Bonus Lecture: Java Streams

Big picture

- Streams are a Java feature but are one realization of big ideas that aren't unique to Java
- Some of the ideas come from “functional programming,” common in languages such as lisp (Racket is a modern lisp-like language which Matthew is one of the leaders of)
- It's an instance of “Declarative Programming” which we'll see again a few times, and is awesome

Declarative Programming

- We've mostly discussed “imperative” programming in this program
- Imperative programming is where we specify the steps that need to be done to achieve some task
- For loops and if statements are classic “imperative” tools
- By contrast, declarative programming we write what result we want and any looping/if statements/etc are hidden away in some library

Advantages of declarative programming

- Clarity: don't pollute our code with details about “how” if we don't need to. Declaritive code tends to be shorter at the “application” level
- Flexibility of implementation: there might be many ways to achieve the result we want. If users specify what they want, library authors can improve their implementations to give users free speedups by upgrading, or can even dynamically choose the fastest version at runtime

Example: HTML/CSS

- HTML/CSS are declarative programming langauges. We specify what we want the rendered document to look like
- The browsers can implement the rendering process in many different ways and can make old web pages render faster by improving browser algorithms

Java Streams Example

```
//square all the odd numbers in the list and add them up
var result = list.stream()
    .filter(x -> x % 2 != 0) //only keep the odd numbers
    .map(x -> x * x) //square each of them
    .mapToInt(Integer::intValue) //Java nonsense...
    .sum(); //then add them up
```

Comparison

```
var sum = 0;
for(var x : list){
    if( x % 2 != 0){
        sum += x*x;
    }
}
```

Advantages of the streams version

- No temporary variables or variable modifications. The sum() method at the end gives us the value and we didn't need to define (name), intialize, and modify a variable ourselves
- Steps are visually separated: filter, then square, then sum them up. The loop mixed/combined those things
- If our list is big, we can use `.parallelStream` instead of stream to automatically use multiple threads to speed up the computation! It would be a pain to do that with the for loop version!
- If the code compiles, and it makes sense reading top to bottom, it's probably correct. It's tough to have edge cases!

Stream Concepts

- A stream is sort of like an iterator in that it provides access to the elements of a collection
- Unlike an iterator, we don't write a loop to explicitly examine each element
- We specify operations which will be applied to the elements of the stream as needed
- We can combine many such operations into a “pipeline”
- Once we've modified the stream how we'd like, we can “collect” it into a Collection (like an ArrayList or Set) or perform a “terminal operation” like summing or counting the number of elements

Getting an initial stream

- We start with a Stream of T objects (there are special types `IntStream` etc that work with primitive types too)
- A `.stream()` method was added to the `Collection` interfaces when streams were added, so all collections can be converted into streams
- There is also an `Arrays.stream` method that will create a stream out of elements of an array
- There's other methods that return streams like `static Stream<string> File.lines(Path filename)`
- Also `Stream.of` which takes all the values you want to pass as arguments

“Intermediate Operations”

- These operations modify a stream and return a new stream
- Some operations like `filter` will remove elements from the original stream
- Some operations like `map` will modify each element of the stream
- These operations are `lazy` meaning that they will not be executed until future code actually asks for values from the stream
- This is pretty similar to how your synthesizer worked!

Classic function: map

- `map` is a function available in most programming languages which is a staple of “functional programming”
- In pseudocode it looks something like this:

```
sequence map(sequence, function)
    return sequence(function(element) for each element in sequence)
```

- In english, it returns a sequence which contains the result of function on each element of the input sequence
- In C++ it's called “transform”

Functional Programming: “Higher Order Functions”

- “Functional Programming” is a philsophy of programming where we build programs by “composing” (nesting) functions
- This requires a language to support “higher order functions” which refers to functions which can take functions as parameters, or return functions as results
- Java approximates this with interfaces, and makes it convenient with lambda functions
- Map is a high order function because one of it's parameters is “the function to apply to each element”

Classic function: filter

```
sequence filter(sequence, predicate)
    return sequence( elements from sequennce where predicate(element) is true)
```

- This gives us a potentially shorter sequence that only keeps elements that “pass a test”
- Remember a predicate is just `boolean function(T)`

Other intermediate methods

- `distinct` — remove all duplicates
- `sorted` — duh, takes an optional comparator
- `limit(maxSize)` — keep up to maxSize elements

Terminal operations

- Terminal operations take a stream and produce some sort of result
- Some of them, like `sum` combine all the results into a value
- Some of them “collect” the elements into a `Collection` such as a list or set
- Interestingly, nothing happens until one of these methods get called because all previous steps are lazy!

Terminal examples

- `allMatch`/`anyMatch` — takes a predicate and returns a boolean sayign whether it returns true for all/any of the elements of the stream. It might only need to look at the first element!
- `count`/`empty` — how many elements are in the stream?
- `toArray` — duh

Classic terminator: “reduce”

- Also known as “fold” in functional languages
- Starts with a “seed value” and then does basically this loop

```
result = seed
for element in sequence:
    result = combine(result, element)
```

- combine is a function provided by the caller
- `sum` could be implemented as `reduce(0, (x, y) → x + y)`

Do-it-all terminator: Collect

- `collect` takes a `Collector` which says how to process the elements
- The `Collectors` class has static methods returning a lot of the collectors you might want to use such as:
  - `joining()` — concatenate them into a big string
  - `toList()` — make an arraylist
  - `groupingBy()` — basically gives you `Map>` so you can get all the elements in each category as a list
  - many more

Gotchas

- Due to generics only working for objects, I often find myself needing to use methods like `mapToFloat` or `mapToInt` to convert a Stream to FloatStream or IntStream which hold primitive types
- Streams generally are single use, so you can't use more than 1 terminal operation on a single stream
- The documentation is hard to follow because there's quite a few weird generic interfaces, especially related to collectors, I often search for examples specifically which I usually don't need to do!

Overview

- Streams are a succinct way to specify data processing pipelines in Java
- Most langauges have similar ideas + libraries
- Writing code in this style can often make it easier to parallelize your implementation with a 1 line change!
- A good mindset to have is that of “data transformation” ... how does my data need to be manipulated as I work towards a result