Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency

Christian Kästner  Vincent Hellendoorn
How was the Recitation?

- Did every solution make the program smaller?
- Did I change everything you would have?
  - Anything you wouldn’t?
Interaction with CLI

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);
}
A backend with no interaction

One Possible Domain Model

This is not a reference solution, it's an example of what a domain model looks like.
What have we not yet seen?
How do you **wait**?

```plaintext
while (true) {
    if (isKeyDown("Alt+Q")
        break;
    if (isKeyDown("F1")
        openHelp();
    if (isMouseDown(10 ...)
        startMovingWindow();
    ...
}
How do you multi-player?

```javascript
while (true) {
  if (player === "player1") {
    hasWon = play("player1");
    if (hasWon) break;
    player = "player2";
  } else if (player === "player2") {
    hasWon = play("player2");
    if (hasWon) break;
    player = "player1";
  }
}
```

https://www.cloudsavvyit.com/2586/how-to-build-your-multiplayer-games-server-architecture/
Today

Beyond serial execution

- Event-based Programming
- Asynchrony & Concurrency
- I/O, GUIs
- Observer Pattern
- React preview
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)
}
```
Event-based GUIs

```java
//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email = emailField.getText();
emaillist.add(email);

//on remove-button click:
int pos = emaillist.getSelectedItem();
if (pos>=0) emaillist.delete(pos);
```
(Blocking) Interactions with users

Game

Dealer

Player

newGame

addCards

addCards

getAction

action

[action==hit] addCard

blocking execution
Interactions with users through events

- Do not block waiting for user response
- Instead, react to user events

![Diagram showing interactions between Game, Dealer, and Player]

- User initiates a new game
- Dealer adds cards
- Player hits
- Dealer adds card
Three Concepts of Importance

- Thread: instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.
Three Concepts of Importance

- **Thread:** instructions executed in sequence
  - Within a thread, everything happens in order.
  - A thread can start, sleep, and die.
  - You often work on the “main” thread.

- **Concurrency:** multiple threads running at the same time
  - Not necessarily *executing* in parallel
Three Concepts of Importance

● Thread: instructions executed in sequence
  ○ Within a thread, everything happens in order.
  ○ A thread can start, sleep, and die.
  ○ You often work on the “main” thread.

● Concurrency: multiple threads running at the same time
  ○ Not necessarily executing in parallel

● Asynchrony: computation happening outside the main flow
Multi-Threading

The natural response to non-serial computation

- Multiple threads can exist concurrently
- Threads share memory space
- You are already using it
  - Garbage collection in the JVM
Asynchrony

Where might this come from?
Asynchrony

Where might this come from?

- People
- Other machines
- Our own *callbacks*
Asynchrony

*Usually*, managing asynchronous events involves concurrency

- Do something while we wait
- Multiple events can overlap
- Even “waiting” is not really doing nothing
- We will focus on constructs for handling both
Asynchrony

Asynchronous but not concurrent

```java
//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);
// And now, wait.
```
Where do we want concurrency?
Where do we want concurrency?

- User interfaces
  - Events can arrive any time
- File I/O
  - Offload work to disk/network/... handler
Where do we want concurrency?

- **Background work**
  - Periodically run garbage collection, check health of service

- **High-performance computing**
  - Facilitate parallelism and distributed computing
User Interfaces

What happens here:

document.addEventListener('click', () => console.log('Clicked!'))
User Interfaces

Callback functions

- Perhaps *the* building blocks of the internet’s UI.
- Work that should be done once something happens
  - Called asynchronously from the literal flow of the code
  - Not concurrent: JS is single-threaded

```javascript
document.addEventListener('click', () => {
    console.log('Clicked!');
    console.log('Clicked again!');
})
```
Concurreny with file I/O

Key chart:

<table>
<thead>
<tr>
<th>Computer Action</th>
<th>Avg Latency</th>
<th>Normalized Human Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GhzCPU Clock cycle 3Ghz</td>
<td>0.3 ns</td>
<td>1 s</td>
</tr>
<tr>
<td>Level 1 cache access</td>
<td>0.9 ns</td>
<td>3 s</td>
</tr>
<tr>
<td>Level 2 cache access</td>
<td>2.8 ns</td>
<td>9 s</td>
</tr>
<tr>
<td>Level 3 cache access</td>
<td>12.9 ns</td>
<td>43 s</td>
</tr>
<tr>
<td>RAM access</td>
<td>70 - 100ns</td>
<td>3.5 to 5.5 min</td>
</tr>
<tr>
<td>NVMe SSD I/O</td>
<td>7-150 μs</td>
<td>2 hrs to 2 days</td>
</tr>
<tr>
<td>Rotational disk I/O</td>
<td>1-10 ms</td>
<td>11 days to 4 mos</td>
</tr>
<tr>
<td>Internet: SF to NYC</td>
<td>40 ms</td>
<td>1.2 years</td>
</tr>
<tr>
<td>Internet: SF to Australia</td>
<td>183 ms</td>
<td>6 years</td>
</tr>
<tr>
<td>OS virtualization reboot</td>
<td>4 s</td>
<td>127 years</td>
</tr>
<tr>
<td>Virtualization reboot</td>
<td>40 s</td>
<td>1200 years</td>
</tr>
<tr>
<td>Physical system reboot</td>
<td>90 s</td>
<td>3 Millenia</td>
</tr>
</tbody>
</table>

