Lecture 10

Working with Streams

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Concordia University

From external iteration to internal iteration

René Witte
Concordia University

Collections API: External Iteration

```
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu){
  if(d.isVegetarian()){
    vegetarianDishes.add(d);
  }
}
```

Streams API: Internal Iteration

```
List<Dish> vegetarianDishes = menu.stream()
                                  .filter(Dish::isVegetarian)
                                  .collect(toList());
```

Outline

René Witte

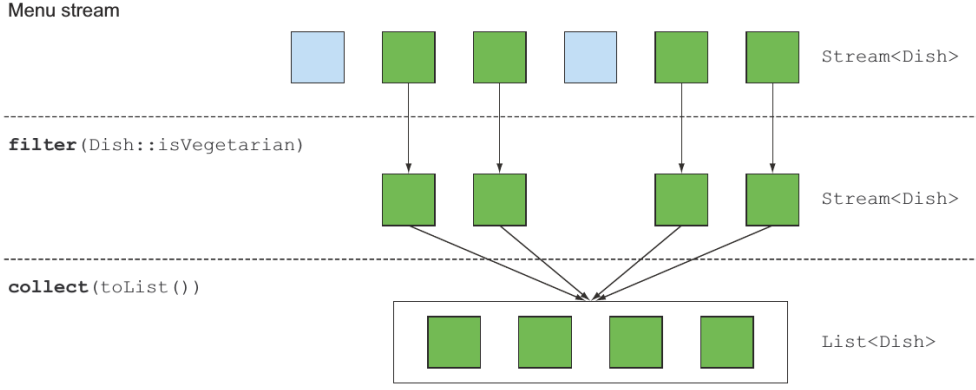
Concordia University

Filtering with a predicate

René Witte
Concordia University

Get all vegetarian friendly dishes

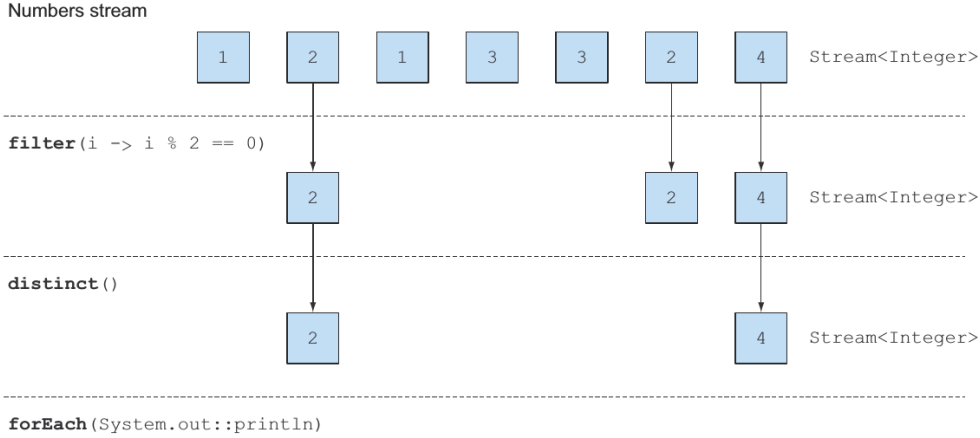
```
List<Dish> vegetarianDishes = menu.stream()
                                  .filter(Dish::isVegetarian)
                                  .collect(toList());
```



Copyright 2015 by Manning Publications Co., [UFM14]

