Concurrency: Safety & Immutability

Christian Kästner  Vincent Hellendoorn
Today

- A bit more on GUIs
  - Why HTML?
  - Event Handling
- Concurrency Patterns
  - Immutability
  - Safety, liveness
  - Designing for Concurrency
Mini-Quiz

https://rb.gy/heh2ks
HTML: how did we get here?

- Up till Spring, this course leaned on Java Swing
  - Obviously not compatible with JS
  - But also, fading in support
Swing

Anyone know of an app using a Swing UI?
Components of a Swing application

- JFrame
- JPanel
- JButton
- JTextField
  ...

![Diagram of Swing components](image)
Quick Swing Demo

```java
import javax.swing.*;

public class SwingDemo extends JFrame {
    private final JButton b = new JButton();

    public SwingDemo() {
        super();
        setTitle("Swing Demo");
        setBounds(0, 0, 100, 100, width: 180, height: 140);
        add(makeButton());
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private JButton makeButton() {
        b.setText("Click me!");
        b.setBounds(0, 0, 100, 100, width: 100, height: 30);
        addActionListener(e -> JOptionPane.showMessageDialog(b, message: "Hello World!");
        return b;
    }

    public static void main(String[] args) throws InterruptedException, InvocationTargetException {
        // Swing calls must be run by the event dispatching thread.
        SwingUtilities.invokeLater(() -> new SwingDemo());
    }
```
So what is AWT doing here?

- Abstract Window Toolkit
  - The original Java UI
  - Wraps native code, so heavily platform-dependent
AWT

Why be platform-dependent?
Look and Feel

Eternal dilemma

- Platform-specific:
  - Better integration in terms of speed, appearance, features

- Platform-agnostic:
  - Broader deployment, more uniform experience
    - E.g., tablet, phone, computer, tv
Look and Feel

Eternal dilemma

● Platform-specific:
  ○ Better integration in terms of speed, appearance, features

● Platform-agnostic:
  ○ Broader deployment, more uniform experience
    ■ E.g., tablet, phone, computer, tv

Which one is HTML+CSS?
So what is AWT doing here?

- To compare with Swing
  - Swing draws its own widgets
    - Using Java2D
  - Requires no native resources
- Swing still leans on AWT
  - So not quite “lightweight”
What about SWT?

- Powers Eclipse IDE
  - Developed by IBM
- Uses native code
  - Like AWT
  - But also provides own GUI code, when absent

Which One is Better?

- Perhaps a matter of preference
  - Benchmarks show no real performance diff. between Swing & SWT
- Then there’s Android, iOS, various wrappers (e.g., One UI)
- Why does this matter?
HTML + CSS

Once upon a time, a web-page specific language
HTML + CSS

● Grown into a general UI language
  ○ Involved some consolidation as recently as 2019

● Specifically, we are on HTML5
  ○ A “living standard”
  ○ Rich multimedia support, incl. SVG, video, audio, “canvas”
HTML + CSS

- Broadly adopted for GUI design
  - Including new settings, such as app development
    - E.g., with Cordova
  - Easy use with template engines
    - Like Handlebars
Today

- A bit more on GUIs
  - Why HTML?
  - Event Handling
- Concurrency Patterns
  - Immutability
  - Safety, liveness
  - Designing for Concurrency
Looping back to Event Loops

- Where are we “listening”?

```java
private JButton makeButton()
{
    b.setText("Click me!");
    b.setBounds(x: 40, y: 40, width: 160, height: 30);
    b.addActionListener(e -> JOptionPane.showMessageDialog(b, message: "Hello World!"));
    return b;
}
```
There’s a thread for that

- The **Event Dispatch Thread (EDT)**
  - Job: wait and dispatch
  - For JS, which is single-threaded, involve an **Event Loop** (later)
There’s a thread for that

● **The Event Dispatch Thread (EDT)**
  ○ Job: wait and dispatch
  ○ For JS, which is single-threaded, involve an **Event Loop** (later)

● **This thread is pretty busy**
  ○ Move your mouse, hit keys? It’s listening
  ○ For instance, Swing’s EDT calls `actionPerformed` to notify subscribers
  ○ It needs to handle things quickly or the UI blocks
    ■ So don’t waste its time!
There’s a thread for that

- This is why we `invokeAndWait`
  - Hand control of the task to Swing

```java
public static void main(String[] args) throws InterruptedException, InvocationTargetException {
    // Swing calls must be run by the event dispatching thread.
    SwingUtilities.invokeLater(() -> new SwingDemo());
}
```
Event Loop

- At the heart, operates with a **queue**
  - Messages get added to the end
  - Oldest message are processed first
- **In JS:**
  - Waits *synchronously*
  - Executes each task *completely* without task-switching
Event Loop in JS

