CSE216 Programming Abstractions
Streams in Java

YoungMin Kwon

Lambda in Java

- Functional Interface (AKA Single Abstract Method)
  - An interface that has only one abstract method
- SAM interfaces can be instantiated using lambda expressions
  - Lambda expression will be used as the body of the single abstract method.

Lambda in Java

- Lambda expression
  - `Parameters -> Body`

- Parameters
  - `(Type1 Name1, Type2 Name2, …) -> Body`
  - `(Name1, Name2, …) -> Body` when types are inferable
  - `Name1 -> Body` for a single parameter function

Lambda in Java

- Body
  - ```
    Parameters -> {
        Statemet1;
        Staement2;
        …
        return statement;
    }
    ```
  - `Parameters -> Expression` for a single return statement of expression
    ```

Lambda in Java

- Example

```
public class Lambda {

    public static interface Fun<P,R> {

        //Single Abstract Method
        public R fun(P p);

        //default method with a body
        public default R app(P p) {
            return fun(p);
        }
    }
    static Fun<Double,Double> abs = x -> x < 0 ? -x: x;
…
    public static void test() {
        System.out.format("abs -2 = %f\n", abs.app(-2.));

        double z = ((Fun<Double,Double>) x -> x * x).app(2.);
        System.out.format("square 2 = %f\n", z);
…
```

```
public static Fun<Double,Double> deriv(Fun<Double,Double> f) {
    return x -> (f.app(x + eps) - f.app(x)) / eps;
}

public static double solve(Fun<Double,Double> f, double x0) {
    double x1 = x0 - f.app(x0) / deriv(f).app(x0);
    return abs.app(x0 - x1) < eps ? x1 : solve(f, x1);
}

public static double sqrt(double x) {
    return solve(y -> y * y - x, 1);
}

public static void test() {
    System.out.format("abs -2 = %f\n", abs.app(-2.));

    System.out.format("sqrt 2 = %f\n", sqrt(2));

    System.out.format("sqrt 2 = %f\n", solve(x -> {
        return x * x - 2;
    }, 1));
    System.out.format("sqrt 2 = %f\n", solve(x -> {
        double y = x * x - 2;
        return y;
    }, 1));
}
```

Streams in Java

- Streams
  - A sequence of data elements made available over time
- Characteristics of java streams
  - Streams are not data storage
    - It conveys data from a source through a pipeline of computational operations
  - Streams are functional
    - Operations on a stream produce data, but they do not modify the source

Streams in Java

- Streams are lazy
  - Until necessary, actual operations are delayed
- Streams are possibly unbounded
  - E.g. the stream of natural numbers
- Streams are consumable
  - In java, the elements of a stream are visited only once

An Introductory Example

- Finding a city: using a for-loop

```
public static class FindCity {

    //check if cityList has city
    public static boolean findCity(String city, List<String> cityList) {
        for(String c: cityList)
            if(c.equals(city))
                return true;
        return false;
    }

    public static void test() {
        List<String> cities = new ArrayList<String>(Arrays.asList(
                "Chicago", "New York", "LA", "Denver", "Seattle", "LA"));
        findCity("Seattle", cities);
    }
```

An Introductory Example

- Finding a city: using a stream

```
public static class FindCity {

    public static boolean findCity2(String city, List<String> cityList) {
        return cityList.stream()  //Collection.stream() returns a stream
                       .map(c -> c.equals(city)) //city -> true/false
                       .filter(c -> c)           //leave only true
                       .reduce(false/*identity*/,
                               (acc, ele) -> (acc || true)/*accumulator*/)
    }

    public static void test() {
        List<String> cities = new ArrayList<String>(Arrays.asList(
                "Chicago", "New York", "LA", "Denver", "Seattle", "LA"));
        findCity2("Seattle", cities);
    }
```

An Introductory Example

- Sorting cities by their names

```
public static void sortCity(List<String> cityList) {
    cityList.stream()
            .sorted((a, b) -> a.compareTo(b))
            .forEach(c -> System.out.println(c));
}

public static void test() {
    List<String> cities = new ArrayList<String>(Arrays.asList(
            "Chicago", "New York", "LA", "Denver", "Seattle", "LA"));
    System.out.println("Cities");
    sortCity(cities);
}
```

Second Example

- Compute the total amount of discounts
  - Items above $20.00 are discounted by 10%
  - Prices are given in the string type

  - 1. using a for loop
    - For each item, if it is above $20.00, accumulate its discount amount

  - 2. using a stream
    - Filter the items above $20.00
    - Accumulate their discounts amounts

Second Example

- Using a for-loop

```
public static double totalDiscount(List<String> priceList) {
    double total = 0;
    for(String str: priceList) {
        double p = Double.parseDouble(str);
        if(p >= 20)
            total += p * 0.1;  //accumulate the discount amount
    }
    return total;
}

public static void test() {
    List<String> prices = new ArrayList<String>(Arrays.asList(
            "9.99", "14.99", "20.99", "21.99", "22.99", "20.99"));

    double t = totalDiscount(prices);
    onFalseThrow(8.69 < t && t < 8.70);
}
```

Second Example

- Using a stream