Filtering unique elements

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach(System.out::println);
```

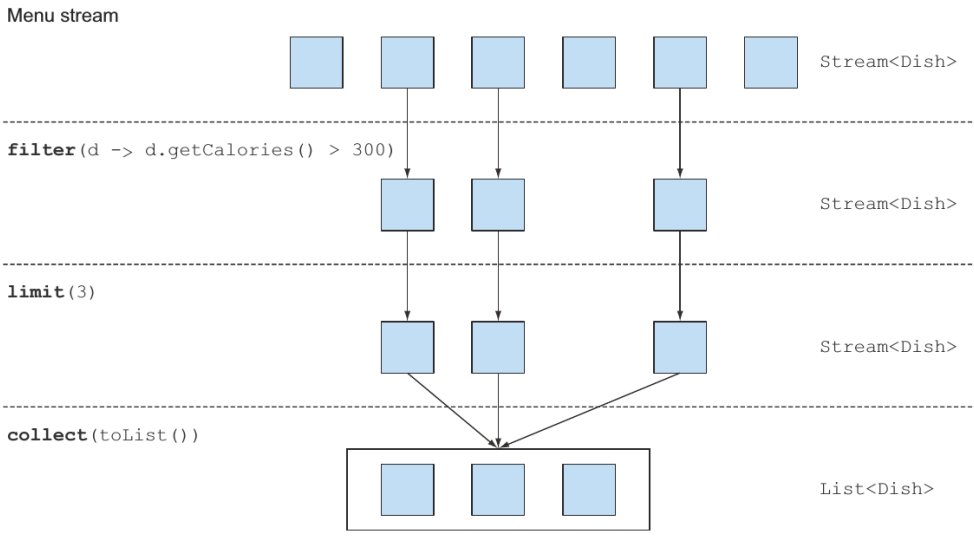


Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

Truncating a stream

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .limit(3)
                        .collect(toList());
```

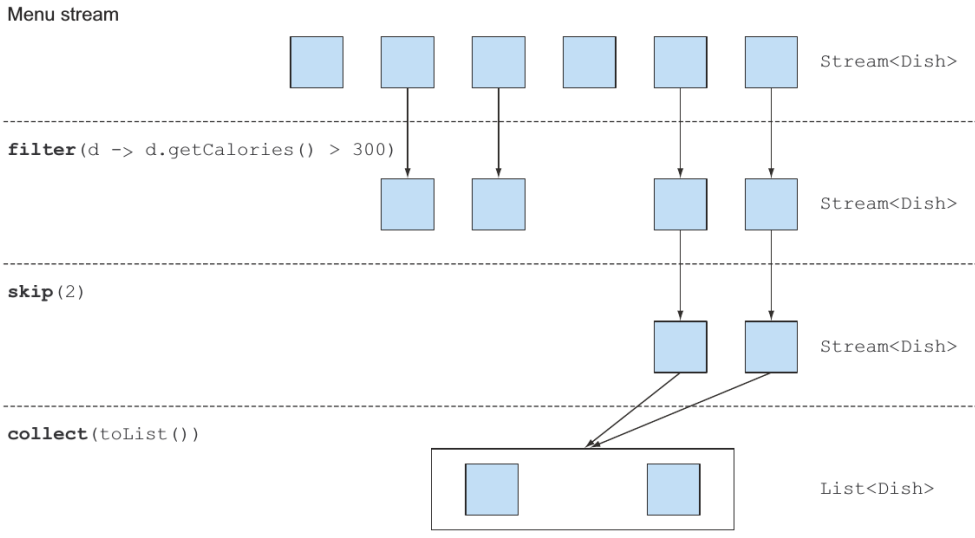


Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

Skipping elements

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```



Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

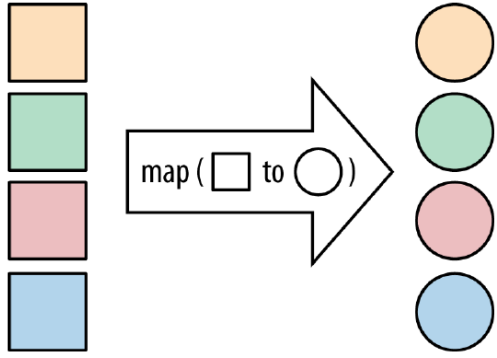
Outline

René Witte

Concordia University

Map: Applying a function to each element of a stream

```
List<String> dishNames = menu.stream()
                             .map(Dish::getName)
                             .collect(toList());
```



Copyright 2014 by Richard Warburton, [War14]

René Witte
Concordia University

Map Examples

René Witte
Concordia University

Return the length of each string in a list

```
List<String> words = Arrays.asList("Java8","Lambdas","In","Action");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
```

Return the length of each name of a dish

```
List<Integer> dishNameLengths = menu.stream()
                                    .map(Dish::getName)
                                    .map(String::length)
                                    .collect(toList());
```

Flattening streams

René Witte

Obtain unique characters for a list of words?

