Principles of Software Construction: A Brief Introduction to Multithreading and GUI Programming

Josh Bloch  Charlie Garrod
Administrivia

- Homework 4b due next Thursday
- HW 3 feedback pushed this morning.
- HW 4a feedback available later this week
- Last day to register to vote is TODAY
Key concepts from Thursday...

• Class invariants must be maintained
  – Make defensive copies where required
• Immutable class have many advantages
• Testing is critical to software quality
  – When fixing bugs, write tests before code
  – Good tests have high power-to-weight ratio
Outline

• Multithreaded Programming basics
• GUI Programming
What is a thread?

• Short for *thread of execution*
• Multiple threads run in same program concurrently
• Threads share the same address space
  – Changes made by one thread may be read by others
• Multithreaded programming
  – Also known as shared-memory multiprocessing
Threads vs. processes

- Threads are lightweight; processes heavyweight
- Threads share address space; processes have own
- Threads require synchronization; processes don’t
  – Threads hold locks while mutating objects
- It’s unsafe to kill threads; safe to kill processes
Why use threads?

• Performance in the face of blocking activities
  – Consider a web server
• Performance on multiprocessors
• Cleanly dealing with natural concurrency
• In Java threads are a fact of life
  – Example: garbage collector runs in its own thread
Example: generating cryptarithms

```java
static List<String> cryptarithms(String[] words, int start, int end) {
    List<String> result = new ArrayList<>();
    String[] tokens = new String[] {"", "+", ",", ":", ""};

    for (int i = start; i < end - 2; i++) {
        tokens[0] = words[i];  tokens[2] = words[i + 1];
        tokens[4] = words[i + 2];
        try {
            Cryptarithm c = new Cryptarithm(tokens);
            if (c.solve().size() == 1)
                result.add(c.toString());
        } catch (IllegalArgumentException e) {
            // too many letters; ignore
        }
    }
    return result;
}
```
Single-threaded driver

```java
public static void main(String[] args) {
    long startTime = System.nanoTime();
    List<String> cryptarithms = cryptarithms(words, 0, words.length);
    long endTime = System.nanoTime();

    System.out.printf("Time: %d
", (endTime - startTime)/1e9);
    System.out.println(cryptarithms);
}
```
Multithreaded driver

public static void main(String[] args) throws InterruptedException {
    int n = Integer.parseInt(args[0]);  // Number of threads
    long startTime = System.nanoTime();
    int wordsPerThread = words.length / n;
    Thread[] threads = new Thread[n];
    Object[] results = new Object[4];
    for (int i = 0; i < n; i++) {  // Create the threads
        int start = i == 0 ? 0 : i * wordsPerThread - 2;
        int end = i == n-1 ? words.length : (i + 1) * wordsPerThread;
        int j = i;  // Only constants can be captured by lambdas
        threads[i] = new Thread(() -> {
            results[j] = cryptarithms(words, start, end);
        });
    }
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    long endTime = System.nanoTime();
    System.out.printf("Time: %d\ns", (endTime - startTime)/1e9);
    System.out.println(Arrays.toString(results));
}
Cryptarithm generation performance

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Seconds to run</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.0</td>
</tr>
<tr>
<td>2</td>
<td>13.5</td>
</tr>
<tr>
<td>3</td>
<td>11.7</td>
</tr>
<tr>
<td>4</td>
<td>10.8</td>
</tr>
</tbody>
</table>

Generating all cryptarithms from a corpus of 344 words
• Test all consecutive 3-word sequences (342 possibilities)
• Test machine is this crappy old laptop (2 cores, 4 hyperthreads)
• I did not follow benchmarking best practices!
What requires synchronization?

• Shared mutable state
• If not properly synchronized, all bests are off!
• You have three choices
  1. **Don’t mutate**: share only immutable state
  2. **Don’t share**: isolate mutable state in individual threads
  3. If you must share mutable state, **synchronize properly**
Synchronization is tricky

• Too little and you risk safety failure
  – Changes aren’t guaranteed to propagate thread to thread
  – Program can observe inconsistencies
  – Critical invariants can be corrupted

• Too much and program may run slowly or not at all
  – Deadlock or other liveness failure
Contention kills performance

- Synchronized is the opposite of concurrent!
- Highly concurrent code is possible to write
  - But it’s very difficult to get right
  - If you get it wrong you’re toast
- Let Doug Lea write it for you!
  - ConcurrentHashMap
  - Executor framework
  - See java.util.concurrent
Safety vs. liveness

