Lambdas and Streams

Monday, March 19, 2018 10:13 AM

Closure

- Lambda expression stores functions + data from defined environment (captured variables)
- Can be stored and passed around

Cross-Barrier State Manipulator

- FP -> assign fn to variable, compose dynamically, partially evaluate
- Access internals without implementer interface

`Optional<T>`

- Wrapper around T object/ null (nullable reference)
- NaN = not a number
- Null explicitly in codomain and handled by Optional
    - Does not throw NullPointerException
    - Null = NoSuchElementException with get() method
    - Not supported by Java Collections Framework

Initialising Optional

- `Optional.of(value);` `// value`
- `Optional.empty();` `// null`
- `Optional.ofNullable(value);` `// null or value`

Delayed Data

- Infinite lists with lazy evaluation
- Store function that generates elements instead of elements themselves
- 2 functions: generate first element/ the rest of the list
    - Values not generated until needed

Method 1

```
public static <T> InfiniteList<T> generate(Supplier<T> supply) {
  return new InfiniteList<T>(supply,
      () -> InfiniteList.generate(supply));
}
```

Method 2

```
public static <T> InfiniteList<T> iterate(T init, Function<T, T> next) {
  return new InfiniteList<T>(() -> init,
    () -> InfiniteList.iterate(next.apply(init), next));
}
```

Stream

- `takeWhile, dropWhile`
- Intermediate and terminal operations
    - Intermediate: map, filter, peek
    - Terminal: forces stream to be evaluated
        - Stream can only be consumed once
- Creating streams

    - Stream.of, Arrays.stream, Collections.stream, Files.lines
    - Generate (Supplier) or iterate (initial value + incremental operation)
- flatMap
    - chars() returns a stream
    - flatMap flattens multiple streams into one stream
- Stateful operations: distinct, sorted
    - Bounded: needs to know every element
- Code becomes more declarative - need not be concerned about implementation of stream operations