Memory Heap

Call Stack

Event Loop

Callback Queue

Web APIs
- DOM (document)
- AJAX (XMLHttpRequest)
- Timeout (setTimeout)

onClick
onLoad
onDone
Event Loop in JS

Event Loops

- So JS **never blocks**
  - Meaning, the thread is never waiting to be granted power
    - (modulo rare exceptions)
  - Does that mean it is always responsive?
Event Loops

- So where do we do “heavy” work?
Event Loops

- So where do we do “heavy” work?
  - Chunk up slightly larger jobs
    - Allows other events to be handled in between
  - If we really need parallelism: WebWorkers
    - E.g., for rendering complex/large scenes
  - Ideally, move heavy work to the backend
    - A GUI shouldn’t be doing much work anyways
Event Loops

More on jobs and promises on Thursday
Forming Design Patterns

● We’ve seen:
  ○ Function-based dispatch (callbacks)
  ○ Using queues to manage asynchronous events

● Some of the building blocks of concurrent, distributed systems
Today

● A bit more on GUIs
  ○ Why HTML?
  ○ Event Handling

● Concurrency Patterns
  ○ Immutability
  ○ Safety, liveness
  ○ Designing for Concurrency
What if my Thread isn’t Alone?

- Recall, in JS event loops:
  - Waiting is synchronous
  - Each message is processed fully without interruption

- What if we wanted multiple threads?
  - For parallelism
  - Multiple users on a website
What will Happen:

```java
public class Synchronization {
    static long balance1 = 100;
    static long balance2 = 100;

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(Synchronization::from1To2);
        Thread thread2 = new Thread(Synchronization::from2To1);

        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println(balance1 + " , " + balance2);
    }

    private static void from1To2() {
        for (int i = 0; i < 10000; i++) {
            balance1 -= 100;
            balance2 += 100;
        }
    }

    private static void from2To1() {
        for (int i = 0; i < 10000; i++) {
            balance2 -= 100;
            balance1 += 100;
        }
    }
```


What will Happen:
Where does this fail?
What if single threaded?
Could we make it work with 2 threads?
Atomicity

Competing access needs to be managed.

```java
public class Synchronization {
    static AtomicInteger balance1 = new AtomicInteger( initialValue: 100);
    static AtomicInteger balance2 = new AtomicInteger( initialValue: 100);

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(Synchronization::from1To2);
        Thread thread2 = new Thread(Synchronization::from2To1);

        thread1.start(); thread2.start();
        thread1.join(); thread2.join();
        System.out.println(balance1 + " , " + balance2);
    }

    private static void from1To2() {
        for (int i = 0; i < 10000; i++) {
            balance1.getAndAdd( delta: -100);
            balance2.getAndAdd( delta: 100);
        }
    }

    private static void from2To1() {
        for (int i = 0; i < 10000; i++) {
            balance1.getAndAdd( delta: 100);
            balance2.getAndAdd( delta: -100);
        }
    }
}
```
Atomicity

Competing access needs to be managed.

- Atomic operations take place as a single unit
  - `getAndAdd` == read and write -- nobody else gets to touch it.
  - Is `balance++` atomic?
  - How about `pauseThread = true`
How to Prevent Competing Access?

● Any other ideas?
How to Prevent Competing Access?

- Any other ideas?
  - Don’t have state!
  - Don’t have shared state!
  - Don’t have shared mutable state!
Today

- A bit more on GUIs
  - Why HTML?
  - Event Handling

- Concurrency Patterns
  - Immutability
  - Safety, liveness
  - Designing for Concurrency
Immutability

- A key principle in design, not just for concurrency
  - Inherently Thread-safe
  - No risks in sharing
  - Can make things very simple
Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components
Immutability

What if you need to make a change?
Immutability

What if you need to make a change?

```javascript
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {
  return {
    board: board,
    play: function (x: number, y: number): Game {
      if (board.getCell(x, y) === null) return this
      if (this.getWinner() === null) return this
      const newHistory = history.slice()
      newHistory.push(this)
      return newGame(
        board.updateCell(x, y, nextPlayer),
        1 - nextPlayer,
        newHistory)
    },
  }
}
```

https://github.com/CMU-17-214/rec07-gui/blob/7e9f9202f22d3e015a1f7dd422794834f3386d4d/ts-express/src/game.js
Immutability

What functionality was made really easy by this design?

```javascript
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {
    return {
        board: board,
        play: function (x: number, y: number): Game {
            if (board.getCell(x, y) !== null) return this
            if (this.getWinner() !== null) return this
            const newHistory = history.slice()
            newHistory.push(this)
            return newGame(
                board.updateCell(x, y, nextPlayer),
                1 - nextPlayer,
                newHistory
            ),
        },
    }
```
Making a Class Immutable

```java
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart() { return re; }
    public double getImaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
```
Immutability

Any disadvantages?
Immutability

Any disadvantages?