Input: `["Hello", "World"]`

Output: `["H", "e", "l", "o", "W", "r", "d"]`

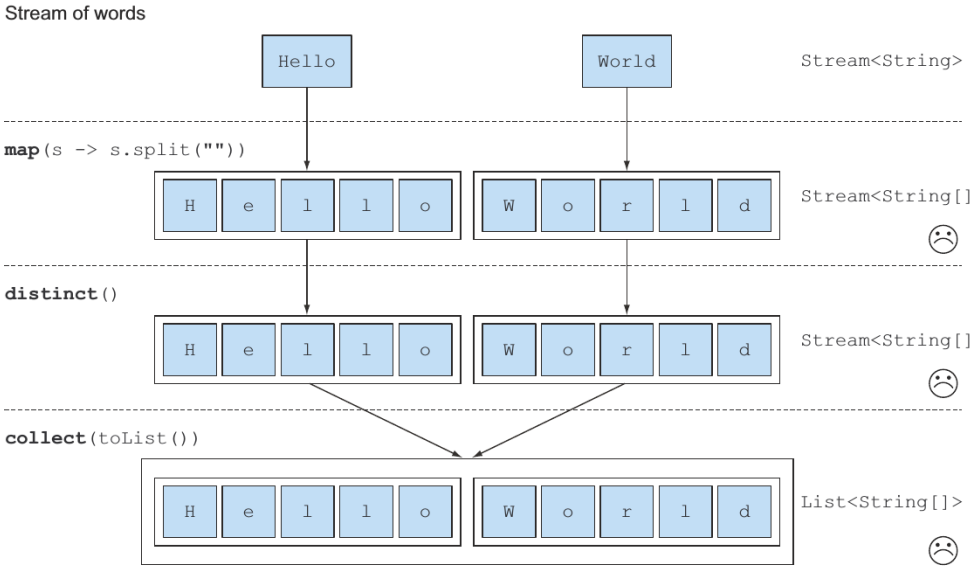
First attempt

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

Mapping from `String` to `String[]`

- here, `map` returns a string array `String[]` for *each* word
- so, the result from `map` is a `Stream<String[]>`
- but what we need is a single string array

Incorrect use of `map`



Copyright 2015 by Manning Publications Co., [UFM14]

René Witte
Concordia University

Using `flatMap`

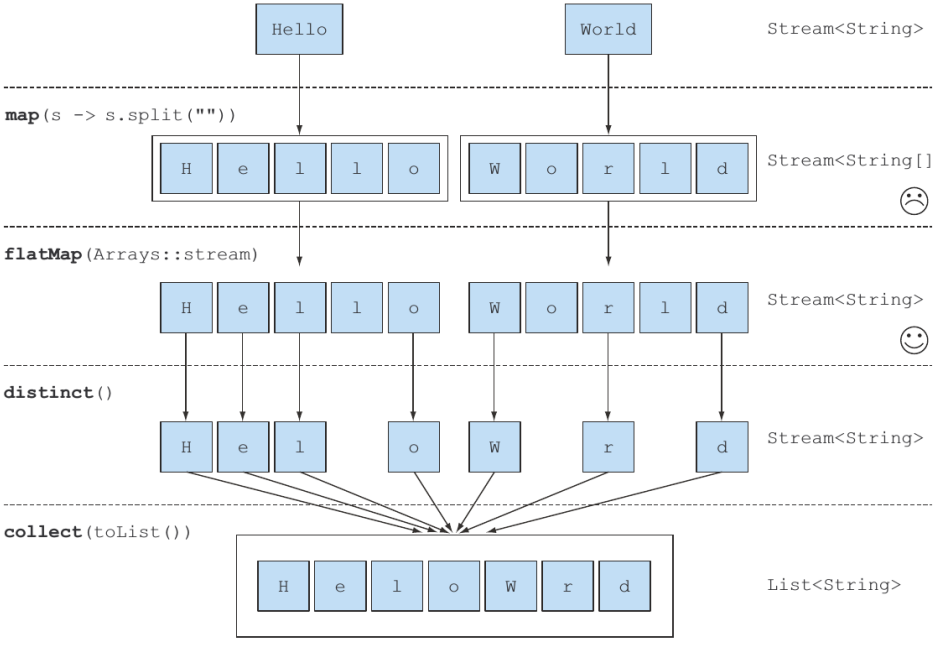
```
List<String> uniqueCharacters = words.stream()
                                     .map(w -> w.split(""))
                                     .flatMap(Arrays::stream)
                                     .distinct()
                                     .collect(Collectors.toList());
```

