

Building Java Programs

Chapter 19

Functional Programming with Java 8

Copyright (c) Pearson 2016.
All rights reserved.

What is FP?

- **functional programming:** A style of programming that emphasizes the use of **functions** (methods) to decompose a complex task into subtasks.
 - Examples of functional languages:
LISP, Scheme, ML, Haskell, Erlang, F#, Clojure, ...
- Java is considered an object-oriented language, not a functional language.
- But Java 8 adds several language features to facilitate a partial functional programming style.

Java 8 FP features

- 1. Effect-free programming
- 2. Processing structured data via functions
- 3. First-class functions
- 4. Function closures
- 5. Higher-order operations on collections

Effect-free code (19.1)

- **side effect:** A change to the state of an object or program variable produced by a call on a function (i.e., a method).
 - example: modifying the value of a variable
 - example: printing output to System.out
 - example: reading/writing data to a file, collection, or network

```
int result = f(x) + f(x);  
int result = 2 * f(x);
```

- Are the two above statements equivalent?
 - Yes, if the function $f()$ has no *side effects*.
 - One goal of functional programming is to minimize side effects.

Code w/ side effects

```
public class SideEffect {  
    public static int x;  
  
    public static int f(int n) {  
        x = x * 2;  
        return x + n;  
    }  
  
    // what if it were 2 * f(x)?  
    public static void main(String[] args) {  
        x = 5;  
        int result = f(x) + f(x);  
        System.out.println(result);  
    }  
}
```

First-class functions (19.2)

- **first-class citizen:** An element of a programming language that is tightly integrated with the language and supports the full range of operations generally available to other entities in the language.
- In functional programming, functions (methods) are treated as first-class citizens of the languages.
 - can store a function in a variable
 - can pass a function as a parameter to another function
 - can return a value from a function
 - can create a collection of functions
 - ...

Lambda expressions

- **lambda expression** ("lambda"): Expression that describes a function by specifying its parameters and return value.
 - Java 8 adds support for lambda expressions.

- Syntax:

(parameters) -> expression

- Example:

*(x) -> x * x // squares a number*

- The above is roughly equivalent to:

```
public static int squared(int x) {  
    return x * x;  
}
```

MathMatrix add / subtract

- Recall the MathMatrix class:

```
public MathMatrix add(MathMatrix rhs) {  
    int[][] res = new int[cells.length][cells[0].length];  
    for (int r = 0; r < res.length; r++)  
        for (int c = 0; c <= res[0].length; c++)  
            res[r][c] = cells[r][c] + rhs.cells[r][c];  
    return new MathMatrix(res);  
}
```

```
public MathMatrix subtract rhs) {  
    int[][] res = new int[cells.length][cells[0].length];  
    for (int r = 0; r < res.length; r++)  
        for (int c = 0; c <= res[0].length; c++)  
            res[r][c] = cells[r][c] - rhs.cells[r][c];  
    return new MathMatrix(res);  
}
```


MathMatrix add / subtract

- GACK!!!
- How do we generalize the idea of "add or subtract"?
 - How much work would it be to add other operators?
 - Would functional programming help?

Code w/ lambdas

- We can represent the math operation as a lambda:

```
public MathMatrix add(MathMatrix rhs) {  
    return getMat(rhs, (x, y) -> x + y);  
}
```

```
public MathMatrix subtract(MathMatrix rhs) {  
    return getMat(rhs, (x, y) -> x - y);  
}
```

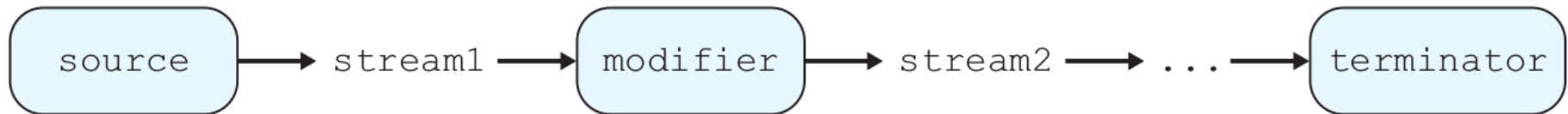
giveProblems method

```
private MathMatrix getMat(MathMatrix rhs,  
                           IntBinaryOperator operator) {  
  
    int[][] res = new int[cells.length][cells[0].length];  
  
    for (int r = 0; r < cells.length; r++) {  
        for (int c = 0; c < cells[0].length; c++) {  
            int temp1 = cells[r][c];  
            int temp2 = rhs.cells[r][c];  
            res[r][c] = operator.applyAsInt(temp1, temp2);  
        }  
    }  
  
    return new MathMatrix(res);  
}
```

