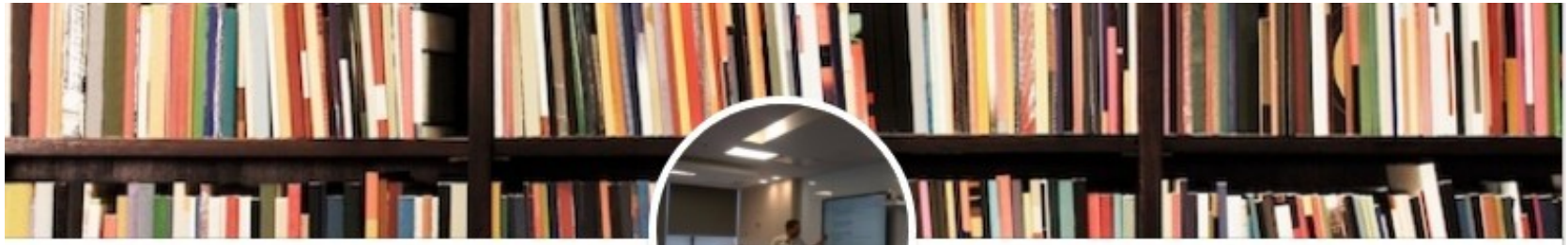
JAVA 8

Rajeev Gupta
Java Trainer & Consultant (MTech CS)
rgupta.mtech@gmail.com



Rajeev Gupta

FreeLancer Corporate Java JEE/ Spring Trainer

freelance • Institution of Electronics and Telecommunication Engineers IETE

New Delhi Area, India • 500+

---

1. Expert trainer for Java 8, GOF Design patterns, OOAD, JEE 7,Spring 5, Hibernate 5 ,Spring boot, microservice, netflix oss, Spring cloud, angularjs, Spring MVC, Spring Data, Spring Security, EJB 3, JPA 2, JMS 2, Struts 1/2, Web service

2. Helping technology organizations by training their fresh and senior engineers in key technologies and processes.

3. Taught graduate and post graduate academic courses to students of professional degrees.

I am open for advance java training /consultancy/ content development/ guest lectures/online training for corporate / institutions on freelance basis

Workshop Java 8

==========

=> Introduction to java 8, Why i should care for it?
=> functional interface
=> Lambda expressions
=> diff bw ann inner class vs lambda expression
=> Passing code with behavior parameterization
=> introduction to stream processing
=> functional interfaces defined in Java 8
=> Case study stream processing
=> Parallel data processing and performance

Why I should care about Java 8? What is offer?

Classical threads vs JUC

Java7 for and join

Java8 paralel processing declarative data processing

SQL vs Java Optimization should be Job of JVM or programmer?

Concurency vs paralel processing?

Java-style functional programming

JSR 335 Lambda Expression Closure
JEP 107 :Bulk data operation for collections forEach, filter, map,reduce

Java DNA

Structured ==> Reflective=> Object Oriented =>Functional => Imperative=>Concurrent => Generic

What Java is now?

".....Is a blend of imperative and object oriented programming enhanced with functional flavors"

Methods vs. Functions

	Method	Pure Function
Mutation	A method mutates , i.e., it modifies data and produces other changes and side effects.	A pure function does not mutat it just inspects data and produces results in form of new data.
How vs. What	Methods describe how things are done.	Pure functions describe what is done rather than how it is done
Invocation Order	Methods are used imperatively order of invocation matters.	Pure functions are used declaratively

Functional interface?
Why?

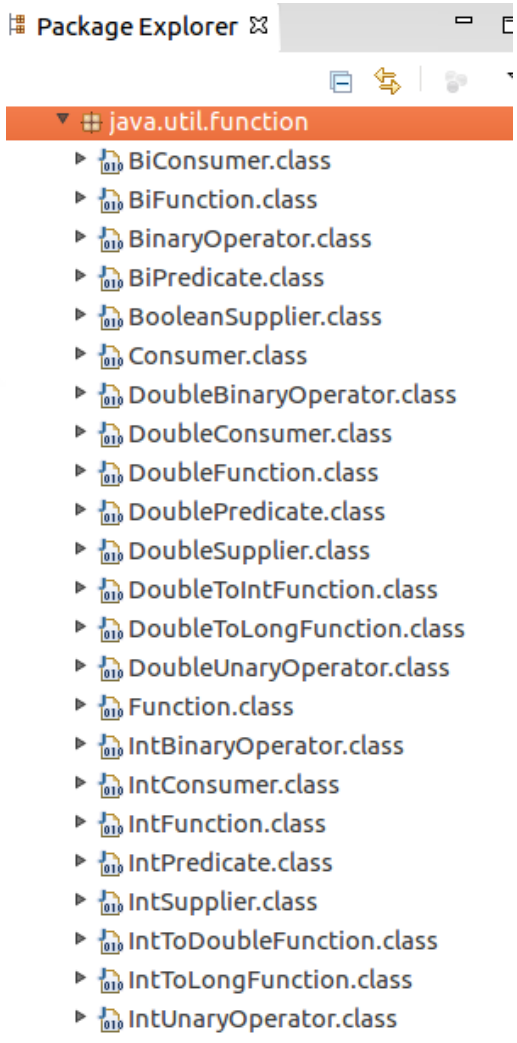
- New term of Java 8
- A functional interface is an interface with only one *abstract* method.

```
public interface Runnable {

   run();
};
```

```
public interface Comparator<T> {

   int compare(T t1, T t2);
};
```

Functional Interface

- Brand new java.util.function package.
- Rich set of function interfaces.
- 4 categories:
  - Supplier
  - Consumer
  - Predicate
  - Function



Interface Default and Static Methods

- Extends interface declarations with two new concepts:
  - Default methods
  - Static methods
- Advantages:

- No longer need to provide your own companion utility classes. Instead, you can place static methods in the appropriate interfaces

java.util.Collections

java.util.Collection

static <T> Collection<T>
synchronizedCollection(Collection<T

- Adding methods to an interface without breaking the existing implementation

Lambda Expression?

- **Old style**

```
Arrays.sort(testStrings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return(s1.length() - s2.length());
    }
});
```

- **New style**

```
Arrays.sort(testStrings,
        (s1, s2) -> { return(s1.length() – s2.length()); });
```

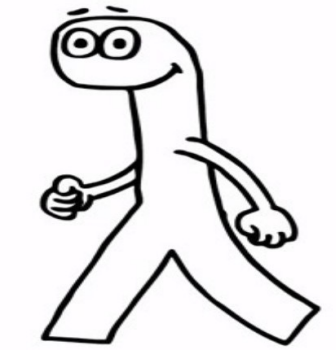
Lambda Expression vs inner classes?

Examples?

When to use lambda expression?

Performance difference?

