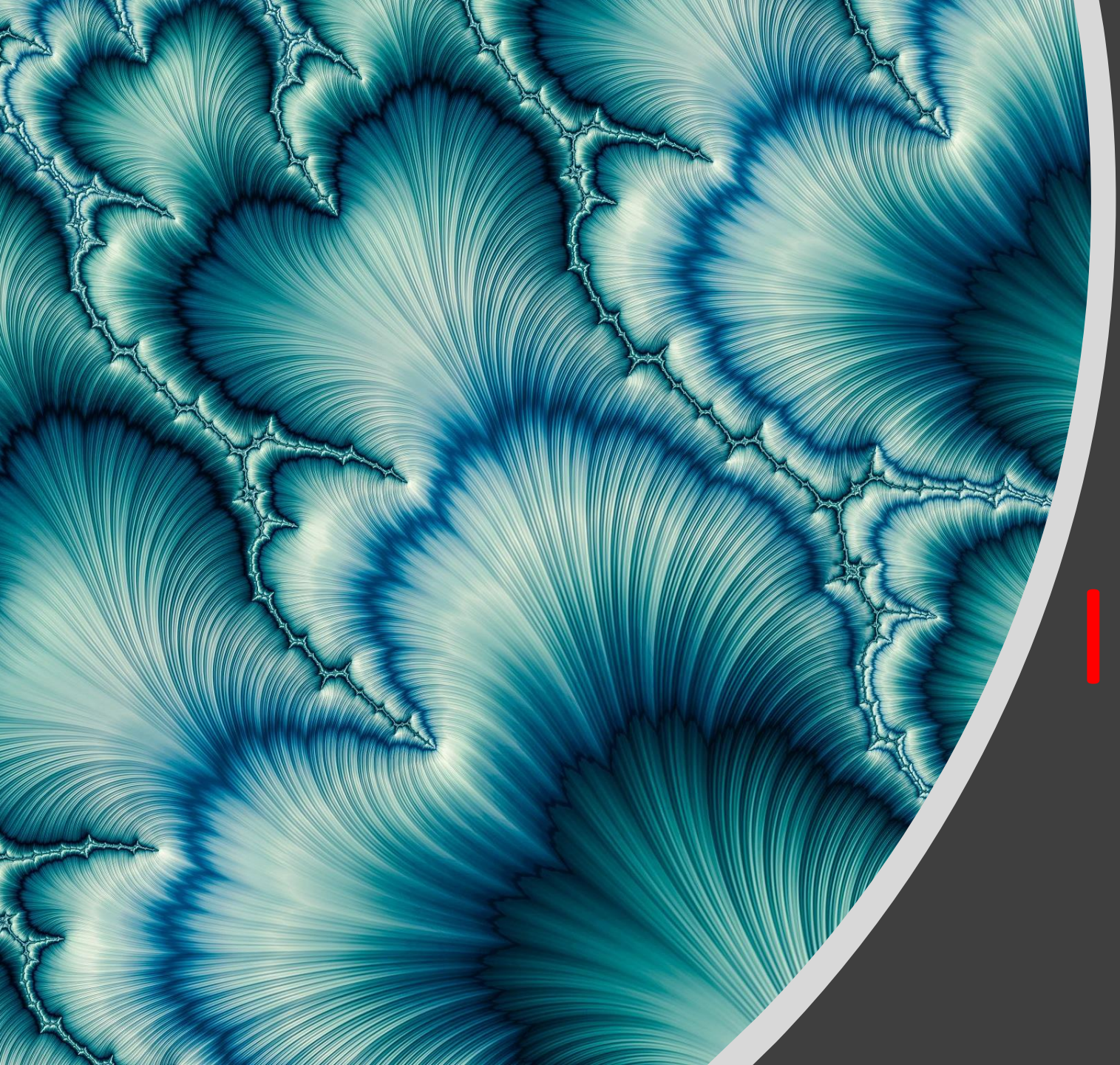
Lecture 12: Functional Programming Spring 2022

Instructor: Brian Cross

Lots of credit: Divya Chaudhary, Adrienne Slaughter

Lecture Agenda

- Functional Programming
  - Programming Paradigms
  - Motivation
  - Terminology



Introduction

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

Functional Programming: What we do?