• Safety failure – incorrect computation
  – Can be subtle or blatant
• Liveness failure – no computation at all
• Temptation to favor liveness over safety
  – Don’t succumb!
• Safety failures offer a false sense of security
• Liveness failures force you to confront the bug
Synchronization in cryptarithm example

• How did we avoid synchronization in our multithreaded cryptarithm generator?
• *Embarrassingly parallelizable computation*
• Each thread is entirely independent of the others
  – They try different cryptarithms
  – And write results to different array elements
• No shared mutable state to speak of
  – Main thread implicitly syncs with workers with `join`
Outline

• Multithreaded Programming
• GUI Programming
There are many Java GUI frameworks

• AWT – obsolete except as a part of Swing
• Swing – the most widely used, by far
• SWT – Little used outside of Eclipse
• JavaFX – Billed as a replacement for Swing
  – Released 2008 – has yet to gain traction
• A bunch of modern (web & mobile) frameworks
  – e.g., Android
GUI programming is multithreaded

• *Event-driven programming*
  
  • Event dispatch thread (EDT) handles all GUI events
    – Mouse events, keyboard events, timer events, etc.

• Program registers callbacks ("listeners")
  – Function objects invoked in response to events
  – Observer pattern
Ground rules for GUI programming

1. All GUI activity is on event dispatch thread
2. No other time-consuming activity on this thread
   – Blocking calls (e.g., I/O) absolutely forbidden

- Many GUI programs violate these rules
  – They are broken
- Violating rule 1 can cause safety failures
- Violating rule 2 can cause liveness failures
Ensuring all GUI activity is on EDT

- Never make a Swing call from any other thread
- Swing calls includes Swing constructors
- If not on EDT, make Swing calls with `invokeLater`:

```java
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new Test().setVisible(true));
}
```
Callbacks execute on the EDT

• You are a guest on the Event Dispatch Thread!
• Don’t abuse the privilege
• If you do, liveness will suffer
  – Your program will become non-responsive
  – Your users will become angry
• If > a few ms of work to do, do it off the EDT
  – javax.swing.SwingWorker designed for this purpose
DEMO – JDICE
Jdice – (a) DieType

/** A game die type. Can also be used as a stateless game die. */
public enum DieType {
    d4(4, 3), d6(6, 4), d8(8, 3), d10(10, 5), d12(12, 5), d20(20, 3);

    private final int sides; // Number of faces
    private final int edges; // Number of edges on each face

    DieType(int sides, int edges) {
        this.sides = sides;
        this.edges = edges;
    }

    public int sides() { return sides; }
    public int edges() { return edges; }

    private static final Random random = new Random();
    public int roll() { return random.nextInt(sides) + 1; }
    public int roll(Random rnd) { return rnd.nextInt(sides) + 1; }
}
/** A single, stateful game die. */
public class Die {
    private final DieType dieType;
    private int lastRoll = 1;

    public Die(DieType dieType) { this.dieType = dieType; }

    public DieType dieType() { return dieType; }

    public int roll() { return lastRoll = dieType.roll(); }

    public int roll(Random rnd) {
        return lastRoll = dieType.roll(rnd);
    }

    public int lastRoll() { return lastRoll; }
}
JDice – (b) Die (Part 2 of 2)

/** Returns array of Die per the std string spec (e.g., "d12", "2d6"). */
public static Die[] dice(String spec) {
    DieType dieType;
    int numDice;
    int dPos = spec.indexOf('d');
    if (dPos == 0) {
        numDice = 1;
        dieType = DieType.valueOf(spec);
    } else {
        numDice = Integer.parseInt(spec.substring(0, dPos));
        dieType = DieType.valueOf(spec.substring(dPos));
    }

    Die[] result = new Die[numDice];
    for (int i = 0; i < numDice; i++)
        result[i] = new Die(dieType);
    return result;
}
/** GUI game die component that provides a view on a Die. */
public class JDie extends JComponent {
    private Die die;
    public JDie(Die die) { this.die = die; }

    @Override protected void paintComponent(Graphics g) {
        super.paintComponent(g); // Boilerplate

        // Get our size from containing component and compute center point
        int componentWidth = getWidth();
        int componentHeight = getHeight();
        int centerX = componentWidth / 2;
        int centerY = componentHeight / 2;