`flatMap`

- replace a value with a `Stream`
- and concatenate all the streams together

René Witte
Concordia University

Stream of words



Copyright 2015 by Manning Publications Co., [UFM14]

René Witte
Concordia University

Outline

René Witte

Concordia University

Checking to see if a predicate matches at least one element

René Witte
Concordia University

Is there at least one vegetarian dish on the menu?

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
  System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

`anyMatch`

- returns a `boolean`
- terminal operation

Checking to see if a predicate matches all elements

René Witte
Concordia University

Is the menu healthy (all dishes < 1000 calories)?

```
boolean isHealthy = menu.stream()
                        .allMatch(d -> d.getCalories() < 1000);
```

`allMatch`

- returns a `boolean`
- terminal operation

Checking to see if a predicate matches no elements

René Witte

Is the menu healthy (all dishes < 1000 calories)?

```
boolean isHealthy = menu.stream()
                        .noneMatch(d -> d.getCalories() >= 1000);
```

`noneMatch`

- returns a `boolean`
- terminal operation

Short-circuiting

The operations `anyMatch`, `allMatch`, and `noneMatch` are short-circuiting.

Short-circuiting evaluation

René Witte
Concordia University

Evaluating Boolean expressions

Evaluating `(A && B)`

- Check if `A` is `false`.
- If yes, return `false`; otherwise,
- evaluate `B` and return its result.

Similarly, evaluate `(A || B)` as

- Check if `A` is `true`.
- If yes, return `true`; otherwise,
- evaluate `B` and return its result.

This is know as short circuit (or lazy) evaluation.

Non-short circuit operators

The operators `&` and `|`, when applied to `boolean` operands, are non-short circuit (usually applied on `integer` for bitwise and/or operations).

Finding an element

René Witte
Concordia University

Find any dish that's vegetarian

```
Optional<Dish> dish = menu.stream()
                          .filter(Dish::isVegetarian)
                          .findAny();
```

`Optional<T>`

What if there is no vegetarian dish?

- Java 7: return `null`
- Java 8: return `Optional<T>`

Working with `Optional` safer than `null` value checking

Java 8 `Optional<T>`

René Witte
Concordia University

`java.util.Optional`

`isPresent()` returns `true` if `Optional` contains a value, false otherwise

`ifPresent(Consumer<T> block)` executes the given block if a value is present

`T get()` returns the value if present; otherwise it throws a `NoSuchElement` Exception.

`T orElse(T other)` returns the value if present; otherwise it returns a default value.

Example

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(d -> System.out.println(d.getName());
```

Finding the first element

René Witte
Concordia University

Find the first square of a number that's divisible by 3

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
    someNumbers.stream()
               .map(x -> x * x)
               .filter(x -> x % 3 == 0)
               .findFirst();
```

`findFirst` vs. `findAny`

Difference is in parallelism:

- `findAny` can process a stream in parallel
- `findFirst` cannot

Outline

René Witte

Concordia University

Summing the elements in a stream

René Witte
Concordia University

Java 7 Style

```
int sum = 0;
for (int x : numbers) {
  sum += x;
}
```

Summing elements

We are reducing a list of numbers to a single number (here: the sum)

- works iteratively
- two parameters: initial value for `sum` (here `0`) and the operator(here `+`)

