CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 12

Divya Chaudhary

Northeastern University
Khoury College of
Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Course logistics
- Functional programming in Java
  - Functional Java – motivation
  - Stream in Java
  - Lambdas in Java
  - Intermediate and terminal operations

COURSE LOGISTICS

- Final Exam – May 2^{nd}.

FUNCTIONAL PROGRAMMING IN JAVA

CS 5004, SPRING 2022– LECTURE 12

JAVA FUNCTIONAL PROGRAMMING

- Key concepts:
  - Functions as first-class objects
  - Pure functions
  - Higher order functions
  - No state
  - No side effect
  - Immutable variables
  - Recursion favored over looping
  - Functional interfaces

JAVA FUNCTIONAL PROGRAMMING

- Functions as first-class objects:
  - We can create an instance of a function
  - We can have a variable referencing to a function
  - Functions can be passed as arguments to other functions
- Note: ordinarily, methods in Java are not first-class objects, but lambda expressions come very close

- Pure functions:
  - The execution of a function has no side effects
  - The return value of a function depends only on input arguments

JAVA FUNCTIONAL PROGRAMMING

- Higher order functions:
  - The function takes one or more functions as input arguments, or
  - The function returns another function as result
- Note: In Java, the closest we can get to a higher order function is a function that takes one a lambda expression as a parameter, and/or returns another lambda expression
- No state external to a function:
  - A method may have local variables containing temporary information, but it cannot reference any member variable of a class or object that it belongs to

JAVA FUNCTIONAL PROGRAMMING

- No side effects:
  - A function cannot change any state outside of that function
- Immutable variables
- Recursions favored over looping
- Functional interfaces
  - An interface that has only one abstract method (i.e., a method that is not implemented on an interface itself)

TERMINOLOGY

- procedural programming
- object-oriented programming
- generic programming
- functional programming
- declarative programming
- imperative programming
- stream
- lambda, lambda expression
- immutability
- concurrency
- reduction
- external vs internal iteration
- terminal operation
- arrow token
- lazy evaluation
- eager
- method reference
- infinite streams

STREAMS IN JAVA

CS 5004, SPRING 2022– LECTURE 12

ACKNOWLEDGEMENT

Notes adapted from Dr. Adrienne Slaughter. Thank you.

STREAMS – HIGH LEVEL IDEA

- Start with a stream of data (primitive or objects)
- Apply a series of operations or transformations to the stream
- Reduce the stream to a single number or collect the stream to collection

HOW MANY TIMES HAVE YOU WRITTEN A CODE LIKE THIS?

```
List<Record> records = new ArrayList<>();

int total = 0;

for (int i=0; i<records.size(); i++){
    total += records.get(i).value();
}
```

HOW MANY TIMES HAVE YOU WRITTEN A CODE LIKE THIS?

```
List<Record> records = new ArrayList<>();

int total = 0;

for (int i=0; i<records.size(); i++){
    total += records.get(i).value();
}
```

What could go wrong?

HOW MANY TIMES HAVE YOU WRITTEN A CODE LIKE THIS?

```
List<Record> records = new ArrayList<>();

int total = 0;

for (int i=0; i<records.size(); i++){
    total += records.get(i).value();
}
```

External Iteration:
The programmer specifies the iteration details

LET'S SIMPLIFY OUR EXAMPLE PROBLEM A BIT

```
int total = 0;

for (int i=0; i<10; i++){
    total += i;
}
```

LET'S SIMPLIFY OUR EXAMPLE PROBLEM A BIT

```
int total = 0;

for (int i=0; i<10; i++){
    total += i;
}
```

```
int total = IntStream.rangeClosed(1, 10)
                     .sum();
```

LET'S SIMPLIFY OUR EXAMPLE PROBLEM A BIT

```
int total = 0;

for (int i=0; i<10; i++){
    total += i;
}
```

"For the stream of ints from 1 to 10, calculate the sum."

```
int total = IntStream.rangeClosed(1, 10)
                     .sum();
```