First Class Functions

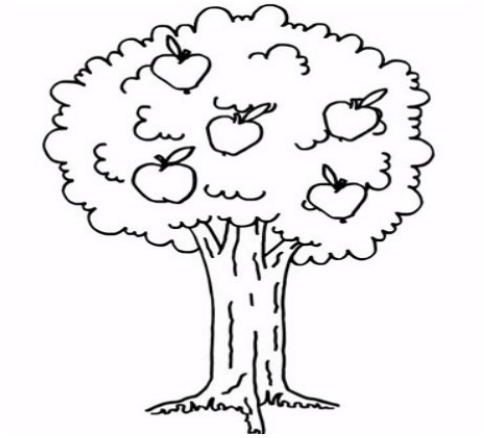


Lambda expression
==
anonymous method call Vs Inner class

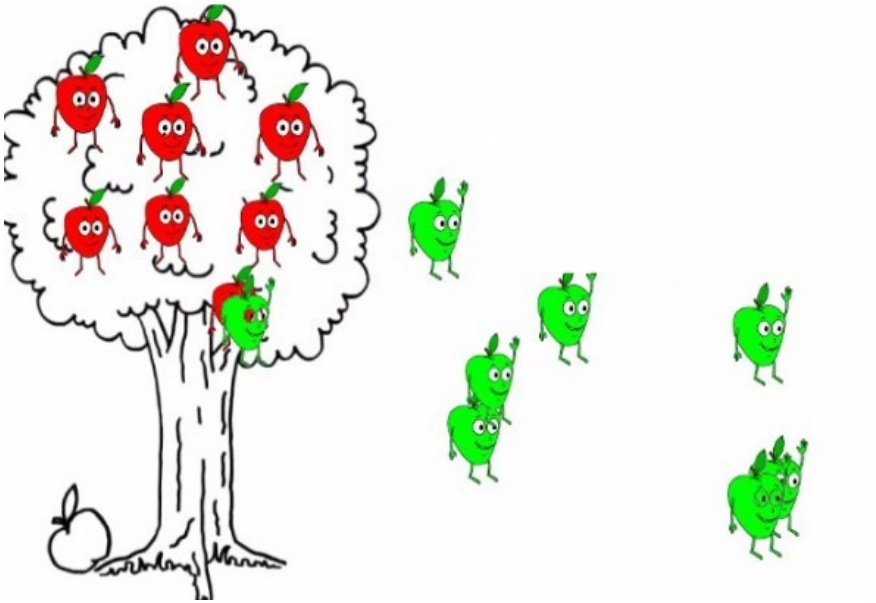
Behavior parameterization java 8

Light weight way to implement strategy design pattern In Java 8?

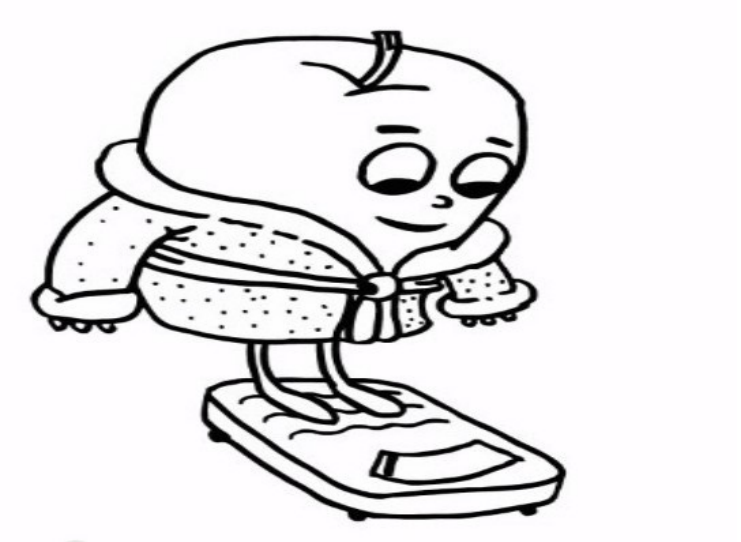
Our Inventory = List of Apples

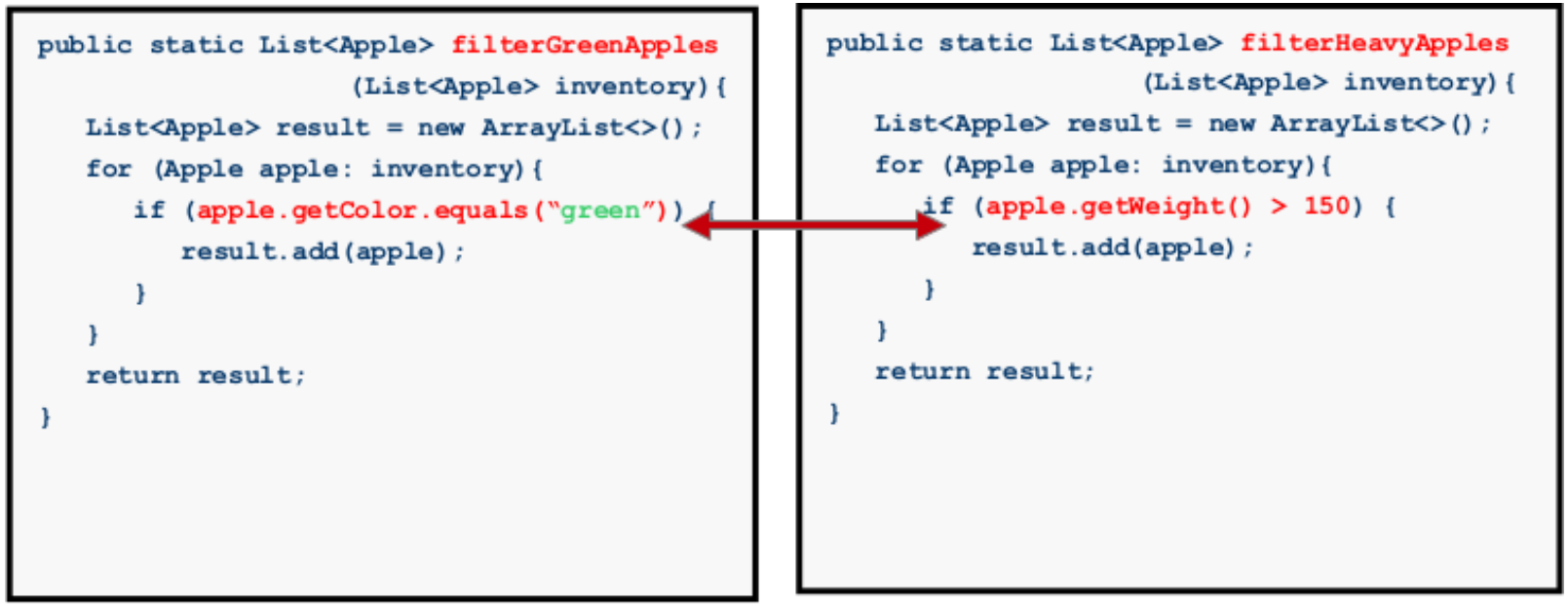


Green Apples, Please...



Now Heavy Ones , Please...




public static List<Apple> filterGreenApples
(List<Apple> inventory){
List<Apple> result = new ArrayList<>();
for (Apple apple: inventory){
if (apple.getColor.equals("green")) {
result.add(apple);
}
}
return result;
}
public static List<Apple> filterHeavyApples
(List<Apple> inventory){
List<Apple> result = new ArrayList<>();
for (Apple apple: inventory){
if (apple.getWeight() > 150) {
result.add(apple);
}
}
return result;
}

New Behavior

apple.getWeight > 150

apple.getColor.equals("green")

Behavior Parameterization

```
static List<Apple> filterApples(
          List<Apple> inventory, Predicate<Apple> p) {
   List<Apple> result = new ArrayList<>();
   for (Apple apple: inventory){
      if (p.test(apple)) {
         result.add(apple);
      }
   }
   return result;
}
```

Output

Heavy Apples

Green Apples

Stream processing An introduction

Data processing requirment?