Table 1: Computer Time in Human Terms
Concurrent with file I/O

Mostly used synchronous IO so far

```javascript
/**
 * in the top-level directory only look for subdirectories and metadata files
 */

processProject (builder: ProjectBuilder, dir: string): void {
  const files = fs.readdirSync(dir)
  for (const filename of files) {
    const file = path.join(dir, filename)
    const fileStats = fs.statSync(file)
    const extension = path.extname(file)
    if (fileStats.isDirectory()) { this.#processDirectory(builder, file) } else if (extension === '.yml') { this.#loadMetadataFile(builder, file) }
  }
}
```
Concurrency with file I/O

Mostly used synchronous IO so far

- Works fine if ‘fetch’ is synchronous
  - But if other work is waiting...

```javascript
let image: Image = fetch('myImage.png');
display(image);
```
Concurrency with file I/O

Mostly used **synchronous** IO so far

- Works fine if ‘fetch’ is synchronous
  - But if other work is waiting...

```javascript
let image: Image = fetch('myImage.png');
display(image);
```

- It’d be nice if we could continue other work
  - How to make it work if ‘fetch’ is asynchronous?
Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
  - Neither fetched, nor failed; we’ll find out eventually

```javascript
let imageToBe: Promise<Image> = fetch('myImage.png');
imageToBe.then((image) => display(image))
  .catch((err) => console.log('aw: ' + err));
```
Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
  - Neither fetched, nor failed; we’ll find out eventually

```javascript
let imageToBe: Promise<Image> = fetch('myImage.png');
imageToBe.then((image) => display(image))
  .catch((err) => console.log('aw: ' + err));
```

- A bit like a callback
  - But better designed
  - Also related to async/await
  - Future in Java
Concurrency with file I/O

Can save you a lot of time

● An example from Machine Learning
● The usual process:
  ○ Read data from a filesystem or network
  ○ Batch samples, send to GPU/TPU/XPU memory
  ○ Train on-device
Concurrency with file I/O

An example from Machine Learning

Different devices:
Aside: Concurrency vs. parallelism

- Concurrency without parallelism:
  - Thread1
  - Thread2
  - Thread3

- Concurrency with parallelism:
  - Thread1
  - Thread2
  - Thread3
Aside: 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
Concurrency

Quite a few advanced topics

- Synchronization
- Immutability
- Parallelism
- More later in the course
  - Except for parallelism; largely out of scope
Designing for Asynchrony & Concurrency

- We are in a new paradigm now
  - We need standardized ways to handle asynchronous and/or concurrent interactions
  - This is how design patterns are born
- A lot of powerful syntax for managing concurrency
  - To be discussed in future classes
A GUI design challenge

- Consider a blackjack game, implemented by a Game class:
  - Player clicks “hit” and expects a new card
  - When should the GUI update the screen?
A GUI design challenge, extended

- What if we want to show the points won?

![Diagram showing a sequence of interactions between Game, GUI, and PointsPanel.]
Game updates GUI?

- What if points change for reasons not started by the GUI? (or computations take a long time and should not block)
Game updates GUI?

- Let the Game tell the GUI that something happened.
Game updates GUI?

- Let the Game tell the GUI that something happened

Problem: This couples the World to the GUI implementation.
Recall the Observer

https://refactoring.guru/design-patterns/observer
Decoupling with the Observer pattern

- Let the Game tell all interested components about updates
Core implementation vs. GUI

- Core implementation: application logic
  - Computing some result, updating data

- GUI
  - Graphical representation of data
  - Source of user interactions

- Design guideline: avoid coupling the GUI with core application
  - Multiple UIs with single core implementation
  - Test core without UI
Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)
An architectural pattern: Model-View-Controller (MVC)

- Manage inputs from user: mouse, keyboard, menu, etc.
- Manage display of information on the screen
- Manage data related to the application domain
Model-View-Controller (MVC)

Passive model

Active model

React Preview

How to handle asynchronous **streams** of data, across many actors?

- Without overwhelming workers
- Or blocking, or wasting resources
React Preview

“ReactiveX combines the **Observer pattern** with the **Iterator pattern** and **functional programming with collections** to fill the need for an ideal way of managing sequences of events.”  [https://rxjs.dev/guide/overview](https://rxjs.dev/guide/overview)

“It extends the **observer pattern** to support sequences of data/events and adds operators that allow you to **compose** sequences together declaratively while abstracting away concerns about things like **low-level threading, synchronization, thread-safety and concurrent data structures**.”  [https://github.com/ReactiveX/RxJava](https://github.com/ReactiveX/RxJava)
Summary

- Thinking past the main loop
  - The world is asynchronous
  - Concurrency helps, in a lot of ways
  - Requires revisiting programming patterns

- Start considering UI design
  - Discussed in more detail next week