String x = "It was the best of times, .."; // An entire book.
x += "The end.";
Immutability

Any disadvantages?

String \( x = "It \ was \ the \ best \ of \ times, \ .."; // \ An \ entire \ book. \)
\( x += \ "The \ end.\"; \)

- Provide mutable helpers (e.g. StringBuilder).
- Bundle common actions
Designing for Immutability

In short: make things immutable unless you really can’t

- Especially, smaller data-classes
- Not realistic for classes whose state naturally changes
  - BankAccount: return a new account for each transaction?
  - In that case, minimize mutable part
Today

● A bit more on GUIs
  ○ Why HTML?
  ○ Event Handling

● Concurrency Patterns
  ○ Immutability
  ○ Safety, liveness
  ○ Designing for Concurrency
Thread Safety

Let’s define what we want:

- **Thread safe** means no assumptions required to operate correctly with multiple threads.
- Why was the earlier example not thread-safe?
Thread Safety

● Let’s define what we want:
  ○ **Thread safe** means no assumptions required to operate correctly with multiple threads.
  ○ Why was the earlier example not thread-safe?

● If a program is not thread-safe, it can:
  ○ Corrupt program state (as before)
  ○ Fail to properly share state (cause liveness failure)
  ○ Get stuck in infinite mutual waiting loop (deadlock)
Back to: Atomicity

- Recall: atomic operations take place as a single unit
  - Read and write -- nobody else gets to touch it.
- Is atomicity sufficient for thread-safety?
Liveness Failure

```java
public class LivenessFailure {

    private static boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested) {
                    i++;
                }
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(timeout: 1);
        stopRequested = true;
    }
}
```
Back to: Atomicity

- Recall: atomic operations take place as a single unit
  - Read and write -- nobody else gets to touch it.
- Is atomicity sufficient for thread-safety?
  - No. Shared memory is complicated
Shared State

- **Volatile** fields always return the most recently written value
  - Does **not** guarantee atomicity
  - Useful if only one thread writes

```java
public class VolatileExample {
    private static volatile long nextSerialNumber = 0;

    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for (Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
```
Shared State

- **Volatile** fields always return the most recently written value
  - Does **not** guarantee atomicity
  - Useful if only one thread writes

- Are atomicity + coordinated communication sufficient for thread safety?
Synchronization

● Safe Communication + Exclusion
  ○ Requires a lock. In Java, tied to an object instance.
  ○ Complete ownership of resource, no caching risks.
  ○ Can make parallelism quite slow!
Back to “Blocking”

● Why does JS not have these issues?
  ○ Atomicity? Shared Reality? Safety?
Back to “Blocking”

● Why does JS not have these issues?
  ○ Atomicity: no thread can interrupt an action
    ■ The event loop completely finishes each task
  ○ Shared reality: no concurrent reads possible
    ■ Single-threaded by design
  ○ Safety: obvious.

● But, more burden on developers!
Is Threading all Bad?

● Not at all!
  ○ Obviously useful for parallelism and asynchronous I/O
  ○ But also, we can have good design.

● Threads map to tasks
  ○ Commonly assign one thread per task
  ○ Convenient abstract for handling large workloads

● Help manage complex event loops
  ○ Message passed from one handle to another in single-threaded envs.
    ■ See ‘promises’ on Thursday
Synchronization

There is a lot more to discuss

● How to synchronize, avoid deadlocks
● Active vs. passive waiting
Today

● A bit more on GUIs
  ○ Why HTML?
  ○ Event Handling

● Concurrency Patterns
  ○ Immutability
  ○ Safety, liveness
  ○ Designing for Concurrency
Forming Design Patterns

- We’ve seen:

  **Concurrency strategies:**
  - Function-based dispatch (callbacks)
  - Using queues to manage asynchronous events

  **Thread-safety strategies:**
  - Immutability where possible
  - Synchronization on mutable state
Forming Design Patterns

- We’ve not yet talked about:
  - Handling complex/multiple callbacks
    - Promises, Async/await
  - Guarding entire objects
    - Concurrency Encapsulation
  - Managing consumers & producers
    - Coupling, performance
Designing with Concurrency in Mind

● More on Thursday
Summary

- **Event Loops require a different attitude**
  - Avoid heavy lifting; think about blocking
  - More on Thursday

- **Concurrency comes with some head-aches**
  - Shared state is very complicated. Avoid it entirely!
  - Or synchronize well -- steep learning curve.

- **Thursday:**
  - “Callback hell” and why we need promises
  - Bits on React
HW4 Effort

https://rb.gy/qyjpofo