[**// IntBinaryOperator Documentation**](#)

Streams (19.3)

- **stream**: A sequence of elements from a data source that supports aggregate operations.
- Streams operate on a data source and modify it:



- example: print each element of a collection
- example: sum each integer in a file
- example: concatenate strings together into one large string
- example: find the largest value in a collection
- ...

Code w/o streams

- Non-functional programming sum code:

```
// compute the sum of the squares of integers 1-5
int sum = 0;
for (int i = 1; i <= 5; i++) {
    sum = sum + i * i;
}
```

The map modifier

- The `map` modifier applies a lambda to each stream element:
 - **higher-order function**: Takes a function as an argument.
- Abstracting away loops

```
// compute the sum of the squares of integers 1-5
int sum = IntStream.range(1, 6)
    .map(n -> n * n)
    .sum();
```

```
// the stream operations are as follows:
```

```
IntStream.range(1, 6) -> [1, 2, 3, 4, 5]
                    -> map -> [1, 4, 9, 16, 25]
                    -> sum -> 55
```

The filter modifier

- The `filter` stream modifier removes/keeps elements of the stream using a boolean lambda:

```
// compute the sum of squares of odd integers
int sum =
    IntStream.of(3, 1, 4, 1, 5, 9, 2, 6, 5, 3)
    .filter(n -> n % 2 != 0)
    .map(n -> n * n)
    .sum();
```

```
// the stream operations are as follows:
IntStream.of -> [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
    -> filter -> [3, 1, 1, 5, 9, 5, 3]
        -> map -> [9, 1, 1, 25, 81, 25, 9]
            -> sum -> 151
```

Streams and methods

- using streams as part of a regular method:

```
// Returns true if the given integer is prime.  
// Assumes n >= 0.  
public static boolean isPrime(int n) {  
    return IntStream.range(1, n + 1)  
        .filter(x -> n % x == 0)  
        .count() == 2;  
}
```


The reduce modifier

- The `reduce` modifier combines elements of a stream using a lambda combination function.
 - Accepts two parameters: an initial value and a lambda to combine that initial value with each next value in the stream.

```
// Returns n!, or 1 * 2 * 3 * ... * (n-1) * n.  
// Assumes n is non-negative.
```

```
public static int factorial(int n) {  
    return IntStream.range(2, n + 1)  
        .reduce(1, (a, b) -> a * b);  
}
```

Stream operators

Method name	Description
<code>anyMatch (f)</code>	returns true if any elements of stream match given predicate
<code>allMatch (f)</code>	returns true if all elements of stream match given predicate
<code>average ()</code>	returns arithmetic mean of numbers in stream
<code>collect (f)</code>	convert stream into a collection and return it
<code>count ()</code>	returns number of elements in stream
<code>distinct ()</code>	returns unique elements from stream
<code>filter (f)</code>	returns the elements that match the given predicate
<code>forEach (f)</code>	performs an action on each element of stream
<code>limit (size)</code>	returns only the next size elements of stream
<code>map (f)</code>	applies the given function to every element of stream
<code>noneMatch (f)</code>	returns true if zero elements of stream match given predicate

Stream operators

Method name	Description
<code>parallel()</code>	returns a multithreaded version of this stream
<code>peek(f)</code>	examines the first element of stream only
<code>reduce(f)</code>	applies the given binary reduction function to stream elements
<code>sequential()</code>	single-threaded, opposite of <code>parallel()</code>
<code>skip(n)</code>	omits the next <code>n</code> elements from the stream
<code>sorted()</code>	returns stream's elements in sorted order
<code>sum()</code>	returns sum of elements in stream
<code>toArray()</code>	converts stream into array