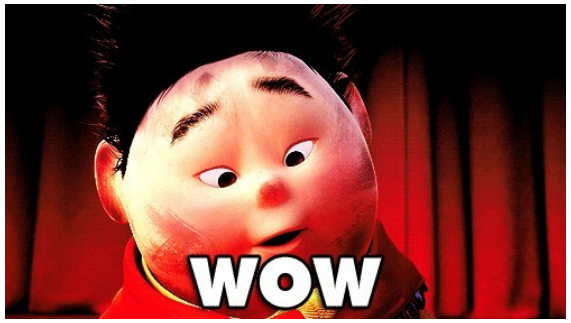
Return the names of dishes that are low in calories (<400), Sorted by number of calories

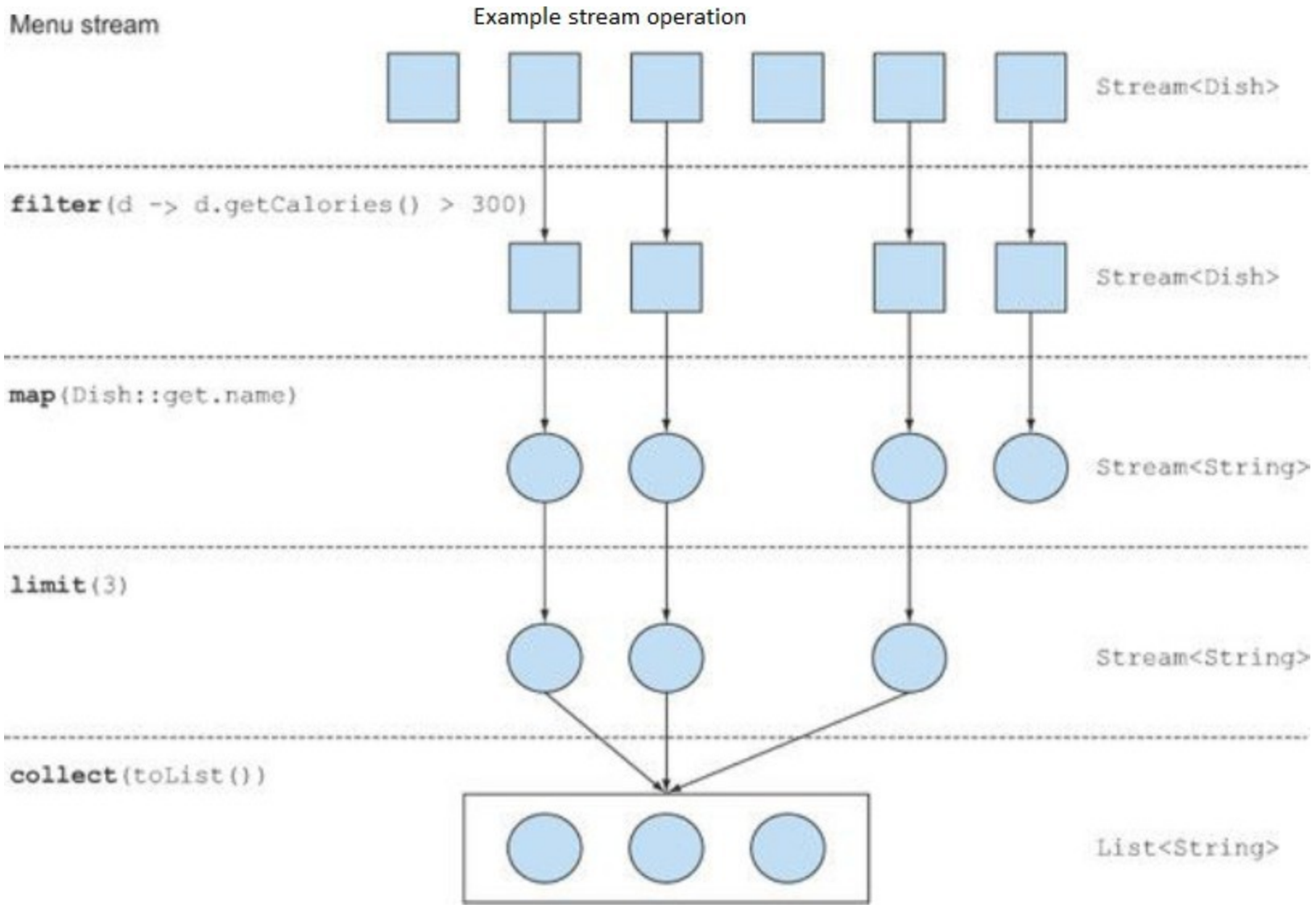
```
List<Dish> foods=new ArrayList<Dish>();
for(Dish d: foodsAll){
    if(d.getCalories()< 400)
        foods.add(d);
}
//now sorting as per calories
```

```
Collections.sort(foods, new Comparator<Dish>() {

    @Override
    public int compare(Dish o1, Dish o2) {
        return Integer.compare(o1.getCalories(), o2.getCalories());
    }
});
//now we want only names
```

```
List<String> names=new ArrayList<String>();
for(Dish d: foods){
    names.add(d.getName());
}
```

```
List<String> selectedFoodsNames=foodsAll
        .stream()
        .filter(f-> f.getCalories()< 400)
        .sorted(Comparator.comparing(Dish::getCalories))
        .map(f-> f.getName())
        .collect(Collectors.toList());
```



