thu Jan 03 16:09:40 EST 2008 Goodbye, World!
. . .
```

This example was adapted from Cay Horstmann's *Big Java*, Chapter 23

Simple example(2) – GreetingThreadTester

```
public class GreetingThreadTester{

  public static void main(String[] args){

    // Create the two Runnable objects
    GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
    GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");

    // Create the threads from the Runnable objects
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);

    // Start the threads running.
    t1.start();
    t2.start();
   }
}
```

We do not call **run()** directly.
Instead we call **start()**, which sets up the thread environment, and calls **run()** for us.

Simple example(3) – our Runnable class

```
import java.util.Date;

public class GreetingRunnable implements Runnable {

   private String greeting;
   private static final int REPETITIONS = 15;
   private static final int DELAY = 1000;

   public GreetingRunnable(String aGreeting) {
      greeting = aGreeting;
   }

   public void run() {
      try {
         for (int i = 1; i <= REPETITIONS; i++){
            Date now = new Date();
            System.out.println(now + " " + greeting);
            Thread.sleep((int)(DELAY*Math.random()));
         }
      } catch (InterruptedException exception){
      }
   }
}
```

If a thread is interrupted while it is sleeping, an InterruptedException is thrown.

Ball Animation

- A simplified version of the way BallWorlds does animation
- When balls are created, they are given position, velocity, and color
- Our `run()` method tells each of the balls to move, then redraws them
- Clicking the mouse turns movement off/on.
- Think about: could this application be written without creating the new thread?
- Demonstrate the program!

Set up the frame

```
public class AnimatedBallViewer {

   static final int FRAME_WIDTH = 600;
   static final int FRAME_HEIGHT = 500;

   public static void main(String[] args){
      JFrame frame = new JFrame();

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("BallAnimation");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      AnimatedBallComponent component = new AnimatedBallComponent();
      frame.add(component);

      frame.setVisible(true);
      new Thread(component).start();
   }
}
```

This class has all of the usual stuff, plus this last line of code that starts the animation.

The Ball class

```
class Ball {
   private double centerX, centerY, velX, velY;
   private Ellipse2D.Double ellipse;
   private Color color;
   private static final double radius = 15;

   public Ball(double cx, double cy, double vx, double vy, Color c){
      this.centerX = cx;
      this.centerY = cy;
      this.velX = vx;
      this.velY = vy;
      this.color = c;
      this.ellipse = new Ellipse2D.Double (
          this.centerX-radius, this.centerY-radius,
          2*radius, 2*radius);
   }

   public void fill (Graphics2D g2) {
      g2.setColor(this.color);
      g2.fill(ellipse);
   }

   public void move (){
      this.ellipse.x += this.velX;
      this.ellipse.y += this.velY;
   }
}
```

Everything here should look familiar, similar to code that you wrote for BallWorlds.

AnimatedBallComponent: Instance Variables and Constructor

```
public class AnimatedBallComponent extends JComponent
                    implements Runnable, MouseListener {

   private ArrayList<Ball> balls = new ArrayList<Ball>();
   private boolean moving = true;
   public static final long DELAY = 30;
   public static final int ITERATIONS = 300;

   public AnimatedBallComponent() {
      super();
      balls.add(new Ball(40, 50, 8, 5, Color.BLUE));
      balls.add(new Ball(500, 400, -3, -6, Color.RED));
      balls.add(new Ball(30, 300, 4, -3, Color.GREEN));
      this.addMouseListener(this);
   }
```

Again, there should be no surprises here!

AnimatedBallComponent: run, paintComponent, mousePressed

```
public void run() {
     for (int i=0; i<ITERATIONS; i++) {
        if (moving){
           for (Ball b:balls)
              b.move();
           this.repaint();
        }
        try {
           Thread.sleep(DELAY);
        } catch (InterruptedException e) {}
     }
  }

  public void paintComponent(Graphics g){
     Graphics2D g2 = (Graphics2D)g;
     for (Ball b:balls)
        b.fill(g2);
  }