Summing elements using `reduce`

René Witte

Java 8 Style

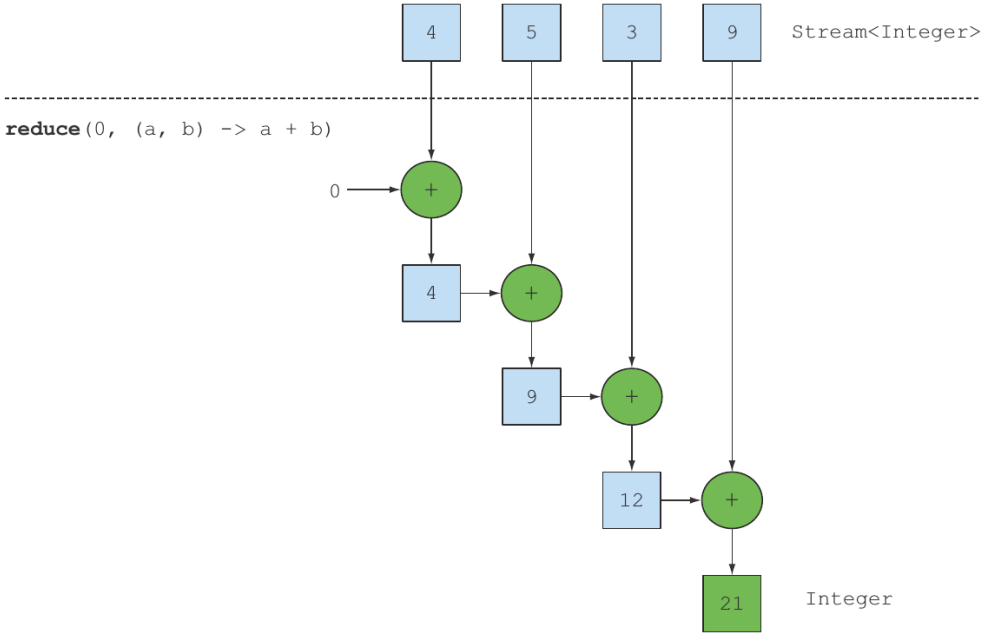
```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

`reduce`

two arguments:

- an initial value (here `0`)
- a `BinaryOperator<T>` to combine two elements and produce a new value here the lambda `(a, b) -> a + b`

Numbers stream



Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

More `reduce` Examples

René Witte
Concordia University

Multiply instead of add

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

No initial value (overloaded `reduce`)

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

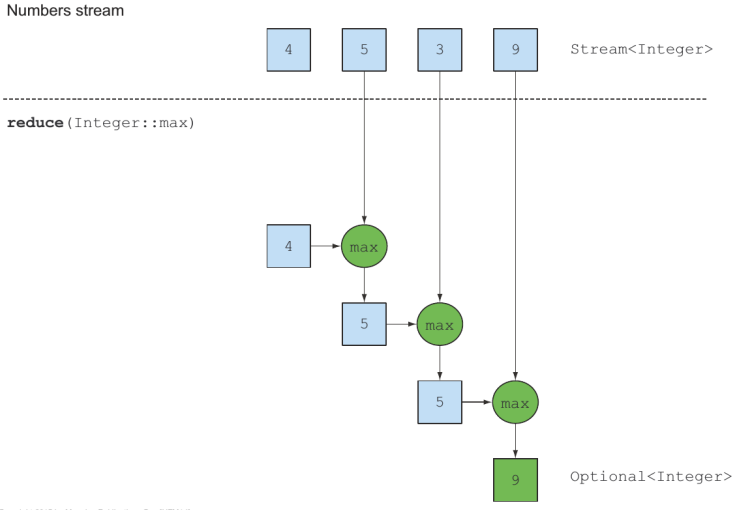
Compute sum in parallel

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

Maximum and Minimum

Compute minimum and maximum in a stream using `reduce`

```
Optional<Integer> min = numbers.stream().reduce(Integer::min);
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```



Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

Stream Operations: Stateful vs. stateless

René Witte
Concordia University

Stateless Operations

- Take each input element and return zero or one result
- E.g., `map` or `filter`
- Do not need internal state (unless added in lambda)

Stateful Operations

- Operations need to keep internal state to accumulate result
- E.g., `reduce`, `sum`, or `max`
- Internal state is of bounded size (independent of stream size)

