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

Josh Bloch Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due **next Thursday**
- HW 4a feedback available later this week
- A few midterms still available

Key concepts from Thursday…

- Testing is critical to software quality
- When fixing bugs, write tests *before* code
- Good tests have high power-to-weight ratio
- In complex environments, design for testability
  - Enable *mocking* with factories
  - Avoid static singletons
  - Consider dependency injection and mocking tools
- Static analysis helps too

Outline

- Multithreaded Programming
- 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

```
private static 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 (RuntimeException e) {
            // too many letters; ignore
        }
    }
    return result.toArray(new String[result.size()]);
}
```

Single-threaded driver

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

    System.out.println("Time: " + ((endTime - startTime)/1e9) + "s.");
    System.out.println(Arrays.toString(cryptarithms));
}
```

Multithreaded driver

```
public static void main(String[] args) throws InterruptedException {
    int n = Integer.parseInt(args[0]);
    long startTime = System.nanoTime();
    int wordsPerThread = words.length / n;
    Thread[] threads = new Thread[n];
    String[][] results = new String[n][];
    for (int i = 0; i < n; i++) {
        int start = i == 0 ? 0 : i * wordsPerThread - 2;
        int end = i == n-1 ? words.length : (i + 1) * wordsPerThread;
        int m = i; // Only constants can be captured by lambdas
        threads[i] = new Thread(() ->
            { results[m] = cryptarithms(words, start, end); });
    }
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    long endTime = System.nanoTime();
    System.out.println("Time: " + ((endTime - startTime)/1e9) + "s.");
    System.out.println(Arrays.deepToString(results));
}
```

Cryptarithm generation performance

Number of Threads	Seconds to run
1	22.0
2	13.5
3	11.7
4	10.8

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 your program may not run 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 cryptarithms

- 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 arrays
- No shared mutable state to speak of
  - Main thread implicitly syncs with workers - `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

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., IO) 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 constructors
- If not on EDT, make a swing call with `invokeLater`:

```
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 – 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; }
}
```

JDice – Die (Part 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 lastRoll() { return lastRoll; }
```

JDice – Die (Part 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) {
        dieType = DieType.valueOf(spec);
        numDice = 1;
    } else {
        numDice = Integer.valueOf(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;
}
```

JDice – Jdie (Part 1)

```
/** 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
        int componentWidth = getWidth();
        int componentHeight = getHeight();
        int boxSize = Math.min(componentWidth, componentHeight);
        double r = boxSize * .4; // Radius of circle polygon inscribed in
```

JDice – Jdie (Part 2) paintComponent, cont.

```
// Compute center of polygon, and angle of first point
int centerX = componentWidth / 2;
int centerY = componentHeight / 2;
double theta0 = -Math.PI / 2;
int edges = die.dieType().edges();
if ((edges & 1) == 0) theta0 += Math.PI / edges; // Even number of sides

// "Draw" polygon
Path2D path = new Path2D.Double();
for (int i = 0; i < edges; i++) {
    double theta = theta0 + i * 2 * Math.PI / edges;
    double x = centerX + r * Math.cos(theta);
    double y = centerY + r * Math.sin(theta);
    if (i == 0)
        path.moveTo(x, y);
    else
        path.lineTo(x, y);
}
path.closePath();
```

JDice – Jdie (Part 3) paintComponent, cont.

```
// Get Graphics 2D object - lets us do actual drawing
Graphics2D g2d = (Graphics2D) g;
g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                     RenderingHints.VALUE_ANTIALIAS_ON);
// Draw the polygon
g2d.setColor(Color.BLACK);
g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
   BasicStroke.JOIN_ROUND));
g2d.draw(path);

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

// Draw number on die face
g2d.setColor(Color.WHITE);
Font f = g2d.getFont();
float fSize = f.getSize();
g2d.setFont(f.deriveFont(Font.BOLD, 3 * fSize));
g2d.drawString(Integer.toString(die.lastRoll()),
               centerX - fSize/2, centerY + .75f * fSize);
```

JDice – 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();
    }
}
```

JDice – Demo (part 1)

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

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

JDice – Demo (Constructor part 2)

```
// 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 – Demo (Constructor part 3 and main)

```
        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., the polygonal faces in JDice
- Doing it well takes a lot of effort
  - Numbers are not properly centered on die face
- 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