STREAMS AND STREAM PIPELINE

Adapted from: https://stackoverflow.com/questions/1216380/what-is-a-stream

- Stream: sequence of elements
- Stream pipeline: sequence of tasks ("processing steps") applied to elements of a stream
- A stream starts with a data source
  - Examples:
    - Terminal I/O
    - Socket I/O
    - File I/O
- A stream can generally be used like a queue– you're reading from it, but you can't go back in the stream
- Once you've pulled an element off the stream, it's no longer in the stream

THE STREAM

```
int total = IntStream.rangeClosed(1, 10).sum();
```

IntStream produces a stream of integers in the given range

rangeClosed is closed– produces ints including 1 and 10

THE STREAM PIPELINE

```
int total =
IntStream.rangeClosed(1, 10)
                        .sum();
```

The processing step to take, or task to complete using the stream

THE STREAM PIPELINE

```
int total =
IntStream.rangeClosed(1, 10)
                        .sum();
```

The processing step to take, or task to complete using the stream

Reduction:

Reduces the stream of values into a single value

THE STREAM PIPELINE

```
int total =
IntStream.rangeClosed(1, 10)
                            .sum();
```

The processing step to take, or task to complete using the stream

Internal Iteration:

IntStream handles all the iteration details– we don't write them ourselves

Reduction:

Reduces the stream of values into a single value

THE STREAM PIPELINE

Declarative Programming:

Internal Iteration: IntStream handles all the iteration details– we don't write them ourselves.

Imperative Programming:

External Iteration: The programmer specifies the iteration details.

THE STREAM PIPELINE

Declarative Programming:
Specify what to do

Internal Iteration: IntStream handles all the iteration details– we don't write them ourselves

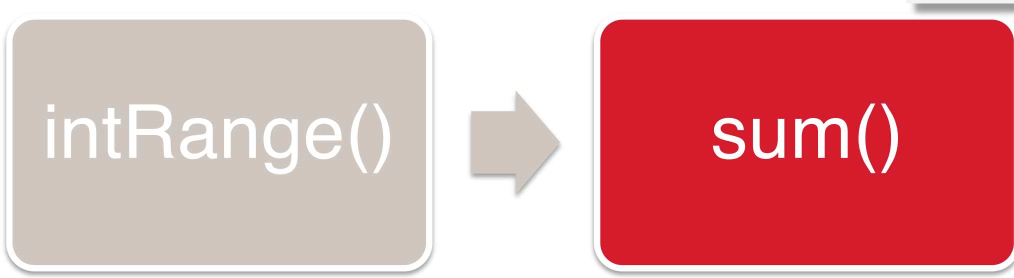
Imperative Programming:
Specify how to do something

External Iteration: The programmer specifies the iteration details.

THE STREAM PIPELINE – EXAMPLE 2

```
int total =
IntStream.rangeClosed(1, 10)
                       .sum();
```

But what if we want to sum the even numbers between 2 and 20?



EXAMPLE 2: SUMMING EVEN INTEGERS FROM 2-20

```
int total = IntStream.rangeClosed(1, 10)
                .map((int x) -> {return x * 2;})
                .sum();
```

EXAMPLE 2: SUMMING EVEN INTEGERS FROM 2-20

```
int total = IntStream.rangeClosed(1, 10)
                  .map((int x) -> {return x * 2;})
                  .sum();
```

This converts the stream from 1:10 to 2:20 by multiplying by 2.

intRange() → map() → sum()

LAMBDAS IN JAVA

CS 5004, SPRING 2022– LECTURE 12

METHOD `MAP()`

- map() - takes a method, and applies it to every element in the stream

```
.map((int x) -> {return x * 2;})
```

Wait, what? A *method*?