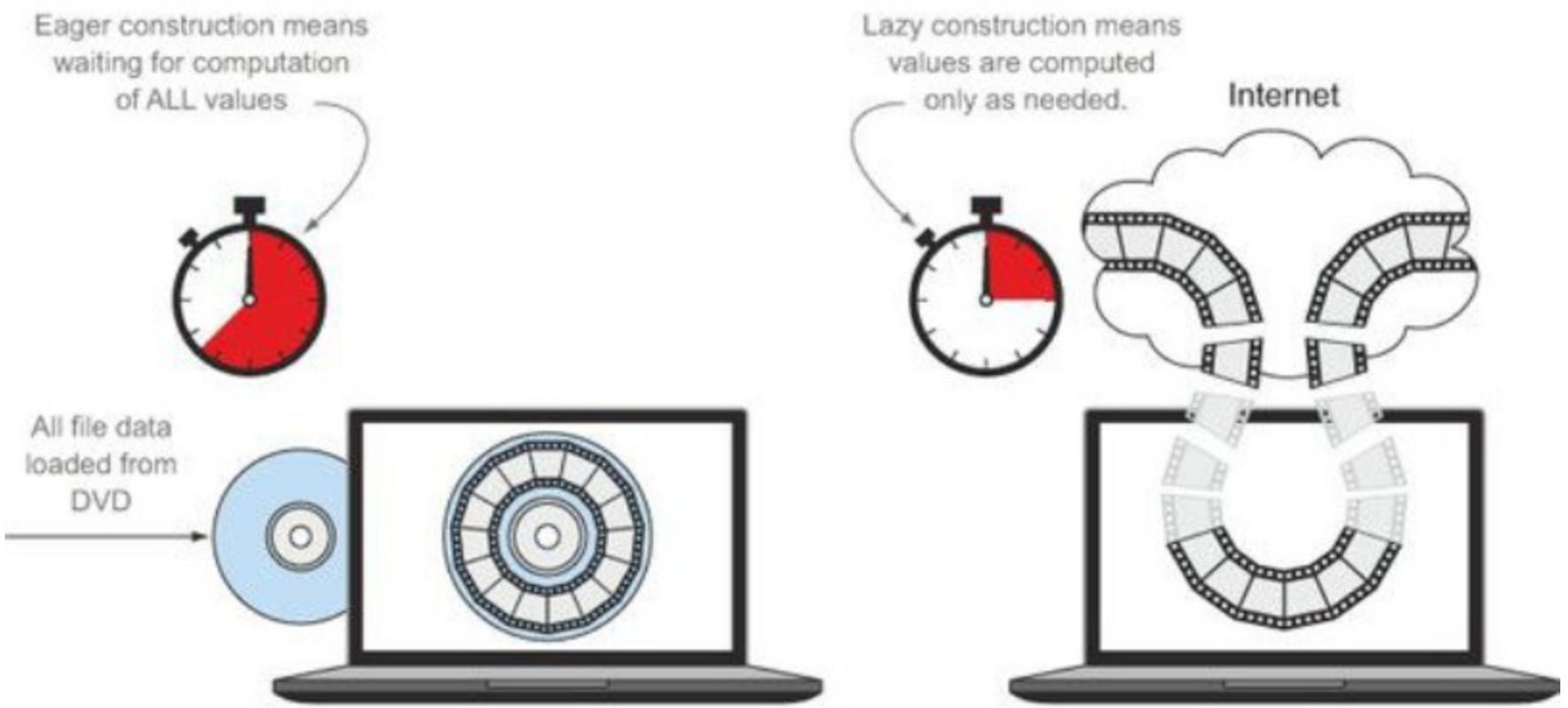

Menu stream
Example stream operation
Stream<Dish>
filter(d -> d.getCalories() > 300)
Stream<Dish>
map(Dish::get.name)
Stream<String>
limit(3)
Stream<String>
collect(toList())
List<String>

Stream vs collection?

Streams vs. collections

A collection in Java 8 is like a movie stored on DVD

A stream in Java 8 is like a movie streamed over the internet.

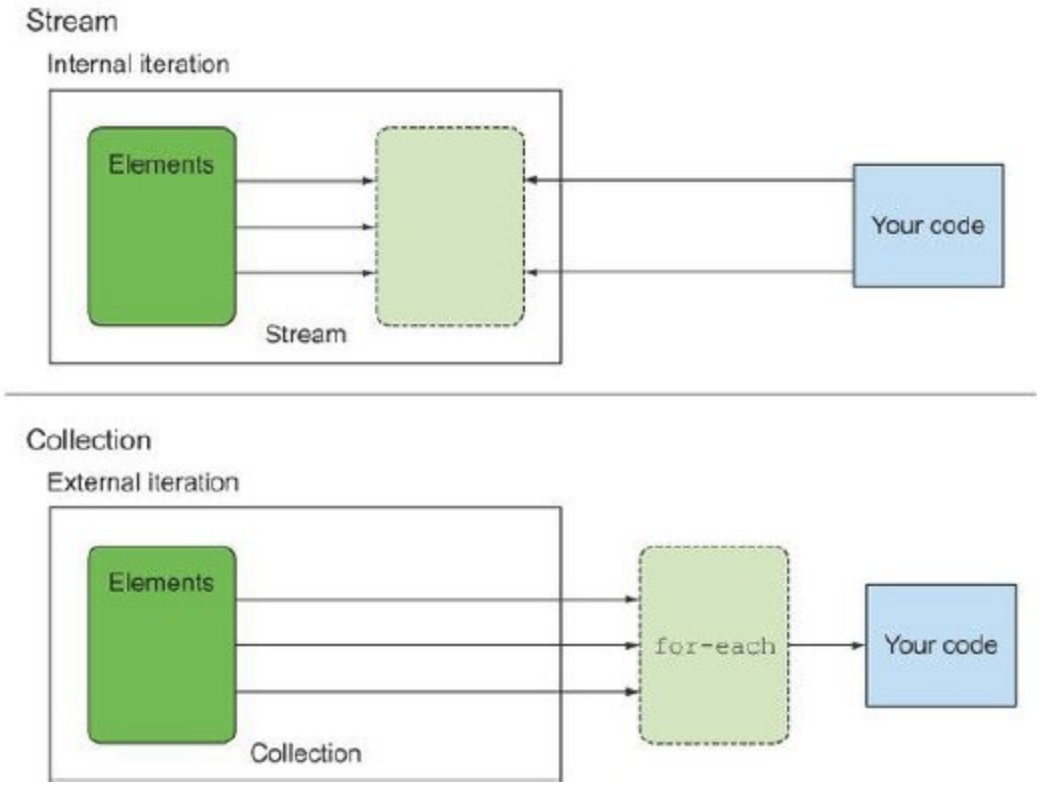


Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.

Like a streaming video, values are computed as they are needed.

Internal vs external iteration?

Internal vs. external iteration



Intermediate operations?

Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

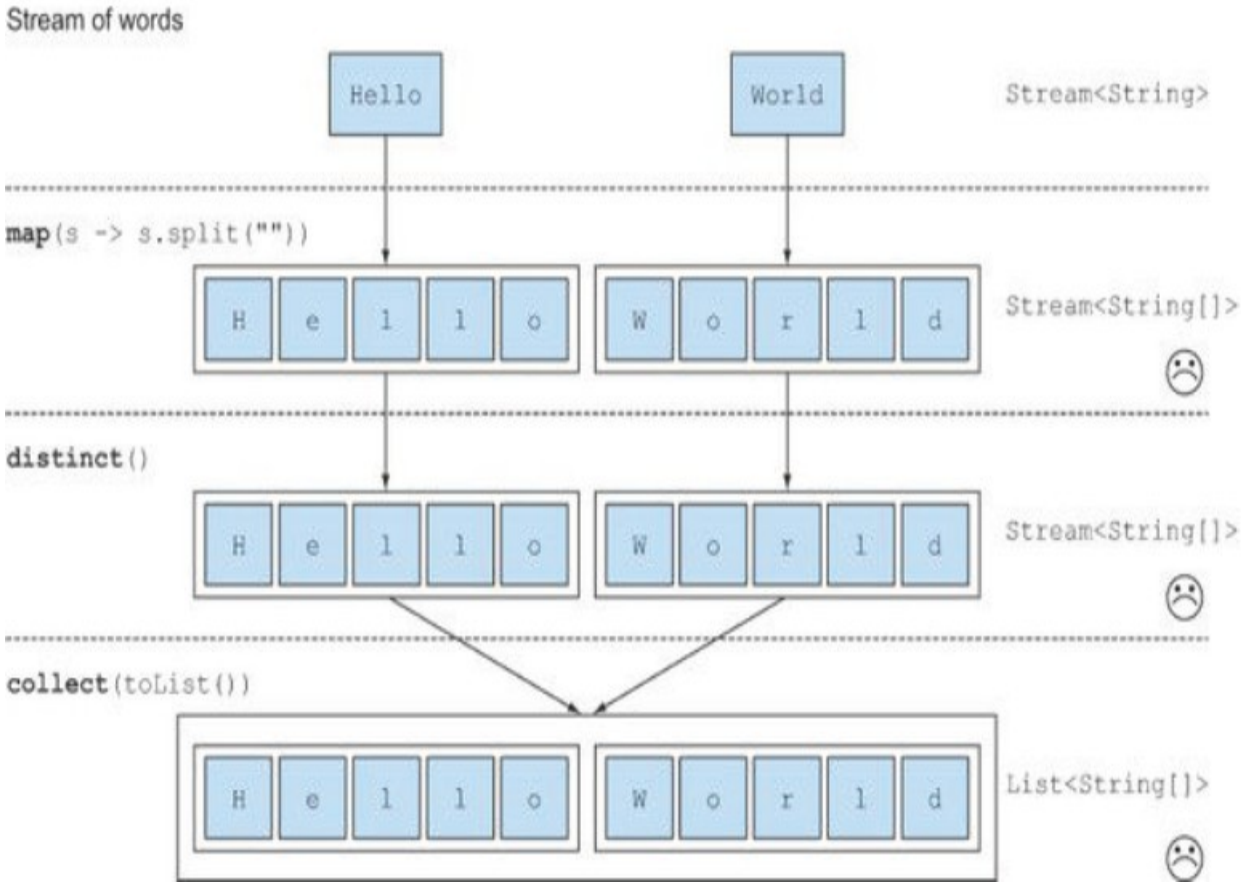
Terminal operation?

