Iteration and Streaming in Java

COMP2209 - Programming III

Dr Julian Rathke
Functional Programming and Lists

• One of the missing ingredients in our round-up of functional programming support in Java is the support for Lists.
• Yes, of course Java has arrays and even ArrayLists but these are not the same as in Haskell.
• For a start, Haskell lists are structural and immutable.
• Moreover, the means of traversing Java lists using loops is not quite in functional style.
• It is far too imperative in that traversals are done manually by the client by explicitly navigating the list.
External vs Internal Iteration

• Java provides excellent support for Collections of data; ArrayLists, HashSets, Stacks, etc.
• Part of this support is Iterator objects for these Collection data structures.
• The idiom for iteration goes something like this:

```java
Iterator it = shapes.getIterator()
while (it.hasNext()){
    Shape s = it.next();
    s.setColor(Color.RED)
}
```

This code repeatedly calls the iterator to visit the next item in the collection. We call this approach **external iteration**.
External vs Internal Iteration

- Although simple, there are some drawbacks to external iteration:
  - It is inherently serial in Java and it is required to process the elements of the Collection in the order specified by the iterator.
  - A library may be able to implement a generalised for loop in a non-standard way to take advantage of data reorderings, parallelism, lazy evaluation, or fast return evaluation.
  - Explicit iteration is not terrible modular - if we have two functions that iterate and perform transformations \( f \) and \( g \) on the elements of a Collection then the composition \( f \cdot g \) uses two iterators where really the loops should be fused.
Internal Iteration

- Internal iteration takes a different approach: the client calls a generalised forEach method and the library does the actual iteration

  \[
  \text{shapes.foreach}(s \rightarrow \{s\text{.setColor(RED)}\})
  \]

- So control flow management for actually doing the iteration is shifted away from client code.

- This then allows for optimisations of the sort mentioned above and offers more flexibility in that respect.

- This style of iteration is also therefore suited to compositional iterations as an implementation that sees a sequence of internal iterations has the flexibility to adjust the control flow to combine the actions during each iteration.

- Support for internal iteration is provided in Java 8 in the form of Streams.
Streams in Java

- Streams are a new list like feature in Java that work smoothly with the functional programming features.
- For example, streams are immutable lists of data that support a number of useful stream operations.
- These operations typically will produce another stream so as to allow composition of stream operations.
- Streams support both lazy and eager implementations of list processing functions
  - **Lazy**: stream values are calculated as required as one iterates over the output stream
  - **Eager**: stream values of the output are calculated at the point the stream function is called.
- In essence, Streams are Java’s version of Haskell’s lazy lists.
Intermediate and Terminal Stream Operations

- There are two natural types of stream operations
  - **Intermediate Stream Operations**
    - naturally implemented as lazy operations
    - return another stream
    - can be composed with other intermediation operations
    - examples are map and filter
  - **Terminal Stream Operations**
    - naturally implemented as eager operations
    - consumes the input stream to produce a side-effect or return value
    - examples are foreach, findFirst, iterator
Working with Streams

• Let’s get down to some practicalities.
• Look in the API documentation for `java.util.Stream<T>`!
• To create a Stream it is usually best to convert an existing Collection object - class Collection now has a method `stream()` that returns a Stream containing all the elements of the given Collection
• You can also create Streams incrementally using a Stream.Builder or you can use `Stream.of(T ... objs)` to supply a number of objects that will be created as a stream.

```java
String[] array = ...
Set<String> hash = ...
Stream<String> arrstr = Arrays.stream(array);
Stream<String> hashstr = hash.stream();
```
Common operations

• The first stream operation should be very familiar

```java
Stream<R> map( Function<T,R> f )
```

• This takes a function and applies the function to every element in the stream to produce a new transformed stream.

• It is a lazy, intermediate operation. For example:

```java
Stream.of(1, 2, 3, 4)
    .map(num -> num * num)
    .forEach(System.out::println); // 1 4 9 16
```

forEach iterates across the stream for its side-effect. This is a terminal stream operation

What’s that? Oh, yes, well spotted, this makes Stream<T> into a functor.
flatMap and the monad Stream\(<T>\>

• This intermediate stream operation has the following type

\[
\text{Stream}<R> \ \text{flatMap} \ (\ \text{Function}<T, \ \text{Stream}<R> \ > f)
\]

• It takes each element of a Stream\(<T>\), producing a Stream\(<R>\) for each but then flattening the output in to one big Stream\(<R>\)

• Yes, that type does look familiar. Given a Stream\(<T>\) and a function Function\(<T, \ \text{Stream}<R>\), we get a Stream\(<R>\)

• In Haskell this would be type

\[
\text{flatMap} :: [ \ a \ ] \to (a \to [b]) \to [b]
\]

• Of course, this is the bind operation \(>>=\) that makes Stream\(<T>\) into a \textbf{monad}!

