WEEK10A

JAVA II — BARRY

TOPICS

- Project Q&A
- Proj07 Read-through
- Stacks & Queues
- Data Mocking
- Using Timers

PROJ07 READ-THROUGH

OUR LAST PROJECT FOR THE QUARTER

PROJ07 BUILDING BLOCKS

- Queues (built on your linked list, not Java's)
- Timers
- Mocked data
- ArrayList (yours, not Java's)

STACKS & QUEUES

TWO COMMON, USEFUL DATA STRUCTURES BUILT ON LINKED LISTS

STACK & QUEUE BASICS

Stacks and **Queues** are linear data structures storing an ordered sequence of values; they can be easily implemented with linked lists

Required methods would include:

- Adding values to the list (**add**)
- Removing values from the list (**remove**)
- Way to know whether the list is empty (**isEmpty**)
- Optionally, we may want a **size** method, and a **peek** method (looks at next item without taking any other action)

STACK BASICS

Stack: a linear, ordered collection that allows adding and removing the top element, providing **L**ast **I**n, **F**irst **O**ut (LIFO) access

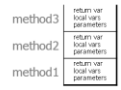
Analogy: trays in the cafeteria. You always get a wet one; the recently washed ones are put on the *top* of the stack

We typically call the add operation "**push**", and the removing operation "**pop**"

Java has a generic Stack class

STACKS IN COMPUTER SCIENCE

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions



- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } and other operators match
  - convert "infix" expressions to "postfix" or "prefix"
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

STACK CASE STUDY

Your text has an informative case study on the use of stacks to evaluate expressions (with error handling, like mismatched parentheses)

How would you write code to evaluate this expression?

```
(18.4 - ((2.3 * 8.5) / (19.5 + (2.7 ^ 4.9))))
```

Approach:

- Parse it (a simple .split or Scanner won't do; we shouldn't depend on spacing)
- Use two stacks, a *symbol* stack and a *number* stack
- push as we encounter new things, pop as we consume things

STACK CASE STUDY: EVALUATION OF EXPRESSION: (2+3)

Token	Action	Symbol Stack	Number Stack
		[ ]	[ ]
(	Push onto symbol stack	[( ]	[ ]
2	Push onto number stack	[( ]	[2.0]
+	Push onto symbol stack	[(, +]	[2.0]
3	Push onto number stack	[(, +]	[2.0, 3.0]
)	Evaluate expression (involves pop operations), push result onto number stack	[ ]	[5.0]

This solves a problem we might have used *recursion* for

QUEUE BASICS

Queue: a linear, ordered collection that allows adding the "back" element and removing the "front" element, providing **F**irst **I**n, **F**irst **O**ut (FIFO) access

Analogy: queues are everywhere! We stand in line at the post office, at the concession stand at a sports event or concert, etc.

Add and remove operations are typically called those names, though you may see the terms "enqueue" and "dequeue"

Java has a Queue interface; instantiate LinkedList object to use it, e.g.,

```
Queue<Integer> q =
    new LinkedList<Integer>();
```

QUEUES IN COMPUTER SCIENCE

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

STACK AND QUEUE OPERATIONS

Typical operations may include…
- Transferring data between them
- Summing up values in them, using add/remove methods
- Removing specified values
- Comparing two structures for similarity

In all of these, we must think about how not to affect the order and not lose values; it's trickier than it first appears. Read the chapter for examples.

MOCKED DATA

REALISTIC DATA FOR YOUR PROOF-OF-CONCEPT APPLICATIONS

MOCKED DATA

When you're showing off your proof-of-concept work, it's good to have realistic data to fill it in and make it seem more complete

There are a variety of websites that will generate this data for you

Let's look at www.mockaroo.com, an example of such a site

TIMERS

CREATING AND RESPONDING TO TIME-BASED EVENTS

TIME-BASED EVENTS

We've introduced ourselves to event-driven programming in a few ways, mostly via Swing events:
- We're called when it's time to paint
- We are informed when the user clicks a button
- We can detect when the user changes a text field's contents

But we can also set up and respond to *time-based events*, using a **Timer**

Note: timers don't offer *real-time guarantees* (because of OS priorities); you'll get called when *at least* the time interval has passed. Timers can bunch up, too

TIMER: STEPS

1. Create a `Timer` object
2. Create a class that extends a `TimerTask` and has a `run` method; we put our code there
3. Schedule the timer event (using the `schedule` method), which lets us specify our `TimerTask`-extending class, when to start the timer, and how many milliseconds to wait between timer messages (see other overloads)
4. We can also cancel the timer at any time via the `cancel` method. If you don't cancel the timer, it will run until you reset the VM or exit the IDE

TIMER: EXAMPLE

```
import java.util.Timer;
import java.util.TimerTask;

public class TimerSample {
    public static void main(String[] args) {
        Timer kidTimer = new Timer();
        class RoadTripTask extends TimerTask {
            public void run() {
                System.out.println("Are we there, yet?");
        }
        kidTimer.schedule(new RoadTripTask(), 0, 1000);
    }
}
```