Unbounded Stateful Operations

- Operations requiring all elements in the stream to be buffered
- E.g., `sorted` or `distinct`
- Internal state is of unbounded size
- Problematic for large (or infinite) input streams

Stream Operations Summary (I)

Operation	Type	Return type	Type/functional interface used	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`distinct`	Intermediate (stateful-unbounded)	`Stream<T>`		
`skip`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`limit`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`flatMap`	Intermediate	`Stream<R>`	`Function<T, Stream<R>>`	`T -> Stream<R>`
`sorted`	Intermediate (stateful-unbounded)	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`

Copyright 2015 by Manning Publications Co., [UFM14]

René Witte

Stream Operations Summary (II)

Operation	Type	Return type	Type/functional interface used	Function descriptor
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

Copyright 2015 by Manning Publications Co., [UFM14]

René Witte
Concordia University

Example: Traders executing transactions

René Witte
Concordia University

Your job is to write programs that can answer:

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions' values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

`Trader` class

```
public class Trader{
  private final String name;
  private final String city;

  public Trader(String n, String c){
    this.name = n;
    this.city = c;
  }

  public String getName(){
    return this.name;
  }

  public String getCity(){
    return this.city;
  }

  public String toString(){
    return "Trader:"+this.name + "␣in␣" + this.city;
  }
}
```

René Witte

`Transaction` class

```
public class Transaction{
  private final Trader trader;
  private final int year;
  private final int value;

  public Transaction(Trader trader, int year, int value){
    this.trader = trader;
    this.year = year;
    this.value = value;
  }

  public Trader getTrader(){ return this.trader; }
  public int getYear(){ return this.year; }
  public int getValue(){ return this.value; }

  public String toString(){
    return "{" + this.trader + ", " +
      "year: "+this.year+", " +
      "value:" + this.value +"}";
  }
```

René Witte

Traders and Transactions Data

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");

List<Transaction> transactions = Arrays.asList(
  new Transaction(brian, 2011, 300),
  new Transaction(raoul, 2012, 1000),
  new Transaction(raoul, 2011, 400),
  new Transaction(mario, 2012, 710),
  new Transaction(mario, 2012, 700),
  new Transaction(alan, 2012, 950)
)
```

René Witte
Concordia University

Find all transactions in 2011 and sort by value (small to high)

Solution

```
List<Transaction> tr2011 =
    transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());
```

René Witte
Concordia University

What are all the unique cities where the traders work?

Solution

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .distinct()
                .collect(toList());
```

Alternative solution

```
Set<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .collect(toSet());
```

René Witte
Concordia University

Find all traders from Cambridge and sort them by name

Solution

```
List<Trader> traders =
    transactions.stream()
                .map(Transaction::getTrader)
                .filter(trader -> trader.getCity().equals("Cambridge"))
                .distinct()
                .sorted(comparing(Trader::getName))
                .collect(toList());
```

René Witte

Return a string of all traders' names sorted alphabetically

René Witte
Concordia University

Solution

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .reduce("", (n1, n2) -> n1 + n2);
```

Alternative solution

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

Are any traders based in Milan?

Solution

```
boolean milanBased =
    transactions.stream()
                .anyMatch(transaction -> transaction.getTrader()
                                                    .getCity()
                                                    .equals("Milan"));
```

René Witte
Concordia University

Print all transactions' values from the traders living in Cambridge

Solution

```
transactions.stream()
            .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
            .map(Transaction::getValue)
            .forEach(System.out::println);
```

René Witte

What's the highest value of all the transactions?

Solution

```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)
                .reduce(Integer::max);
```

René Witte
Concordia University

Find the transaction with the smallest value

Solution

```
Optional<Transaction> smallestTransaction =
    transactions.stream()
                .reduce((t1, t2)
                        -> t1.getValue() < t2.getValue() ? t1 : t2);
```

Alternative solution

```
Optional<Transaction> smallestTransaction =
    transactions.stream()
                .min(comparing(Transaction::getValue));
```

René Witte
Concordia University

Outline

René Witte

Concordia University

Numeric Streams

René Witte
Concordia University

Compute total number of calories in a menu

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

Issues

- Each `Integer` needs to be unboxed (costly!)
- Cannot call `sum` directly, i.e.,

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .sum();
```

  does not work.