Terminal operations

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

Need of flatMap?

Incorrect implementation : need of flatMap

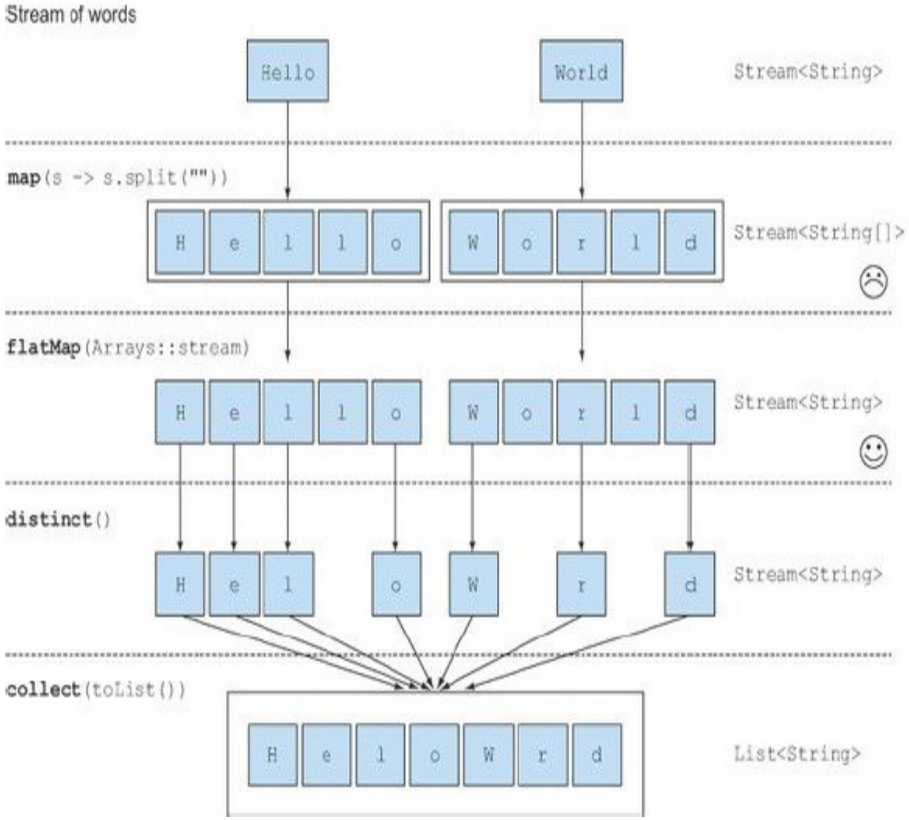



```
words.stream()
        .map(word -> word.split(""))
        .distinct()
        .collect(toList());
```

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

Converts each word into an array of its individual letters

Flattens each generated stream into a single stream



Correct implementation using flatMap:

Using the flatMap method has the effect of mapping each array not with a stream but *with the contents of that stream*. All the separate streams that were generated when using map(Arrays::stream) get amalgamated—flattened into a single stream

Intermediate and terminal operations List

Intermediate and terminal operations

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
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean

Operation	Type	Return type	Type/functional interface used	Function descriptor
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		
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		

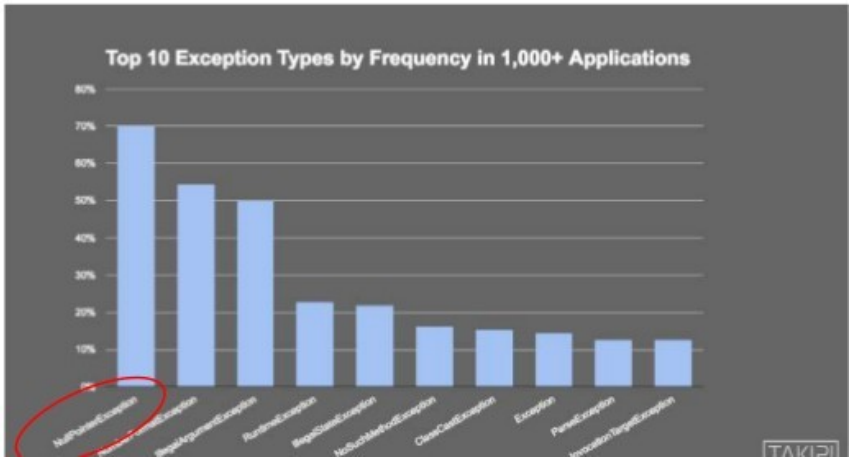
Optional java 8

The “Billion-Dollar mistake”

- Tony Hoare the creator of the null reference
- Created while designing ALGOL W
- Meant to model the absence of a value
- “… simply because it was so easy to implement”
- The cause of the dreaded Null Pointer Exceptions

NullPointerException

- Thrown when we attempt to use null value
- The most thrown exception



```
public class Person {
    private Car car;
    public Car getCar() { return car; }
}
public class Car {
    private Insurance insurance;
    public Insurance getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

```
public String getCarInsuranceName(Person person) {
        return person.getCar().getInsurance().getName();
}
```

Attempt 1: Deep Doubts

```
public String getCarInsuranceName(Person person) {
      if (person != null) {
            Car car = person.getCar();
            if (car != null) {
                  Insurance insurance = car.getInsurance();
                  if(insurance != null) {
                        return insurance.getName();
                  }
            }
      }
      return "Unknown";
}
```

Attempt 2: Multiple Exits

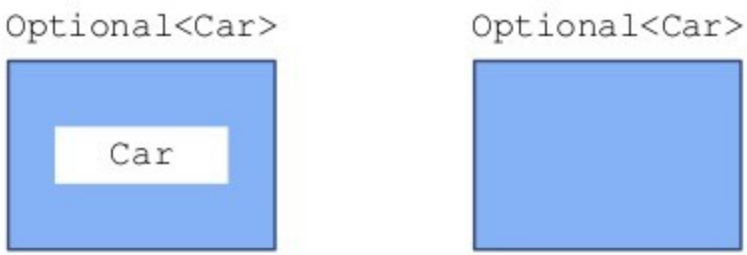
```
public String getCarInsuranceName(Person person) {
      if (person == null) {
            return "Unknown";
      }
      Car car = person.getCar();
      if (car == null) {
            return "Unknown";
      }
      Insurance insurance = car.getInsurance();
      if(insurance == null) {
            return "Unknown";
      }
      return insurance.getName();
}
```

So what's wrong with null?

- It's a source of error
  - NullPointerException
- It bloats code
  - Worsens readability
- It meaningless
  - Has no semantic meaning
- It breaks Java's philosophy
  - Java hides pointers from developers, except in one case

Optional

- New class located at `java.util.Optional<T>`
- Encapsulates optional values
- It wraps the value if it's there, and is empty if it isn't
  - `Optional.empty()`
- Signals that a missing value is acceptable
- We can't deference `null`, but we can dereference `Optional.empty()`



Progressive Refactored

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}
public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

How do we use Optional?