LAMBDAS: ANONYMOUS METHODS

- lambda or lambda expression
  - aka anonymous method
  - aka method-without-a-name
  - aka the method that shall not be named

```
(int x) -> {return x * 2;}
```

LAMBDAS: ANONYMOUS METHODS

- Methods that can be treated as data
  - pass lambdas as arguments to other methods (map)
  - assign lambdas to variables for later use
  - return a lambda from a method

```
(int x) -> {return x * 2;}
```

LAMBDAS: SYNTAX

```
(parameter list) -> {statements}
```

```
(int x) -> {return x * 2;}
```

Parameter: one int named x

Statement: return 2*x

LAMBDAS: SYNTAX

```
(parameter list) -> {statements}
```

```
(int x) -> {return x * 2;}
```

Same as:

```
int multiplyBy2(int x){
    return x * 2;
}
```

Difference:

- the lambda doesn't have a name
- compiler infers return type

LAMBDAS: SIMPLIFYING SYNTAX

Eliminate parameter type

```
(int x) -> {return x * 2;}
```

```
(x) -> {return x * 2;}
```

Type is inferred.
If it can't be inferred, compiler throws an error.

LAMBDAS: SIMPLIFYING SYNTAX

Simplify the body

```
(x) -> {return x * 2;}
```

```
(x) -> x * 2
```

- return is inferred
- semicolon and brackets not necessary

LAMBDAS: SIMPLIFYING SYNTAX

Simplify parameter list

```
(x) -> x * 2
```

```
x -> x * 2
```

We can remove parentheses for single parameter

LAMBDAS: SIMPLIFYING SYNTAX

lambda without parameters

```
() -> System.out.println("Hello Lambda!")
```

LAMBDAS: SIMPLIFYING SYNTAX

method references

```
.map(x -> System.out.println(x))
```

```
.map(System.out::println)
```

```
objectName::instanceMethodName
```

Sometimes, you want to just pass the incoming parameter to another method

LAMBDAS: SCOPE

- Lambdas do not have their own scope
  - We cannot shadow a method's local variable with lambda parameters with the same name
  - Lambdas share scope with the enclosing method

INTERMEDIATE AND TERMINAL OPERATIONS

CS 5004, SPRING 2022– LECTURE 12

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
                .map((int x) -> {return x * 2;})
                .sum();
```

- map() is an intermediate operation
- sum() is a terminal operation

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
            .map((int x) -> {return x * 2;})
            .sum();
```

- map() is an intermediate operations
- sum() is a terminal operation

Intermediate operations use lazy evaluation

The operation produces a new stream object, but no operations are performed on the elements until the terminal operation is called to produce a result

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
                     .map((int x) -> {return x * 2;})
                     .sum();
```

- map() is an intermediate operations
- sum() is a terminal operation

Terminal operations use are eager.

The operation is performed when called.

EXAMPLES

Intermediate Operations

- `filter()`
- `distinct()`
- `limit()`
- `map()`
- `sorted()`

Terminal Operations

- `forEach()`
- `collect()`

Reductions:

- `average()`
- `count()`
- `max()`
- `min()`
- `reduce()`

BACK TO OUR EXAMPLE 2…

```
int total = IntStream.rangeClosed(1, 10)
                     .map((int x) -> {return x * 2;})
                     .sum();
```

For this example, we chose to create a stream of event ints from 2 to 20 by mapping from 1:10, multiplying by 2.

How else can we do this?

BACK TO OUR EXAMPLE 2…

```
int total = IntStream.rangeClosed(1, 20)
                     .filter(x -> x%2 == 0)
                     .sum();
```

Filter!

The lambda for the filter operation needs to return a boolean indicating whether the given element should be in the output stream.

CLARIFYING ELEMENTS THROUGH A PIPELINE

```
int total = IntStream.rangeClosed(1, 10)
    .filter(
        x -> {
          System.out.printf("%nFilter: %d%n", x);
          return x % 2 == 0;
        })
     .map(
         x -> {
           System.out.printf("map: %d", x);
           return x * 3;
         }
     )
    .sum();
 System.out.println("\n\nTotal: " +total);
```

CLARIFYING ELEMENTS THROUGH A PIPELINE

```
int total = IntStream.rangeClosed(1, 10)
    .filter(
        x -> {
          System.out.printf("%nFilte
          return x % 2 == 0;
        })
     .map(
         x -> {
           System.out.printf("map: %d"
           return x * 3;
         }
     )
    .sum();
 System.out.println("\n\nTotal: "
```