1

Start with a **stream** of data (primitive or objects)

2

Apply a series of operations or transformations to the stream

3

Reduce the stream to a single number or **collect** the stream to collection

What's a stream, and is a list a stream?

An array?

A hashmap?

How many times have you written 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 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 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 for a moment.

```
int total = 0;

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

Let’s simplify for a moment.

```
int total = 0;

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

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

Let's simplify for a moment.

```
int total = 0;

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

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

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

Stream and Stream Pipeline

- 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.

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

Imperative Programming:

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

External Iteration:
The programmer specifies the iteration details.

The Stream Pipeline

Declarative Programming:
Specifies *what* to do

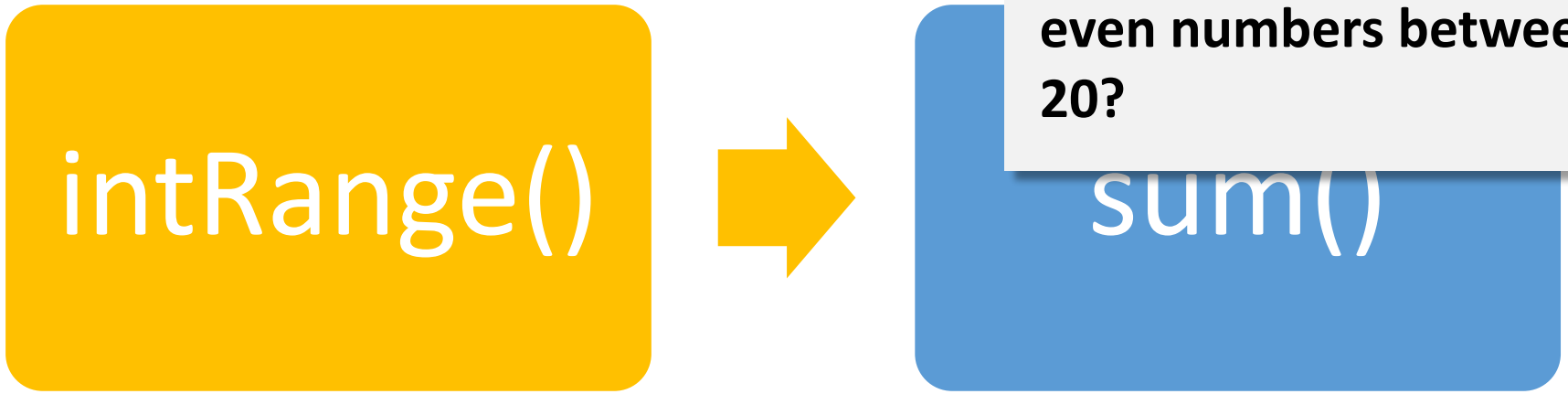
Imperative Programming:
Specifies *how* to do something.

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

External Iteration:
The programmer specifies the iteration details.

The Stream Pipeline

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



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

Summing even ints 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.

.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;
}
```

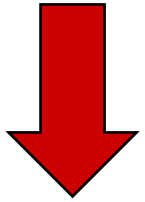
Difference:

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

lambdas: simplifying syntax

Eliminate parameter type

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



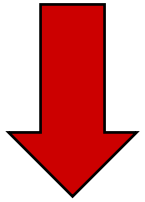
```
(x) -> {return x * 2;}
```

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

lambdas: simplifying syntax

Simplify the body

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



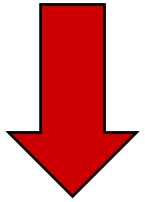
```
(x) -> x * 2
```

- return is inferred
- semicolon and brackets not necesary

lambdas: simplifying syntax

Simplify parameter list

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



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

Can remove parens for single parameter

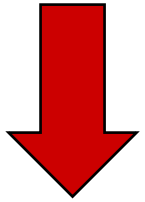
lambdas: simplifying syntax

lambda with no parameters

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

Method Referenes

```
.map(x -> x.toUpperCase())
```



```
.map(String::toUpperCase)
```

```
objectName::instanceMethodName
```

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

lambdas: scope

- Lambdas do not have their own scope
  - Can’t shadow a method’s local variable with lambda params with the same name
  - Lambdas share scope with the enclosing method

Fun Note:

- Why is it called lambda?
  - Alonzo Church: Church's λ calculus
    - Developed to provide a rigorous foundation for studying functions and function application

Stream Pipeline: Intermediate & 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 & 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

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 & 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

Terminal operations 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…

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

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

How else can we do this?

