Lecture 38:
Parallel Streams

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki

Streams in Java 8

- (Lazy) Streams added in Java 8 to enable simpler list processing
  - Similar to functional languages
- Example:
  - names.stream().filter(name -> name.startsWith(“B”)
    .count()
  - Returns count of number of elements of names starting with “B”
  - Compare with how write with loops.
  - Add values in arr: arr.stream().reduce(0,((m,n) -> m+n));

More Streams

- Different kinds of streams
  - IntStream, LongStream, DoubleStream
    - Holds primitive values
  - Stream<T>
    - Holds objects
- Don’t use up storage: Lazy
  - Can have infinite streams ...
  - Intermediate operations always lazy (like filter)
  - Can’t change source

Creating Streams

- Collection classes have stream() and parallelStream() methods
- Array has static method
  - Array.stream(Array<T> arr) returns Stream<T>
- IntStream and LongStream have range(start,end) methods
  - range exclusive at top, rangeClosed inclusive.
- BufferedReader.lines()

Stream Operations

- Filtering Operations on Stream<T>:
  - Stream<T> filter(Predicate<T> prop)
  - Stream<R> map(Function<T,R> f)
  - Stream<T> distinct()
  - Stream<R> flatMap(Function<T,Stream<R>> f)
- Terminal Operations:
  - int count()
  - void forEach(Consumer<T> action)
  - boolean allMatch(Predicate<T> f)   anyMatch

Parallel Streams

- Stream<T> parallelStream()
- Tries a divide and conquer approach to solving problem.
  - Requires no explicit effort by programmer if data structure set up properly (Spliterator)

Parallel Streams Example

```
public class Streaming {
    private long countPrimes(int max) {
       return LongStream.range(1, max).parallel().filter(this::isPrime).count();
    }

    private boolean isPrime(long n) {
       return n > 1 && LongStream.rangeClosed(2, (long)Math.sqrt(n)).
                                     noneMatch(divisor -> n % divisor == 0);

    }

    public static void main(String[] args) {
        Streaming streamer = new Streaming();
        System.out.println(streamer.countPrimes(13));
        System.out.println(streamer.countPrimes(10000000));
    }
```

Static Parallel Streams Ex.

```
public class StaticStreaming {
    private static long countPrimes(int max) {
       return LongStream.range(1, max).parallel().
                                       filter(StaticStreaming::isPrime).count();
    }

    private static boolean isPrime(long n) {
       return n > 1 && LongStream.rangeClosed(2, (long)Math.sqrt(n)).
    noneMatch(divisor -> n % divisor == 0);

    }

    public static void main(String[] args) {
        System.out.println(StaticStreaming.countPrimes(13));
        System.out.println(StaticStreaming.countPrimes(10000000));
    }
}
```

Double Colon Operator

- The code obj::isPrime is an abbreviation for a lambda expression formed from isPrime:
  - (n -> obj.isPrime(n))

OO-Design

What are objects?

- Objects have
  - State/Properties — represented by instance variables
  - Behavior — represented by methods
    - accessor and mutator methods

Calculator

- Calculator class: User interface
  - including buttons and display
  - No real methods — construct & associate listeners
- State class: Current state of computation
  - Methods invoked by listeners
  - Communicate results to user interface
- Listener classes: Communicate from interface to state

Model-View-Controller

State

- Instance variables:
  - partialNumber, numberInProgress?, numStack, calcDisplay
- Methods:
  - addDigit(int Value)
  - doOp(char op)
  - enter, clear, pop

Model-View-Controller

- Dissociate user interface with the “model”
  - “model” represents actual computation
  - May have multiple alternate user interfaces
    - Mobile vs laptop versions of UI
- Model should be unaffected by change in UI.
- In Java UI generally served by “event thread”
  - If tie up event-thread with computation then user-interface stops being responsive.

Designing Programs

- Identify the objects to be modeled
  - E.g., Frogger game, Shell game
- List properties and behaviors of each object
  - Model properties with instance variables
  - Model behavior with methods (*write spec*)
- Refine by filling in the details
  - Hold off committing to details of representation as long as possible.

Implementation

- Write in small pieces. Test thoroughly before moving on.
- Solve simpler problem first — use “stubs” if necessary.
- Refactor as code becomes more complex.

Reading on Object-Oriented Design

- **Practical Object-Oriented Design in Ruby: An Agile Primer** by Sandi Metz, 2013
- **Design Patterns: Elements of Reusable Object-Oriented Software** by “Gang of Four”, 1994