```
Filter: 1

Filter: 2
map: 2
Filter: 3

Filter: 4
map: 4
Filter: 5

Filter: 6
map: 6
Filter: 7

Filter: 8
map: 8
Filter: 9

Filter: 10
map: 10

Total: 90
```

COLLECTORS

CS 5004, SPRING 2022– LECTURE 12

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

Returns the number of elements in the stream.

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - **Collectors.joining()**
  - Collectors.toList()
  - Collectors.groupingBy()

Joins the elements of the stream together into a String, with a specified delimiter

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - **Collectors.toList()**
  - Collectors.groupingBy()

Puts the elements of the stream into a List<> and returns it.

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

Groups the elements in the stream according to some parameter and returns a HashMap keyed by the "groupingBy" parameter.

ANOTHER TERMINAL: FOREACH

- forEach() applies the given method to each element of the stream
- The method must receive one argument and return void

METHOD REDUCE ()

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
                .reduce(1, (x, y) -> x * y);
```

METHOD REDUCE()

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
               .reduce(1, (x, y) -> x * y);
```

The starting value.
This is the value for reduce(0)

METHOD REDUCE ()

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
                .reduce(1, (x, y) -> x * y);
```

The operation to perform.
Must take 2 parameters.
(Because it takes 2 params, we need to use the parens in the lambda)

PRODUCING A STREAM FROM AN ARRAY

```
int total = IntStream.of(someInts)
                .sum();
```

PRODUCING A STREAM FROM A COLLECTION

```
List<String> strings = new ArrayList<>();
strings.stream();
```

CREATING A STRING FROM AN ARRAY

```
String out = IntStream.of(someInts)
             .mapToObj(String::valueOf)
             .collect(Collectors.joining(" "));
```

Here, the mapToObj() operator is new.

It uses the specified method to convert the input element to a new type.

USING LINES IN FILES AS A STREAM

```
Files.lines(Paths.get("src/main/resources/OODAssignment.csv"))
```

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any
other words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                             .map(line ->
splitAtSpaces.splitAsStream(line))
                             .collect(Collectors.toList());
```

What is the type of list after this is run?
How many elements are in the list?
4 elements in the final list.
(one for each entry in someStrings)

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any
other words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                            .map(line ->
splitAtSpaces.splitAsStream(line))
                            .collect(Collecto
```

```
Pattern splitAtSaces = Pattern.compile("\\s+")
String someStrings[] = {"one row", "some more
other words", "and once upon a time"};
Object list =  Stream.of(someStrings)
                            .flatMap(line ->
splitAtSpaces.splitAsStream(line))
                            .collect(Collectors.toList());
```

What is the type of list after this is run?
How many elements are in the list?
4 elements in the final list.
(one for each entry in someStrings)

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any other
words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                             .flatMap(line ->
splitAtSpaces.splitAsStream(line))
                             .collect(Collectors.toList());
```

When I really want 13 items in the final list (one for every word in the original input), I use `flatMap()`.
When the output of a map() is a collection, `flatMap()` flattens the result by adding all the items in the output to the stream individually, rather than as a collection.

FUNCTIONAL PROGRAMMING SO FAR - SUMMARY

- Stream that gets mapped, filtered, reduced, and collected… in some order
  - Intermediate operations are not executed until a terminal operation is called
- Lambdas: unnamed methods (functions) that can be applied to a stream
- Declarative vs. imperative

FUNCTIONAL PROGRAMMING SO FAR - SUMMARY

- A tenet of functional programming is immutability
  - An object is not mutable– it can't change
  - Rather than change state (mutate it), create a new copy with the new state
  - Helps with concurrency

FUNCTIONS AS OBJECTS

CS 5004, SPRING 2022– LECTURE 12

FUNCTIONAL INTERFACES

- Introduced in Java 8
  - An interface that contains only a single abstract (unimplemented) method
- Example 1:

```
public interface MyFunctionalInterface {
    public void run();
    }
```

- Example 2:

```
public interface MyFunctionalInterface2{
    public void execute();
    public default void print(String text) {
        System.out.println(text);
  }
}
```

FUNCTIONAL INTERFACES

- Can be implemented by lambda expressions

- Example 3:

```
MyFunctionalInterface lambda = () ->
 {System.out.println("Executing...");
}
```

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java

1. Function (**`java.util.function.Function`**) – represents a function that takes a single parameter, and returns a single value

```
public interface Function<T,R> {
   public <R> apply(T parameter); }
```

2. Predicate (**`java.util.function.Function`**) – represents a simple function that takes a single value parameter, and returns true or false

```
public interface Predicate {
 boolean test(T t); }
```

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java

3. UnaryOperator – represents an operation which takes a single parameter, and returns a parameter of the same type
   - It can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it again

4. BinaryOperator– represents an operation that takes two parameters and returns a single value, where both parameters and return value have to be of the same type

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java
  3. Supplier – represents an operation that supplies a value of some sort
     - This interface can be thought of as a factory interface

  4. Consumer – represents a function that consumers a value without returning any value

FUNCTIONS AS OBJECTS

- Let's consider functional interface **`Function`** again

```
public interface Function<T,R> {
   public <R> apply(T parameter); }
```

- The given interface can be implemented with a function object:

```
Function<T, R> functionName = {t → operation returning R}
```

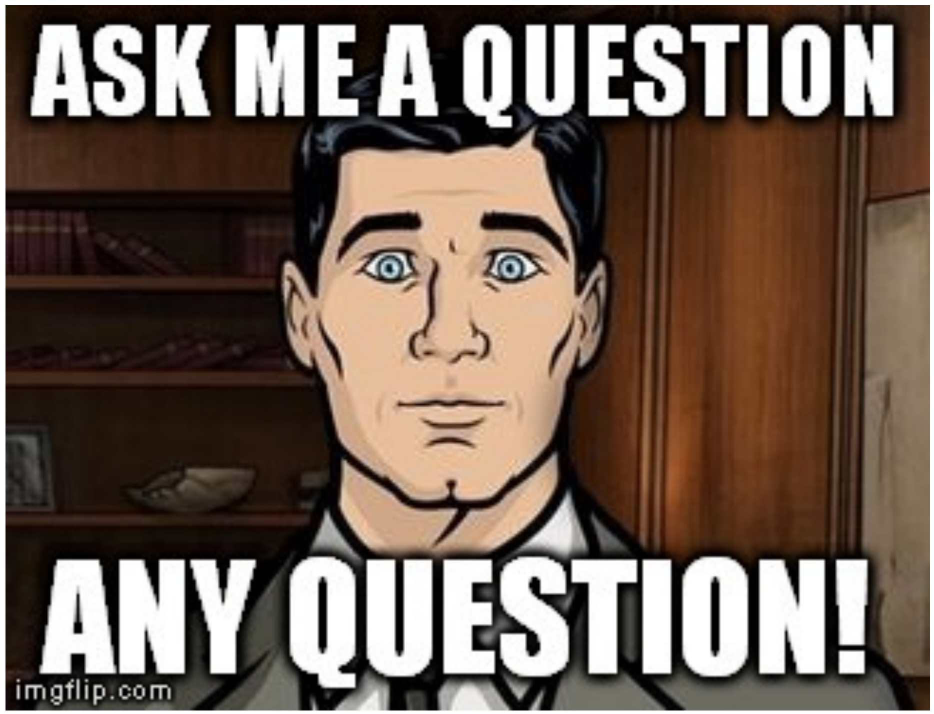
- Calling a function object:

```
T oldObject = new T();
R newObject - functionName.apply(oldObject);
```

USE OF FUNCTIONS AS OBJECTS

- To tidy up stream operations
- If we have a higher-order method, and we need to pass a function as a parameter to it
- If we want a function to be accessible to only one object (e.g., even listeners)
- A design choice

YOUR QUESTIONS



[Meme credit: imgflip.com]