Back to our example…

```
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 the 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 the 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);
```

```
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

- 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`.

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);
```

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)`

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)

Miscellaneous tid-bits

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 a file as a stream

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

Returns a **Stream<String>**

…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(Collectors.toList());
```

```
list => [["one", "row"], ["some", "more", "words"], ["any", "other", "words"], …]
```

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.

…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());
```

```
list => ["one", "row", "some", "more", "words", "any", "other", "words", …]
```

Immutability

- 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

Applying of this to Objects, not just primitive Types

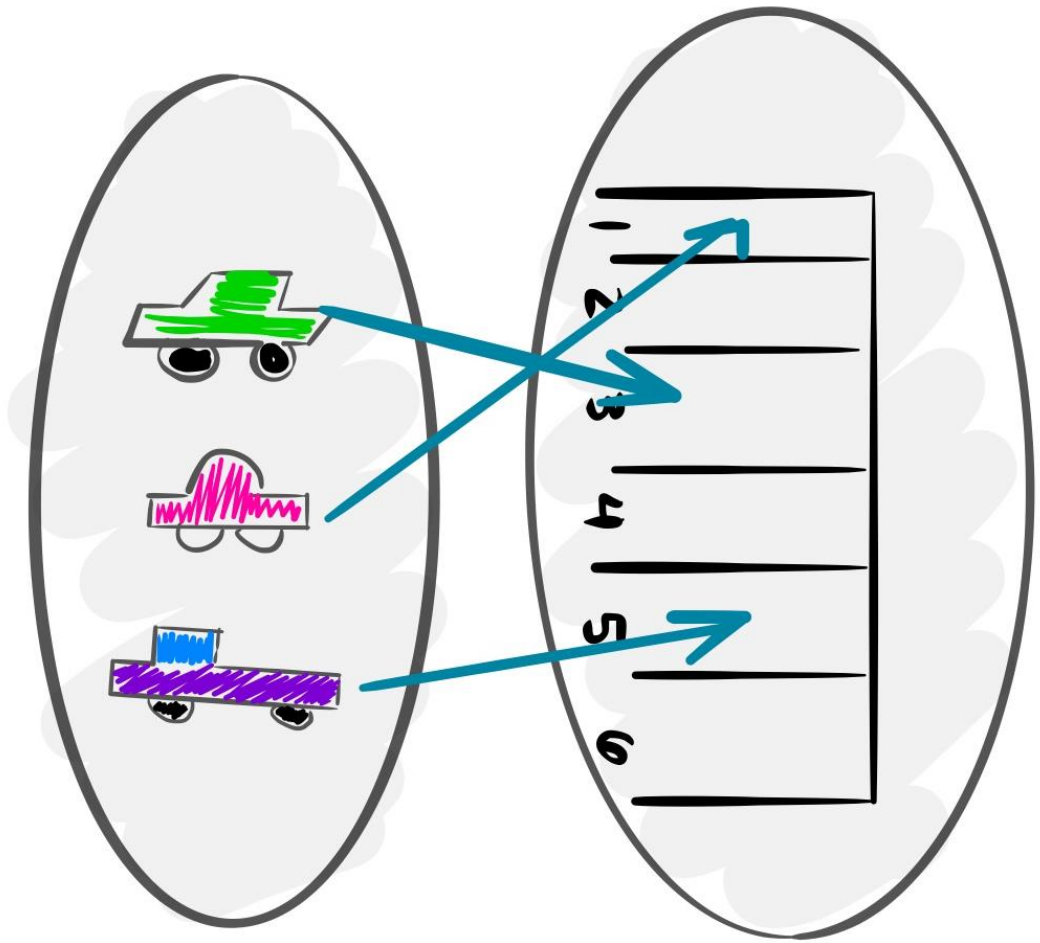
Functional Programming in Java

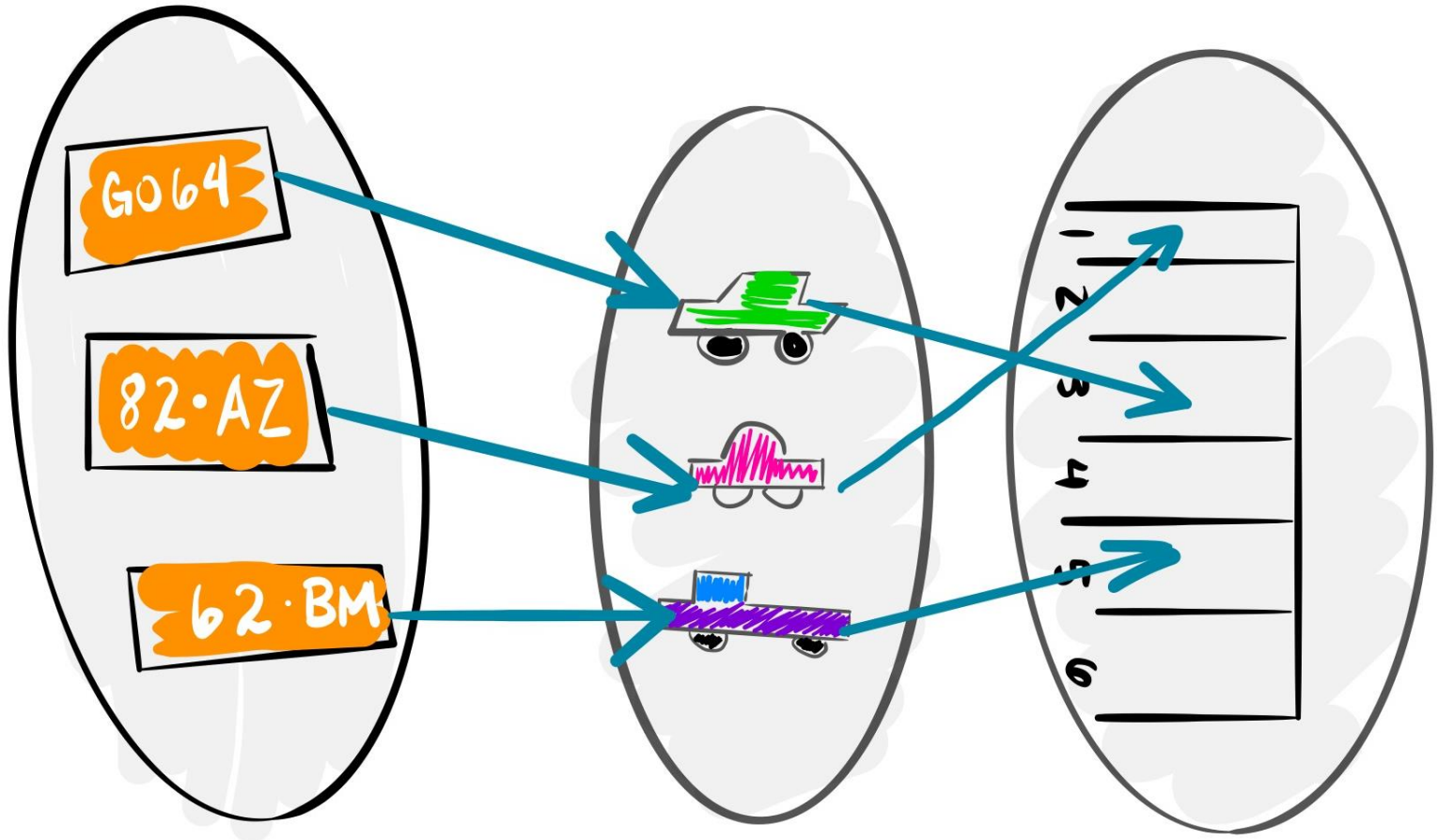
- Stream that gets mapped, filtered, reduced, and collected… in some order.
  - Intermediate operations are not executed until a terminal operation is called.
- Lambdas: unnamed methods (functions) that can be applied to a stream
- Declarative vs. imperative

Backing up a bit…

What is a function?

- All elements in the domain are mapped to a value in the codomain
- There cannot exist elements in the domain w/ no image in the codomain
- A single element in the domain can only map to one element in the codomain


1
2
3
4
5
6


G064
82·AZ
62·BM
1
2
3
4
5
6

A function ***IS***.
It doesn’t ***DO***.

Important Concepts in Functional Programming

- First class functions
- Anonymous functions
- Closures
- Currying
- Lazy evaluation
- Referential transparency

What does it mean to have no side effects?

