Building Java Programs
Chapter 19

Functional Programming with Java 8

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 the same?
  - 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;
  }
  ```

Add/multiply tutor

- Consider a program that gives addition and multiplication quiz problems to the user:

```
9 + 6 = 15
you got it right
3 * 7 = 18
incorrect...the answer was 21
```

- How do we generalize the idea of "add or multiply"?
  - 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:

```
Scanner console = new Scanner(System.in);

// quiz the user on 3 addition problems
giveProblems(console, 3, "+", (x, y) -> x + y);

// quiz the user on 3 multiplication problems
giveProblems(console, 3, "*", (x, y) -> x * y);
```

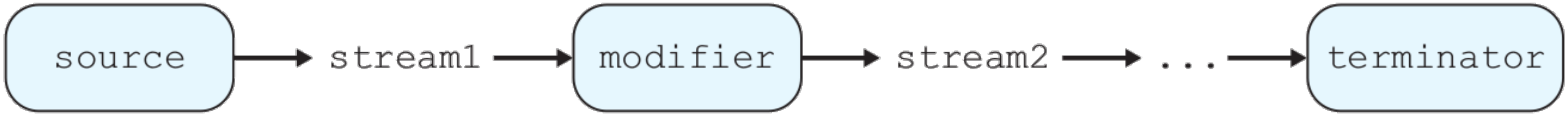
giveProblems method

```
public static void giveProblems(Scanner console, int count,
        String text, IntBinaryOperator operator) {
    Random r = new Random();
    int numRight = 0;
    for (int i = 1; i <= count; i++) {
        int x = r.nextInt(12) + 1;
        int y = r.nextInt(12) + 1;
        System.out.print(x + " " + text + " " + y + " = ");
        int answer = operator.applyAsInt(x, y);
        int response = console.nextInt();
        if (response == answer) {
            System.out.println("you got it right");
            numRight++;
        } else {
            System.out.println("incorrect...the answer was "
                               + answer);
        }
    }
    System.out.println(numRight + " of " + count + " correct");
    System.out.println();
}
```

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:



source → stream1 → modifier → stream2 → . . . → terminator

  - 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.

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

Stream operators

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

Static method	Description
concat(**s1**, **s2**)	glues two streams together
empty()	returns a zero-element stream
iterate(**seed**, **f**)	returns an infinite stream with given start element
of(**values**)	converts the given values into a stream
range(**start**, **end**)	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 **pigLatin** that uses stream operations to convert a String parameter into its "Pig Latin" form. For example, if the string passed is "go seattle mariners", return "o-gay eattle-say ariners-may".

- 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.