        // Get Graphics 2D object - lets us do actual drawing
        Graphics2D g2d = (Graphics2D) g;
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                              RenderingHints.VALUE_ANTIALIAS_ON);
// Draw the face outline
g2d.setColor(Color.BLACK);
g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
double r = .4 * Math.min(componentWidth, componentHeight);
Path2D path = polygon(die.dieType().edges(), centerX, centerY, r);
g2d.draw(path);

// Fill the face outline
g2d.setColor(Color.RED);
g2d.fill(path);

// Draw the number on the face
g2d.setColor(Color.WHITE);
Font font = g2d.getFont();
g2d.setFont(font = font.deriveFont(Font.BOLD, 3 * font.getSize()));
String number = Integer.toString(die.lastRoll());
drawCenteredString(g2d, number, centerX, centerY);
helper method polygon

/** Returns a polygonal path with an edge parallel to X axis. */
static Path2D polygon(int edges, int ctrX, int ctrY, double r) {
   // Compute angle of first point in polygon path
   double theta0 = -Math.PI / 2;
   if ((edges & 1) == 0)
      theta0 += Math.PI / edges; // Even # of sides

   Path2D path = new Path2D.Double();
   for (int i = 0; i < edges; i++) {
      double theta = theta0 + i * 2 * Math.PI / edges;
      double x = ctrX + r * Math.cos(theta);
      double y = ctrY + r * Math.sin(theta);
      if (i == 0)
         path.moveTo(x, y);
      else
         path.lineTo(x, y);
   }
   path.closePath();
   return path;
}
/** Prints a string centered at the given point */
static void drawCenteredString(Graphics2D g2d, String text, int x, int y) {
    Rectangle stringBounds = g2d.getFontMetrics().getStringBounds(text, g2d).getBounds();
    GlyphVector glyphVector = g2d.getFont().createGlyphVector(g2d.getFontRenderContext(), text);
    Rectangle visualBounds = glyphVector.getVisualBounds().getBounds();

    g2d.drawString(text, x - stringBounds.width / 2,
                   y - visualBounds.height / 2 - visualBounds.y);
}
JDice - (d) JDice

/** GUI game dice panel that provides a view on a Die array. */
public class JDice extends JPanel {
    public JDice(Die[] dice) {
        setLayout(new GridLayout(1, dice.length, 5, 0));
        for (Die d : dice)
            add(new JDie(d));
    }

    public void resetDice(Die[] dice) {
        removeAll();
        for (Die d : dice)
            add(new JDie(d));
        revalidate(); // Required boilerplate
        repaint();
    }
}

public class Demo extends JFrame {
    String diceSpec = "2d6"; // Default dice spec.
    Die[] dice = Die.dice(diceSpec);
    JDice jDice = new JDice(dice);

    private Demo() {
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setSize(600, 300); // Default dimensions

        // Implement roll button and dice type field
        JTextField diceSpecField = new JTextField(diceSpec, 5); // Field width
        JButton rollButton = new JButton("Roll");
        rollButton.addActionListener(event -> { // Callback!
            if (!diceSpecField.getText().equals(diceSpec)) {
                diceSpec = diceSpecField.getText();
                dice = Die.dice(diceSpec);
                jDice.resetDice(dice);
            }
            for (Die d : dice) d.roll();
            jDice.repaint();
        });
    }
}
JDice – (e) Demo (Part 2 of 2)

// End of constructor: build roll panel and content pane
JPanel rollPanel = new JPanel(new FlowLayout());
rollPanel.add(diceSpecField);
rollPanel.add(rollButton);

getContentPane().add(jDice, BorderLayout.CENTER);
getContentPane().add(rollPanel, BorderLayout.SOUTH);

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new Demo().setVisible(true));
}
Observations on JDice

• GUI programming is a bit tedious
• The Swing APIs are huge
• And yet you still have to do a lot yourself
  – e.g., drawing polygons, centering text properly
  – Doing it well takes a lot of effort
• Getting the threading right isn’t that hard
  – So do it
For help writing Swing code

• Sun wrote a good tutorial
  – http://docs.oracle.com/javase/tutorial/uiswing/

• The many components shown with examples
  – http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html

• Listeners supported by each component
  – http://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html
Summary

- Multithreaded programming is genuinely hard
  - But it’s a fact of life in Java
- Neither under- nor over-synchronize
  - Immutable types are your best friend
  - `java.util.concurrent` is your next-best friend
- GUI programming is limited form of multithreading
  - Swing calls *must* be made on event dispatch thread
  - No other significant work should be done on EDT