```
Optional<Car> optCar = Optional.ofNullable(car);
Car realCar = optCar.get(); DO NOT DO THIS! (ANTI-PATTERN)
```

- Throws a `NoSuchElementException` if empty
- Same conundrum as using null

Creating Optional objects

- Empty Optional
  - `Optional<T> opt = Optional.empty();`
  - `Optional<Car> optCar = Optional.empty();`
- Optional from a non-null value
  - `Optional<T> opt = Optional.of(T value);`
  - `Optional<Car> optCar = Optional.of(car);`
- Optional from a null value
  - `Optional<T> opt = Optional.ofNullable(T value);`
  - `Optional<Car> optCar = Optional.ofNullable(car);`

Chaining Optionals

Get the name of an insurance policy that a person may have.

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

```
Optional<Person> optPerson = Optional.ofNullable(person)
Optional<String> name = optPerson.map(Person::getCar)
                                 .map(Car::getInsurance)
                                 .map(Insurance::getName);
```

- Doesn't compile
- `getCar` returns `Optional<Car>`
- Results in `Optional<Optional<Car>>`

Get the name of an insurance policy that a person may have.

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

```
Public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 // default value if Optional is empty
                 .orElse("Unknown");
}
```

Sometimes you need to throw an exception if a value is absent.

```
Public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(age -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 // Throw an appropriate exception
                 .orElseThrow(IllegalStateException::new);
}
```

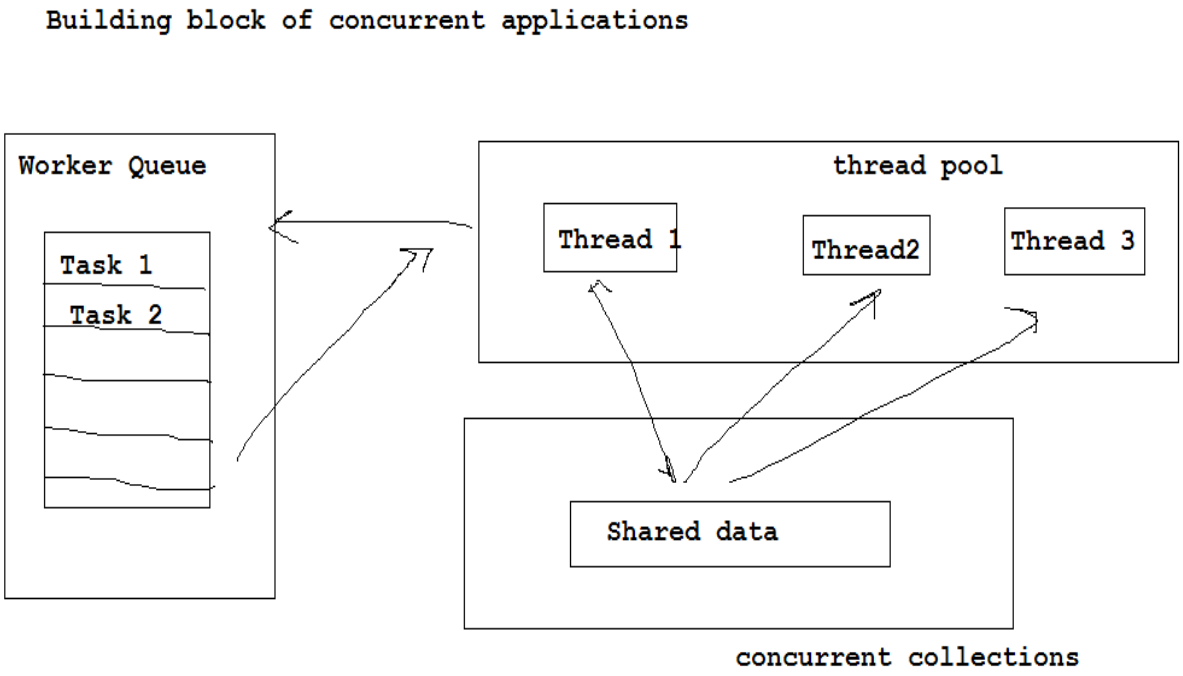
Method	Description
empty()	Returns an empty **Optional** instance.
equals(Object obj)	Indicates whether some other objects is “equal to” this Optional.
filter(Predicate<? super T>predicate)	If a value is present, and this value matches the given predicate, return an **Optional** describing the value, otherwise return an empty **Optional**.
flatMap(Function<? super T, Optional<U> mapper)	If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
get()	If a value is present in this Optional, returns the value, otherwise throws **NoSuchElementException**.
hashCode()	Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
ifPresent(Consumer<? Super T>consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
isPresent()	Returns true if there is a value present, otherwise false.

JUC, Multithreading enhancement Java 1.5

java.util.concurrency j.u.c

- Introduction
  - Concurrency : why very active now?
    Building block of concurrent application
    Java concurrency –over the year
  - Fork join -introduction Coding examples….

Building block of concurrent application



- Thre
- Shared data
- Worker queue

Java concurrency –Pre Java 5

- Task Executor
  - No such framework Runnable/Thread
  - No of task= no of threads
  - Use wait() and notify() for thread intercommunication
- Vector, Hash table
  - Collection
  - Worker queue
  - No framework for task dispatching Create an thread when a task to run
  - Too primitive low level
  - High cost of thread management Synchronization=poor performance <<Runnable>> can not return values

Java concurrency –Java 5

- Task Executor
  - Executor Service Framework
  - Support scheduling (Timed execution)
  - Thread pools (lifecycle management)
  - Specify order of execution
  - Task
    - Object of <<Callable>> or <<Runnable>>
    - <<Callable>> support for asynchronous communication call() and return values
    - Future: hold result of processing
- Collections
  - Concurrent HashMap, Copy On WriteList
- Worker queue
  - Blocking Queue, Deque…..
- Issues
  - No of task can't altered at run time
  - Very difficult and performance intensive when applied to recursive style of problems……

Java concurrency –Java 7

- Fork join framework
- For supporting a style of programming in which problem are solved recursively splitting them into subtask that are solved in parallel, waiting for them to complete and then composing results !
- JSR 166 Java 7
  - Extend Executor framework for recursive style of problems
- Implements Work stealing algorithm
  - Idle worker “steal” the woks from worker who are busy..

F&J framework Java 1.7
Parellelprocessing

Fork Join Framework

Parallel version of Divide and Conquer

1. Recursively break down the problem into sub problems
2. Solve the sub problems in parallel
3. Combine the solutions to sub-problems to arrive at final result

General Form

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

Java library for Fork–Join framework

Started with JSR 166 efforts

JDK 7 includes it in java.util.concurrent

Can be used with JDK 6.

- Download jsr166y package maintained by Doug Lea
- http://gee.cs.oswego.edu/dl/concurrency-interest/

Selecting a max number

```
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;

    public SelectMaxProblem(int[] numbers2, int i, int j) {
        this.numbers = numbers2;
        this.start = i;
        this.end = j;
        this.size = j - i;
        System.out.println("start:" + start + ",end:" + end + ",size:" + size);
    }

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        System.out.println("returning max:"  + max);
        return max;
    }

    public SelectMaxProblem subproblem(int subStart, int subEnd) {
        return new SelectMaxProblem(numbers, start + subStart,
                                    start + subEnd);
    }
}
```

Sequential algorithm

Selecting max with Fork-Join framework