  public void mousePressed (MouseEvent arg0) {
     moving = !moving;
  }
```

Each time through the loop (if moving), tell each ball to move, then repaint

Sleep for a while

Draw each ball

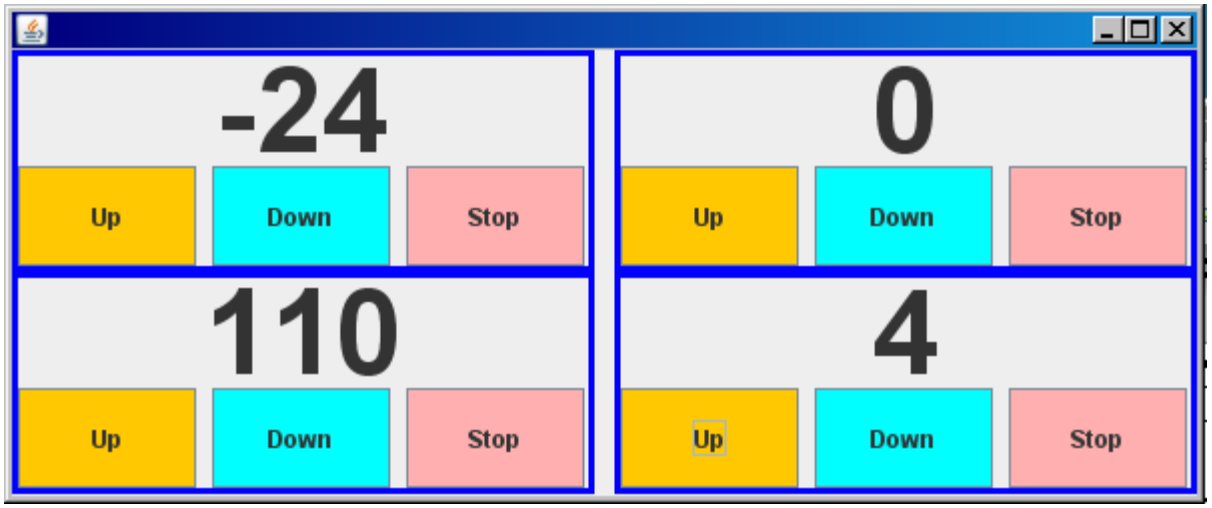
Toggle "moving" when the mouse is pressed

Is the thread necessary?

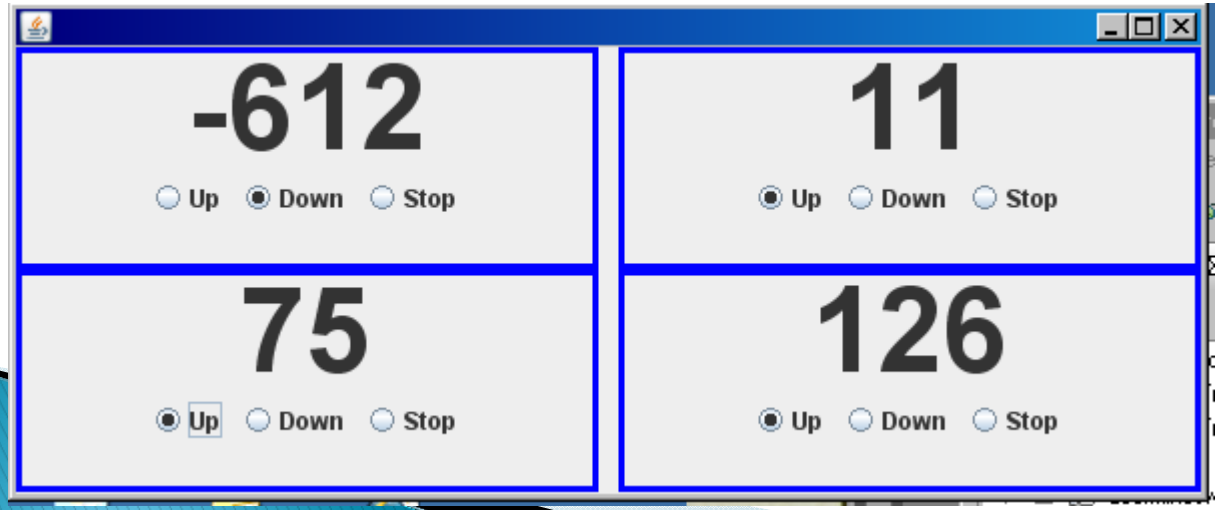
- Could this program have been written without creating the new thread?

Another animation: CounterThreads

- With regular buttons



With radio buttons



How many threads does this application appear to have?

CounterThreads setup

```
public class CounterThreads {

  public static void main (String []args) {
    JFrame win = new JFrame();
    Container c = win.getContentPane();
    win.setSize(600, 250);
    c.setLayout(new GridLayout(2, 2, 10, 0));
    c.add(new CounterPane(200));
    c.add(new CounterPane(500));
    c.add(new CounterPane(50)); // this one will count fast!
    c.add(new CounterPane(1000));

    win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    win.setVisible(true);
   }
}
```

Same old stuff!

CounterPane Basics

```
class CounterPane extends JComponent implements Runnable {

  private int delay;    // sleep time before changing counter
  private int direction = 0; //  current increment of counter
  private JLabel display = new JLabel("0");

  // Constants to define counting directions:
  private static final int COUNT_UP    =   1; // Declaring these
  private static final int COUNT_DOWN   = -1; // constants avoids
  private static final int COUNT_STILL  =  0; // "magic numbers"

  private static final int BORDER_WIDTH =  3;
  private static final int FONT_SIZE    = 60;
```

CounterPane Constructor

```
public CounterPane(int delay) {

    JButton upButton   = new JButton("Up");      // Note that these do
    JButton downButton = new JButton("Down");    // NOT have to be fields
    JButton stopButton = new JButton("Stop");    // of this class.

    this.delay = delay; // milliseconds to sleep

    this.setLayout(new GridLayout(2, 1, 5, 5));
       // top row for display, bottom for buttons.