- No mutation of variables
- No printing to the console or devices
- No writing to files, databases, networks, …
- No exception throwing

(really, this is “no intentional side effects”)

Functional Programming Benefits

- Easier to reason about b/c they are deterministic
- Easier to test– no side effects
- More modular
  - Functions have an input and output; no side effects, concurrent modifications, etc…
- Composition and recombination easier
- Inherently thread-safe

Referential Transparency

Properties:

- Self-contained
- Deterministic
- Will never throw an Exception
- Won’t cause conditions that cause other code to fail
- Won’t hang due to an external device

Using substitution to reason about code

3 * 2	+	4 * 5
6	+	4 * 5
6	+	20

Using substitution to reason about code

```
public static void main(String[] args) {
  int x = add(mult(2, 3), mult(4, 5));
}

public static int add(int a, int b) {
  log(String.format("Returning %s as the result of %s + %s", a + b , a , b));
  return a + b;
}

public static int mult(int a, int b) {
  return a * b;
}

public static void log(String s) {
  System.out.println(s);
}
```

What happens if I instead call:

```
int x = (26);
```

An example

```
public class DonutShop {
  public static Donut buyDonut(CreditCard creditCard) {
    Donut donut = new Donut();
    creditCard.charge(Donut.price);
    return donut;
  }
}
```

Can you see the side effect?

Create a Payment

```
public class Payment {
  public final CreditCard creditCard;
  public final int amount;

  public Payment(CreditCard card, int amt) {
    this.creditCard = card;
    this.amt = amount;
  }
}
```

Create a Purchase

```
public class Purchase {
  public Donut donut;
  public Payment payment;

  public Purchase(Donut donut, Payment payment) {
    this.donut = donut;
    this.payment = payment;
  }
}
```

Modify buyDonut

```
public static Purchase buyDonut(CreditCard creditCard) {
  Donut donut = new Donut();
  Payment payment = new Payment(creditCard, Donut.price);
  return new Purchase(donut, payment);
}
```

Functional programming replaces side effects with returning a representation of the effects

Pure functions: functions without side effects

First-class functions: can be passed around as data/values.

Simplifying functions by Currying

- In some scenarios, you may want to simplify functions to only take a single parameter
- Some languages favor curried functions to achieve multiple parameters
  - ML and Haskell
  - Lamda calculus
- Simplifies your functions
  - Allows scalability

Currying

- Let's start with $f(x, y) = x + y$
  - This is really a function of 1 variable: $f((x, y))$
- What if I have $f(x)(y) = g(y)$?
  - $g(y) = x + y$
- Then, $f(x) = g$
  - The result of applying function *f* to *x* is a new function *g*
  - Then $g(y) = x + y$
- $f(x)(y)$ is the curried version of $f(x, y)$

Add 2 numbers

```
int add (int a, int b) {
      return a + b;
   }
```

As a lambda:

```
(a, b) -> a + b;
```

Curried:

```
(a) -> ((b) -> a + b);
a -> b -> a + b;
```

Functional Interface: Function<T, R>

T – Type of the input to the function

R – Type of the result of the function

Modifier and Type	Method and Description
default <V> **Function**<**T**,V>	**andThen**(**Function**<? super **R**,? extends V> after) Returns a composed function that first applies this function to its input, and then applies the after function to the result.
R	**apply**(**T** t) Applies this function to the given argument.
default <V> **Function**<V,**R**>	**compose**(**Function**<? super V,? extends **T**> before) Returns a composed function that first applies the before function to its input, and then applies this function to the result.
static <T> **Function**<T,T>	**identity**() Returns a function that always returns its input argument.

Add 2 numbers

```
Function<Integer, Function<Integer, Integer>> add =   (a) -> (b) -> a + b;


Int result = add
.apply(5)
.apply(6);
```

Functional in Java

A method is functional if:

- Does not mutate anything outside the function
- Does not mutate argument
- Does not throw errors or exceptions
- Always returns a value
- When called with the same argument, always returns the same result

Let’s look at some code

Questions?

Finishing up

- UML Design Draft due Monday
- Next Wednesday is our last class
  - Course wrap-up
  - Ask me (almost) anything (AMA)
- Homework #6 due **Sunday** @ 11:59pm