```
13 public class MaxWithForkJoin extends RecursiveAction {
14     private final int threshold;
15     private final SelectMaxProblem problem;
16     public long result;
17
18     public MaxWithForkJoin(SelectMaxProblem problem, int threshold) {
19         this.problem = problem;
20         this.threshold = threshold;
21     }
22
23     @Override
24     protected void compute() {
25
26         if (problem.size < threshold) {
27             result = problem.solveSequentially();
28         }
29         else {
30             int midpoint = problem.size / 2;
31
32
33             MaxWithForkJoin left = new MaxWithForkJoin(problem.subproblem(0, midpoint), threshold);
34             MaxWithForkJoin right = new MaxWithForkJoin(problem.subproblem(midpoint + 1, problem.size), threshold);
35
36
37             invokeAll(left, right);
38
39
40
41             result = Math.max(left.result, right.result);
42         }
43     }
```

Solve sequentially if problem is sma
Otherwise Create sub tasks &

Fork

Join the results

Fork-Join framework implementation

Basic threads (Thread.start() , Thread.join())

- May require more threads than VM can support

Conventional thread pools

- Could run into thread starvation deadlock as fork join tasks spend much of the time waiting for other tasks

ForkJoinPool is an optimized thread pool executor (for fork-join tasks)

ForkJoin and Map Reduce

	Fork-Join	Map-Reduce
Processing on	Cores on a single compute node	Independent compute nodes
Division of tasks	Dynamic	Decided at start-up
Inter-task communication	Allowed	No
Task redistribution	Work-stealing	Speculative execution
When to use	Data size that can be handled in a node	Big Data
Speed-up	Good speed-up according to #cores, works with decent size data	Scales incredibly well for huge data sets

Source:
http://www.macs.hw.ac.uk/cs/techreps/docs/files/HW-MACS-TR-0096.pdf

Java 8 Parellel processing ‘declerative way’

Parallel Streams

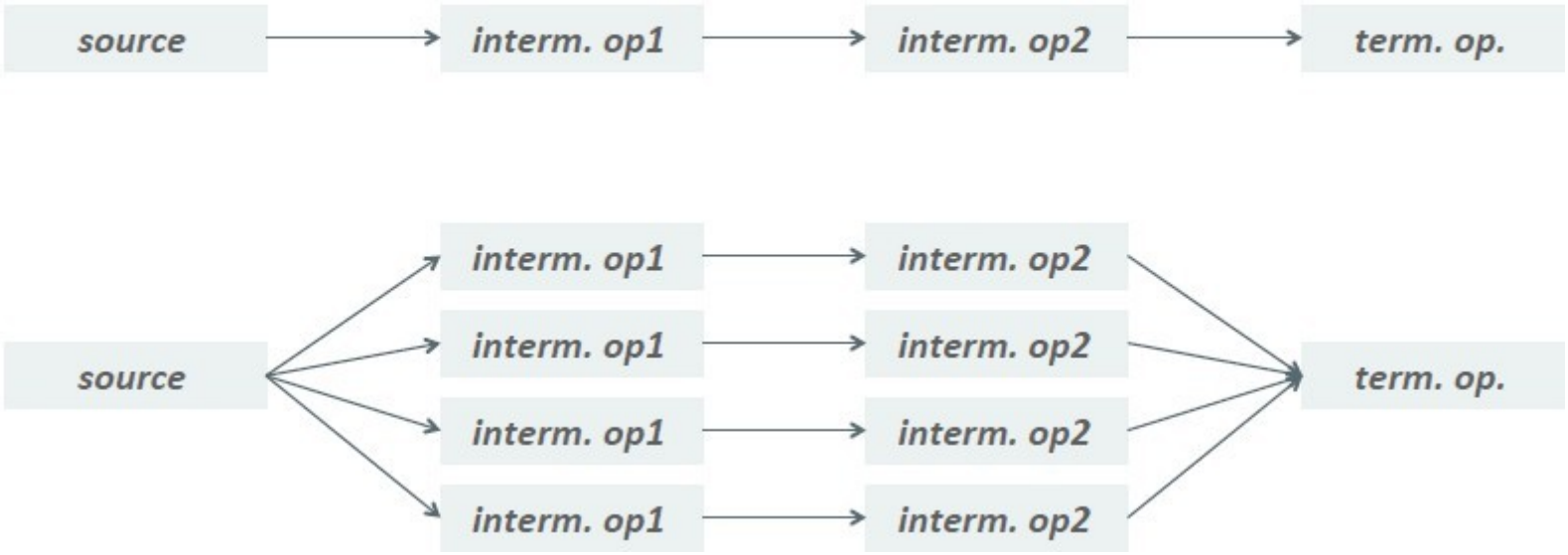
- Source starts with stream(), parallelStream(), or other stream factory
- Can be switched using parallel() or sequential() calls
- Parallel vs sequential is a property of the entire pipeline
  - can't switch between parallel and sequential in the middle
  - "last one wins"
- Parallel makes it auto-magically go faster, right?

```
collection.stream()
          .filter(...)
          .parallel()
          .map(...)
          .sequential()
          .collect(...);

// entire stream runs sequentially
```

Parallel Streams Considerations

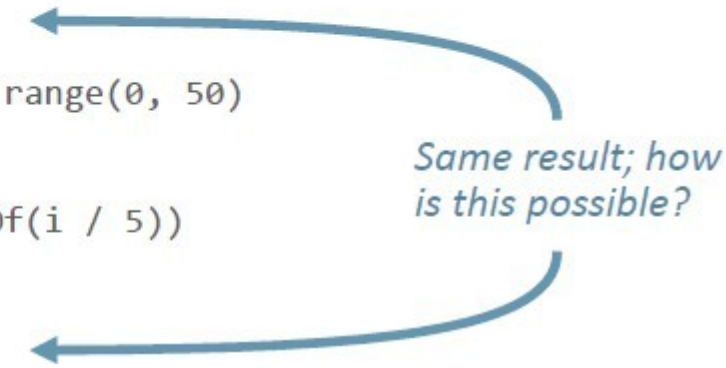
- Parallel and sequential streams should give the same result
  - parallelism leads to nondeterminism, usually bad! Need to control it.
- Encounter order vs. processing order
- Stateless vs. stateful: managing side effects
- Accumulation vs. Reduction
- Reduction: identity and associativity
- Explicit nondeterminism can speed things up
- Parallelism has overhead, might slow things down

Sequential vs. Parallel Streams



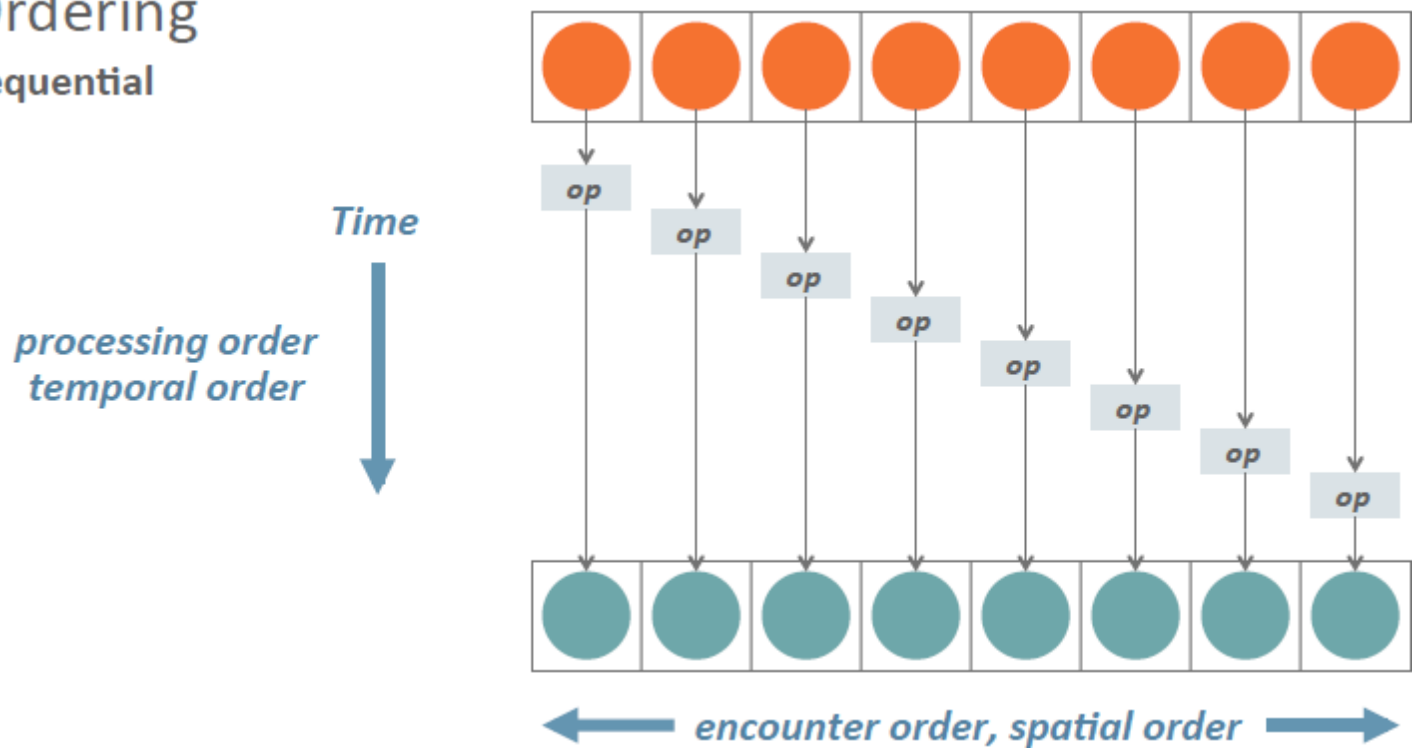
Sequential and Parallel Streams