• Well, with the single element method as pure too.
More operations in java.util.Stream

- We’ve seen map and flatMap. There are a whole load more stream operations listed in java.util.Stream

- Intermediate operations
  - **filter** - takes a predicate and returns a stream with elements that satisfy the predicate
  - **peek** - takes a consumer (T → void) and returns a stream with the same elements but also feeds each element to the consumer.
  - **distinct** - returns a stream with duplicate elements removed (using .equals)
  - **sorted** - for streams of Comparable elements it returns the stream sorted to their natural order
  - **iterate** - takes a unary operator and a start element and produces an infinite stream by repeatedly applying the function
More operations in java.util.Stream

- Terminal operations
  - forEach: takes a consumer (*T → void*) and feeds each element in the stream to it. Unlike peek it does not produce an output stream.
  - toArray: simply converts the stream to an array and closes the stream.
  - reduce: similar to fold in Haskell. Takes a base element, and a binary operator and applies it element wise across the stream by accumulating the result.
    - Neither foldl or foldr is specified by the API - no guarantee of order.
  - max, min, count: examples of reduce
  - collect: a convenient combination of map, reduce and filter - look up Collector in the API documentation.
A typical example

Suppose we have a class Person with a bunch of data including a set of the jobs that they have done. Assume an accessor method for this:

```java
Set<String> getJobs()
```

Suppose also that we have an array `ps` of Person and wish to list all the different nationalities that appear in `ps`

```java
ps.stream()  // Stream<Person>
   .map(x -> x.getJobs())  // Stream<Set<String>>
   .flatMap(x -> x.stream())  // Stream<String>
   .distinct()  // Stream<String>
   .collect(Collectors.toList())  // List<String>
```

Uses a terminal helper method to create a Collector that takes all results from Stream and dumps them in a List
You use it, you lose it.

• An interesting feature of Streams in Java is that they are *stateful*.

• Once you have operated on a Stream object it remembers that it has been operated on and will not allow further operations.

• An `IllegalStateException` is thrown if you try.

• This allows Stream implementations to optimise traversal and protect against concurrent modifications etc.

```java
Stream<Integer> str = Stream.of(10, 2, 33, 45);
Stream<Integer> strF = str.filter( x -> x < 30 );
Stream<Integer> strM = str.map( x -> x + 5 );
```

...
Optional\textless T\textgreater 

- A method of note in java.util.Stream is \texttt{findFirst}
- It returns the first element of the stream but it is interesting because of its return type viz \texttt{Optional<T>}
- An \texttt{Optional<T>} object is a wrapper for a value of type T. It may hold an actual value, or it may not
  - The method \texttt{isPresent()} lets you check whether the optional actually holds a value.
- Other interesting methods in \texttt{Optional<T>} are
  - \texttt{Optional<R> map( Function<T,R> )} and
  - \texttt{Optional<R> flatMap (Function<T, Optional<R>>)}

\textbf{Maybe} you’ve already spotted that these make \texttt{Optional<T>} into a monad.

\textbf{I said, MAYBE} you’ve already spotted …
Parallel Streams

• We have seen there is a method in the Collection class that allows the collection object to be converted to a stream, i.e. Collection.stream()

• Similar to this is the method Collection.parallelStream()

• This converts the collection object to a stream that is suitable for parallel processing.

• This allows non-serial access to the stream object!

• Be wary when processing these that order does not matter

• For example, sorting can end up being incorrect if the algorithm depends on the order of comparisons.

• For streams where parallel processing is possible this can greatly improve efficiency though.
A perfect example

- Let’s write a short program to determine whether an integer is perfect, that is, equal to its product of factors (not including itself).

```java
import java.util.stream.*;
import java.util.Collection;

public class Perfect {

    static boolean perfect(int n){
        int factorSum = IntStream.range(1,n)
            .filter( x -> (n % x) == 0 )
            .sum();
        return (factorSum == n );
    }

    ...
```
We may as well compare this to the Haskell version.
A perfect Haskell example

```haskell
perfect :: Int -> Bool
perfect n = factorSum == n
    where factorSum =
        sum [ m | m <- [1..n-1], n `mod` m == 0 ]

main :: IO()
main = print [ n | n <- [1..50000], perfect n]
```

This is much slicker code but unfortunately it runs much more slowly than the Java version. 😞
YOUR QUESTIONS

Next Lecture:
Functional Programming in JavaScript