Principles of Software Construction: Objects, Design, and Concurrency

Introduction to concurrency and GUls

Charlie Garrod  Josh Bloch
Administrivia

• Reading due Tuesday: UML and Patterns 26.1 and 26.4
• Homework 4a due tonight
  – Homework 4a feedback coming next week
• Homework 4b due Thursday, March 5th
  – An aside: testing
Key concepts from Tuesday

- Internal representations matter
- Good code is clean and concise
- Good coding habits matter
Key concepts from yesterday's recitation

• Discovering design patterns
• Observer design pattern
Observer pattern (a.k.a. publish/subscribe)

- **Problem:** Must notify other objects (observers) without becoming dependent on the objects receiving the notification
- **Solution:** Define a small interface to define how observers receive a notification, and only depend on the interface
- **Consequences:**
  - Loose coupling between observers and the source of the notifications
  - Notifications can cause a cascade effect

See edu.cmu.cs.cs214.rec06.alarmclock.AlarmListener...
Today

• The observer pattern
• Introduction to concurrency
• Introduction to GUIs
A thread is a thread of execution

• Multiple threads in the same program concurrently
• Threads share the same memory 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 are heavyweight
- Threads share address space; processes don't
- Threads require synchronization; processes don't
- It's unsafe to kill threads; safe to kill processes
Reasons to use threads

• Performance needed for blocking activities
• Performance on multi-core processors
• Natural concurrency in the real-world
• Existing multi-threaded, managed run-time environments
A simple threads example

```java
public interface Runnable {
    // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };

    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```
A simple threads example

```java
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = () -> System.out.println("Hi mom!");
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```
A simple threads example

```java
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    for (int i = 0; i < n; i++) {
        new Thread(() -> System.out.println("Hi mom!")).start();
    }
}
```
Aside: Anonymous inner class scope in Java

```java
public interface Runnable { // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]); // Number of threads;

    for (int i = 0; i < n; i++) {
        new Thread(() -> System.out.println("T" + i)).start();
    }
}
```

won't compile because `i` mutates
Aside: Anonymous inner class scope in Java

```java
public interface Runnable { // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]); // Number of threads;

    for (int i = 0; i < n; i++) {
        int j = i; // j unchanging within each loop
        new Thread(() -> System.out.println("T" + j)).start();
    }
}

j is effectively final
```
Example: generating cryptarithms

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

    // Check if each adjacent triple in words is a "good" cryptarithm
    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()); // We found a "good" one
        } catch (IllegalArgumentException e) {
            // too many letters in cryptarithm; ignore
        }
    }

    return result;
}
```
Single-threaded driver

```java
public static void main(String[] args) {
    Instant start = Instant.now();
    List<String> cryptarithms = cryptarithms(words, 0, words.length);
    Instant end = Instant.now();

    Duration time = Duration.between(start, end);
    System.out.printf("Time: %d%n ms", time.toMillis());
    System.out.println(cryptarithms);
}
```
Multithreaded driver

```java
public static void main(String[] args) throws InterruptedException {
    int n = Integer.parseInt(args[0]); // Number of threads
    Instant startTime = Instant.now();
    int wordsPerThread = words.length / n;
    Thread[] threads = new Thread[n];
    Object[] results = new Object[n];
    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();
    Instant endTime = Instant.now();

    Duration time = Duration.between(startTime, endTime);
    System.out.printf("Time: %d\n ms", time.toMillis());
    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 crappy old laptop (2 cores, 4 hyperthreads)
• These numbers are at-best approximate
Shared mutable state requires synchronization

- Three basic 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
The challenge of synchronization

• Not enough synchronization: *safety failure*
  – Incorrect computation

• Too much synchronization: *liveness failure*
  – Possibly: No computation at all
Synchronization in the cryptarithm example

• How did we avoid sync in multithreaded cryptarithm generator?
• *Embarrassingly parallelizable computation*
• Each thread is entirely independent of the others
  – They solve different cryptarithms...
  – And write results to different array elements
• No shared mutable state to speak of
  – Main thread implicitly synchronizes with workers using `join`
Today

- The observer pattern
- Introduction to concurrency
- Introduction to GUIs
Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```java
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```
Examples of events in GUIs

- User clicks a button, presses a key
- User selects an item from a list, an item from a menu
- Mouse hovers over a widget, focus changes
- Scrolling, mouse wheel turned
- Resizing a window, hiding a window
- Drag and drop

- A packet arrives from a web service, connection drops, ...
- System shutdown, ...
Blocking interaction with command-line interfaces

```java
Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = question.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
```
Blocking interactions with users

Diagram:
- Game
- Dealer
- Player

Activities:
- newGame
- addCards
- addCards
- getAction
- action
- [action==hit] addCard

Note: Blocking execution
Interactions with users through events

- Do not block waiting for user response
- Instead, react to user events
GUIs: To be continued...
Paper slides from lecture are scanned below..