Solution: primitive stream specializations.

Primitive stream specializations

René Witte
Concordia University

Mapping to a numeric stream

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
```

mapToInt

- returns an `IntStream` (rather than `Stream<Integer>`)
- can call `sum` on `IntStream`
- other operations: `max`, `min`, `average`
- empty stream: result = 0

Converting back to a stream of objects

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);

Stream<Integer> stream = intStream.boxed();
```

René Witte

Default Values: `OptionalInt`

Find maximal element in an `IntStream`

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

Process the `OptionalInt`

```
int max = maxCalories.orElse(1);
```

René Witte
Concordia University

Numeric ranges

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                 .filter(n -> n % 2 == 0);

System.out.println(evenNumbers.count());
```

René Witte
Concordia University

Outline

René Witte

Concordia University

Streams from Values

Example: Stream of strings

```
Stream<String> stream =
    Stream.of("Java␣8␣", "Lambdas␣", "In␣", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

Example: Empty stream

```
Stream<String> emptyStream = Stream.empty();
```

René Witte
Concordia University

Streams from arrays

Example: Stream of primitive `ints`

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

René Witte
Concordia University

Streams from files

René Witte

Example: Count lines in a file

```
long lineCount = Files.lines(Paths.get("data.text")).count();
```

Example: Count unique words in a file

```
long uniqueWords = 0;
try(Stream<String> lines =
      Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){
  uniqueWords = lines.flatMap(line -> Arrays.stream(line.split("␣")))
                     .distinct()
                     .count();
} catch(IOException e){
 // Deal with the exception
}
```

Streams from functions: creating infinite streams!

René Witte
Concordia University

Example

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

`iterate`

- takes initial value and a lambda (`UnaryOperator<T>`)
- infinite stream: creates values on demand (unbounded)
- usually should be bounded through `limit(n)`
- computed sequentially: new result depends on previous

Generate

René Witte
Concordia University

Generate five random numbers in $[0, 1)$

```
Stream.generate(Math::random)
      .limit(5)
      .forEach(System.out::println);
```

`generate`

- takes a lambda of type `Supplier<T>`
- like `iterate`, and infinite stream: creates values on demand
- Supplier lambda should not have internal state for parallelization

How *not* to do it: Generating the Fibonacci Sequence

Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

First two numbers are 0, 1; each subsequent number is the sum of the previous two.

Writing a function for `generate`

```
IntSupplier fib = new IntSupplier(){
  private int previous = 0;
  private int current = 1;
  public int getAsInt(){
    int oldPrevious = this.previous;
    int nextValue = this.previous + this.current;
    this.previous = this.current;
    this.current = nextValue;
    return oldPrevious;
  }
};

IntStream.generate(fib).limit(10).forEach(System.out::println);
```

Mutable State

- The `fib` object has mutable state (the code has side effects)
- Cannot be used in parallel execution
- In general, prefer immutable approach (side effect-free code)

René Witte
Concordia University

Summary

Streams

- The Streams API lets you express complex data processing queries.
- You can filter and slice a stream using `filter`, `distinct`, `skip`, and `limit`
- You can extract or transform elements of a stream using `map` and `flatMap`
- You can find elements in a stream using the `findFirst` and `findAny` methods
- You can match a given predicate in a stream using `allMatch`, `noneMatch`, and `anyMatch`
- These methods make use of short-circuiting
- You can combine all elements of a stream using `reduce`
- Some operations such as `filter` and `map` are stateless; some operations such as `reduce` store state to calculate a value and are called stateful
- There are three primitive specializations of streams: `IntStream`, `DoubleStream`, and `LongStream`
- Streams can be created from a collection or from values, arrays, files, and specific methods such as `iterate` and `generate`
- An infinite stream is a stream that has no fixed size.

René Witte
Concordia University

Outline

René Witte

Concordia University

Reading Material

Required

- [UFM14, Chapter 5] (Working with Streams)

Supplemental

- [War14, Chapter 3] (Streams)

René Witte
Concordia University

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.