```
public static double totalDiscount2(List<String> priceList) {
    return priceList.stream()
                    .map(str -> Double.parseDouble(str))
                    .filter(p -> p >= 20)
                    .reduce(0./*identity: initial value*/,
                            (sum, x) -> sum + x * 0.1 /*accumulator*/);
}

public static void test() {
    List<String> prices = new ArrayList<String>(Arrays.asList(
            "9.99", "14.99", "20.99", "21.99", "22.99", "20.99"));

    double t = totalDiscount2(prices);
    onFalseThrow(8.69 < t && t < 8.70);
}
```

Obtaining a Stream

- Some of the ways to obtain a stream
  - `stream()` and `parallelStream()` from a Collection
  - `Arrays.stream(Object[])`
  - `Stream.of(Object[])`
  - `BufferedReader.lines()` for the lines of a file

  …

Streams are Lazy

- Lazy evaluation
  - Elements are not evaluated until they are necessary
- Intermediate operations
  - Operations that produce streams (map, filter, …)
  - Intermediate operations are lazy
  - E.g. filter does not perform filtering
    - Instead, it creates a stream that, when traversed, contains the filtered elements
    - The traversal does not begin until a terminal operation is executed

Streams are Lazy

- Stateless/Stateful intermediate operations
  - Stateless operation: retain no previous elements to produce next elements (filter, map)
  - Stateful operation: may incorporate previous elements when processing new elements (sort)

- Terminal operations
  - Operations that produce values or side-effects (reduce, forEach)
  - They may traverse the stream
  - After a terminal operation, the stream is consumed and can no longer be used.

```
public static class DemonstrateLazyness {

    public static double toDouble(String str) {
        System.out.println("toDouble: " + str); //print a message
        return Double.parseDouble(str);
    }

    public static double discount(Double price) {
        System.out.println("discount: " + price); //print a message
        return price * 0.1;
    }

    public static boolean hasDiscountItem(List<String> priceList) {
        return priceList.stream()
                        .filter(str -> toDouble(str) >= 20.0)
                        .map(str -> discount(Double.parseDouble(str)))
                        .findFirst()  //findFirst returns an Optional
                        .isPresent(); //isPresent returns whether
                                      //Optional has a valid value
    }
```

```
public static void hasDiscountItem2(List<String> priceList) {
    System.out.println("Filtering discount items...");
    Stream<String> discountItems =
            priceList.stream()
                    .filter(str -> toDouble(str) >= 20.0);

    System.out.println("Collecting the discount amounts...");
    Stream<Double> discounts =
            discountItems.map(str -> discount(Double.parseDouble(str)));

    System.out.println("Finding the first discount amount...");
    Double firstDiscountAmount =
            discounts.findFirst()  //findFirst returns an Optional
                    .get();        //returns a valid value or an exception

    System.out.format("First discount amount: %5.2f\n",
                      firstDiscountAmount);
}
```

```
public static void test() {
        System.out.println("Testing DemonstrateLazyness...");

        List<String> prices = new ArrayList<String>(Arrays.asList(
                "9.99", "14.99", "20.99", "21.99", "22.99", "20.99"));

        onFalseThrow(hasDiscountItem(prices) == true);
        hasDiscountItem2(prices);
        System.out.println("Success");
    }
}
```

```
Testing DemonstrateLazyness...
toDouble: 9.99
toDouble: 14.99
toDouble: 20.99
discount: 20.99

Filtering discount items...
Collecting the discount amounts...
Finding the first discount amount...
toDouble: 9.99
toDouble: 14.99
toDouble: 20.99
discount: 20.99
First discount amount:  2.10
```

Unbounded Streams

- Iterate method

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

  - Returns an infinite sequence produced by iteratively applying f to seed
  - E.g. seed, f(seed), f(f(seed)), …

Rand Stream

```
//infinite stream using iterate
public static class RandStream {
    static long nextRand(long x) {
        return (x * 16807) % 0x7fffffff;
    }

    static Stream<Long> randStream(int seed) {
        return Stream.iterate(nextRand(seed), x -> nextRand(x));
    }

    public static void test() {
        System.out.println("Testing RandStream...");
        Iterator<Long> iter = randStream(8).iterator();
        onFalseThrow(iter.next() == 134456);
        onFalseThrow(iter.next() == 112318345);
        onFalseThrow(iter.next() == 96298702);
        onFalseThrow(iter.next() == 1437098323);
        System.out.println("Success");
    }
}
```

Parallel Stream

- Streams can facilitate parallel execution
  - All stream operations can execute either in serial or in parallel
    - For-loops are inherently serial
  - The only difference is the creation of the initial stream
    - Collection.stream() produces a sequential stream
    - Collection.parallelStream() produces a parallel stream

```
public static boolean hasDiscountItem(List<String> priceList) {
    return priceList.parallelStream()  //create a parallel stream
                    .filter(str -> toDouble(str) >= 20.0)
                    .map(str -> discount(Double.parseDouble(str)))
                    .findFirst()
                    .isPresent();
}

public static void hasDiscountItem2(List<String> priceList) {
    System.out.println("Filtering discount items...");
    Stream<String> discountItems =
            priceList.parallelStream() //create a parallel stream
                     .filter(str -> toDouble(str) >= 20.0);

    System.out.println("Collecting the discount amounts...");
    Stream<Double> discounts =
            discountItems.map(str -> discount(Double.parseDouble(str)));

    System.out.println("Finding the first discount amount...");
    Double firstDiscountAmount =
            discounts.findFirst()
                     .get();

    System.out.format("First discount amount: %5.2f\n", firstDiscountAmount);
}
```

Output: Parallel Stream

```
Filtering discount items...
Collecting the discount amounts...
Finding the first discount amount...
toDouble: 21.99
toDouble: 20.99
toDouble: 14.99
toDouble: 9.99
discount: 20.99
toDouble: 20.99
discount: 21.99
toDouble: 22.99
discount: 20.99
discount: 22.99
First discount amount:  2.10
```

Output: Sequential Stream

```
Filtering discount items...
Collecting the discount amounts...
Finding the first discount amount...
toDouble: 9.99
toDouble: 14.99
toDouble: 20.99
discount: 20.99
First discount amount:  2.10
```

Thank you for your attention during the semester!

Any questions or comments?