    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(1, 3, 8, 1));
    display.setHorizontalAlignment(SwingConstants.CENTER);
    display.setFont(new Font(null, Font.BOLD, FONT_SIZE));
       // make the number display big!

    this.add(display);
    this.add(buttonPanel);
    this.setBorder(BorderFactory.createLineBorder(Color.blue,
                                                  BORDER_WIDTH));

    // Any Swing component can have a border.
    this.addButton(buttonPanel, upButton,   Color.orange, COUNT_UP);
    this.addButton(buttonPanel, downButton, Color.cyan,   COUNT_DOWN);
    this.addButton(buttonPanel, stopButton, Color.pink,   COUNT_STILL);

    Thread t = new Thread(this);
    t.start();
```

Put a simple border around the panel. There are also more complex border styles that you can use.

A lot of the repetitive work is done by the calls to **addButton()**.

CounterPane's addButton method

```
// Adds a control button to the panel, and creates an
// ActionListener that sets the count direction.
private void addButton(Container container,
                       JButton button,
                       Color color,
                       final int dir) {
  container.add(button);
  button.setBackground(color);
  button.addActionListener(new ActionListener () {
     public void actionPerformed(ActionEvent e) {
       direction = dir;
      }
  });
}
```

JPanel is a subclass of Container

The value of **dir** will be 1, –1, or 0, to indicate counting up, down, or neither.

- The action listener added here is an anonymous inner class that implements ActionListener.
- Because it is an inner class, its method can access this CounterPane's **dir** instance variable.

Note that each button gets its own ActionListener class, created at runtime. This is Swing's "preferred way" of providing ActionListeners.

CounterPane's run method

- This method is short and simple, because dir is always the amount to be added to the counter (1, –1, or 0).

```
public void run() {
    try {
      do {
        Thread.sleep(delay);
        display.setText(Integer.parseInt(display.getText())
                        + direction  + "");
      } while (true);
    } catch (InterruptedException e) { }
  }
}
```

CounterThreads questions

- Look through the code, discussing it with your partner and/or lab assistants until you think you understand it all. Answer the following questions:

1. How does a CounterPane know whether to count up or down or stay the same?
2. When a counter is not changing, does its thread use less CPU time than one that is changing?
3. Would it be easy to add code to the *main* method that creates a SuperStop button, so that clicking this button stops all counters? Explain.

RadioButton version

```
public CounterPaneRadio(int delay) {

    JRadioButton upButton   = new JRadioButton("Up");
    JRadioButton downButton = new JRadioButton("Down");
    JRadioButton stopButton = new JRadioButton("Stop");

    ButtonGroup group = new ButtonGroup();
    group.add(upButton);
    group.add(downButton);
    group.add(stopButton);
    stopButton.setSelected(true);
```

. . .

And we remove the **`Color`** parameter from **`addButton()`**

Ending a thread

- A thread t ends when its run method terminates.
- Threads used to have a stop method, but it is now deprecated.
- Instead of stopping a thread, you notify it that it should stop itself (return from its run method) by calling t.interrupt();
- The thread can check to see if it has been interrupted by calling this.isInterrupted();
- If so, the thread can decide to clean up and stop itself.

Section 02 ended here

Interlude

- Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

--Martin Golding

Program efficiency, part 2

- **Some simple efficiency tips**
  - If a statement in a loop calculates the same value each time through, move it outside the loop
  - Store and retain data on a “need to know” basis.
  - Don’t store what you won’t reuse!
    - Do store what you need to reuse!
  - Don’t put everything into an array when you only need one or two consecutive items at a time.
  - Don’t make a variable be a field when it can be a local variable of a method.

Familiar example:
Linear search of a sorted array of Comparable items

```
for (int i=0; i < a.length; i++)
  if ( a[i].compareTo(soughtItem) > 0 )
    return NOT_FOUND;
  else if ( a[i].compareTo(soughtItem) == 0 )
    return i;
return NOT_FOUND;
```

- What should we count?
- Best case, worst case, average case?

Work on Minesweeper

- Don't forget to commit your progress report to the repository before the end of class.
- Please turn in your in-class quiz now.