Static method	Description
<code>concat(s1, s2)</code>	glues two streams together
<code>empty()</code>	returns a zero-element stream
<code>iterate(seed, f)</code>	returns an infinite stream with given start element
<code>of(values)</code>	converts the given values into a stream
<code>range(start, end)</code>	returns a range of integer values as a stream

Optional results

- Some stream terminators like `max` return an "optional" result because the stream might be empty or not contain the result:

```
// print largest multiple of 10 in list
// (does not compile!)
int largest =
    IntStream.of(55, 20, 19, 31, 40, -2, 62, 30)
        .filter(n -> n % 10 == 0)
        .max();
System.out.println(largest);
```

Optional results fix

- To extract the optional result, use a "get as" terminator.
 - Converts type OptionalInt to Integer

```
// print largest multiple of 10 in list
// (this version compiles and works.)
int largest =
    IntStream.of(55, 20, 19, 31, 40, -2, 62, 30)
        .filter(n -> n % 10 == 0)
        .max()
        .getAsInt();
System.out.println(largest);
```

Stream exercises

- Write a method `sumAbsVals` that uses stream operations to compute the sum of the absolute values of an array of integers. For example, the sum of `{-1, 2, -4, 6, -9}` is 22.
- Write a method `largestEven` that uses stream operations to find and return the largest even number from an array of integers. For example, if the array is `{5, -1, 12, 10, 2, 8}`, your method should return 12. You may assume that the array contains at least one even integer.

Closures (19.4)

- **bound/free variable:** In a lambda expression, parameters are bound variables while variables in the outer containing scope are free variables.
- **function closure:** A block of code defining a function along with the definitions of any free variables that are defined in the containing scope.

```
// free variables: min, max, multiplier
// bound variables: x, y
int min = 10;
int max = 50;
int multiplier = 3;
compute((x, y) -> Math.max(x, min) *
        Math.max(y, max) * multiplier);
```

Streams and arrays

- An array can be converted into a stream with `Arrays.stream`:

```
// compute sum of absolute values of even ints
int[] numbers = {3, -4, 8, 4, -2, 17,
                 9, -10, 14, 6, -12};
int sum = Arrays.stream(numbers)
    .map(n -> Math.abs(n))
    .filter(n -> n % 2 == 0)
    .distinct()
    .sum();
```


Method references

ClassName :: methodName

- A method reference lets you pass a method where a lambda would otherwise be expected:

```
// compute sum of absolute values of even ints
int[] numbers = {3, -4, 8, 4, -2, 17,
                 9, -10, 14, 6, -12};
int sum = Arrays.stream(numbers)
    .map(Math::abs)
    .filter(n -> n % 2 == 0)
    .distinct()
    .sum();
```

Streams and lists

- A collection can be converted into a stream by calling its `stream` method:

```
// compute sum of absolute values of even ints
ArrayList<Integer> list =
    new ArrayList<Integer>();
list.add(-42);
list.add(-17);
list.add(68);
list.stream()
    .map(Math::abs)
    .forEach(System.out::println);
```

Streams and strings

```
// convert into set of lowercase words
List<String> words = Arrays.asList(
    "To", "be", "or", "Not", "to", "be");
Set<String> words2 = words.stream()
    .map(String::toLowerCase)
    .collect(Collectors.toSet());
System.out.println("word set = " + words2);
```

output:

```
word set = [not, be, or, to]
```

Streams and files

```
// find longest line in the file
int longest = Files.lines(Paths.get("haiku.txt"))
    .mapToInt(String::length)
    .max()
    .getAsInt();
```

stream operations:

```
Files.lines -> ["haiku are funny",
    "but sometimes they don't make sense",
    "refrigerator"]
-> mapToInt -> [15, 35, 12]
    -> max -> 35
```

Stream exercises

- Write a method **fourLetterWords** that accepts a file name as a parameter and returns a count of the number of unique lines in the file that are exactly four letters long. Assume that each line in the file contains at least one word.
- Write a method using streams that finds and prints the first 5 perfect numbers. (Recall a perfect number is equal to the sum of its unique integer divisors, excluding itself.)