```
List<String> output = IntStream.range(0, 50)
    .filter(i -> i % 5 == 0)
    .mapToObj(i -> String.valueOf(i / 5))
    .collect(toList());

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

List<String> output = IntStream.range(0, 50)
    .parallel()
    .filter(i -> i % 5 == 0)
    .mapToObj(i -> String.valueOf(i / 5))
    .collect(toList());

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



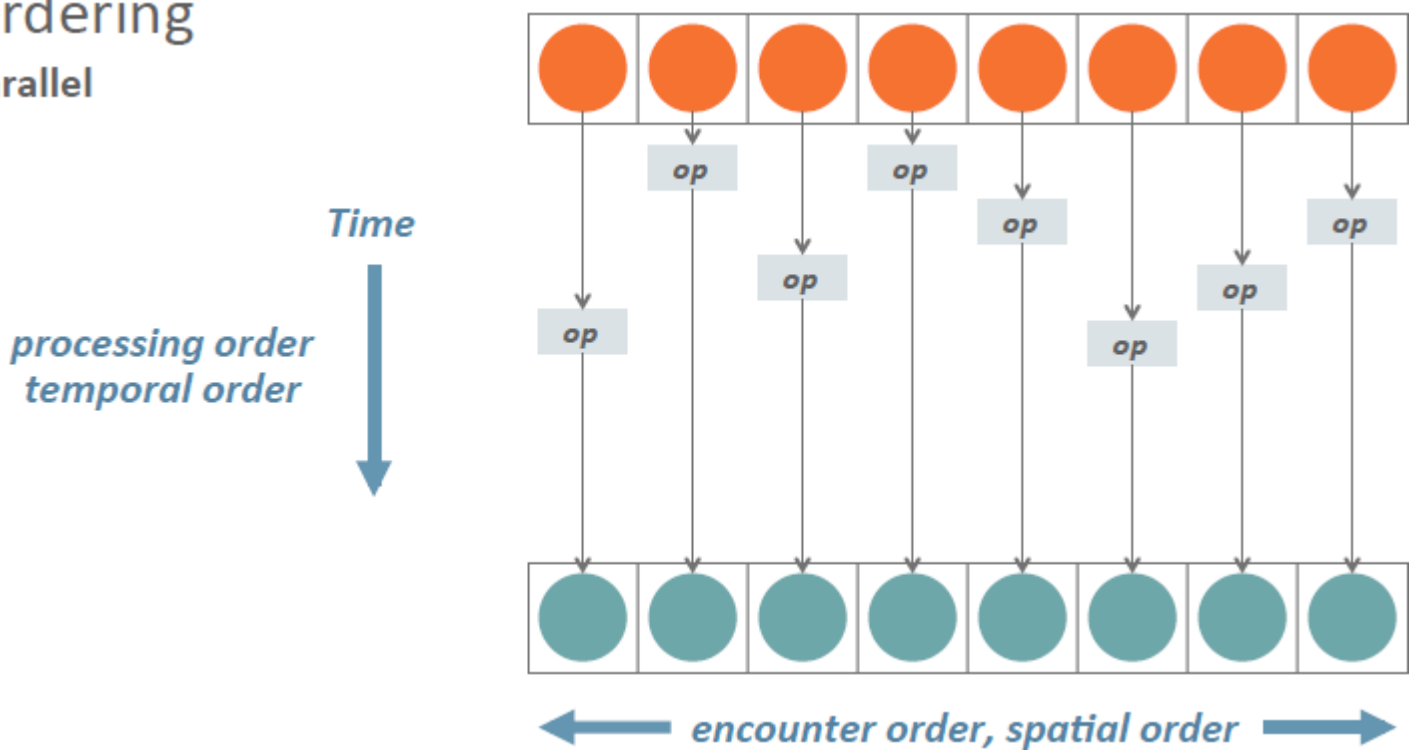
Ordering

Sequential



Ordering

Parallel

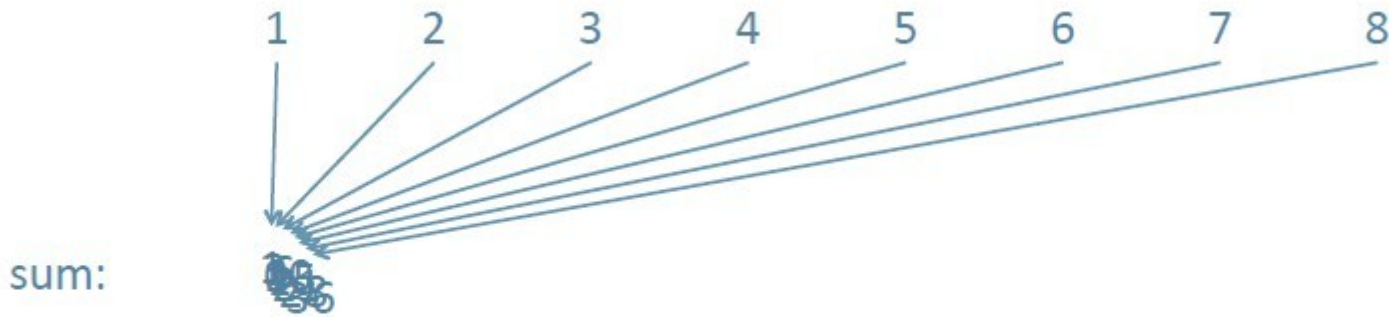


Accumulation vs. Reduction

```
long sum = 0L;

for (long i = 1L; i <= 1_000_000L; i++) {
    sum += i;
}

System.out.println(sum);
```

```
500000500000
```

Summation by Accumulation



Contention!

A Better Way: Reduction

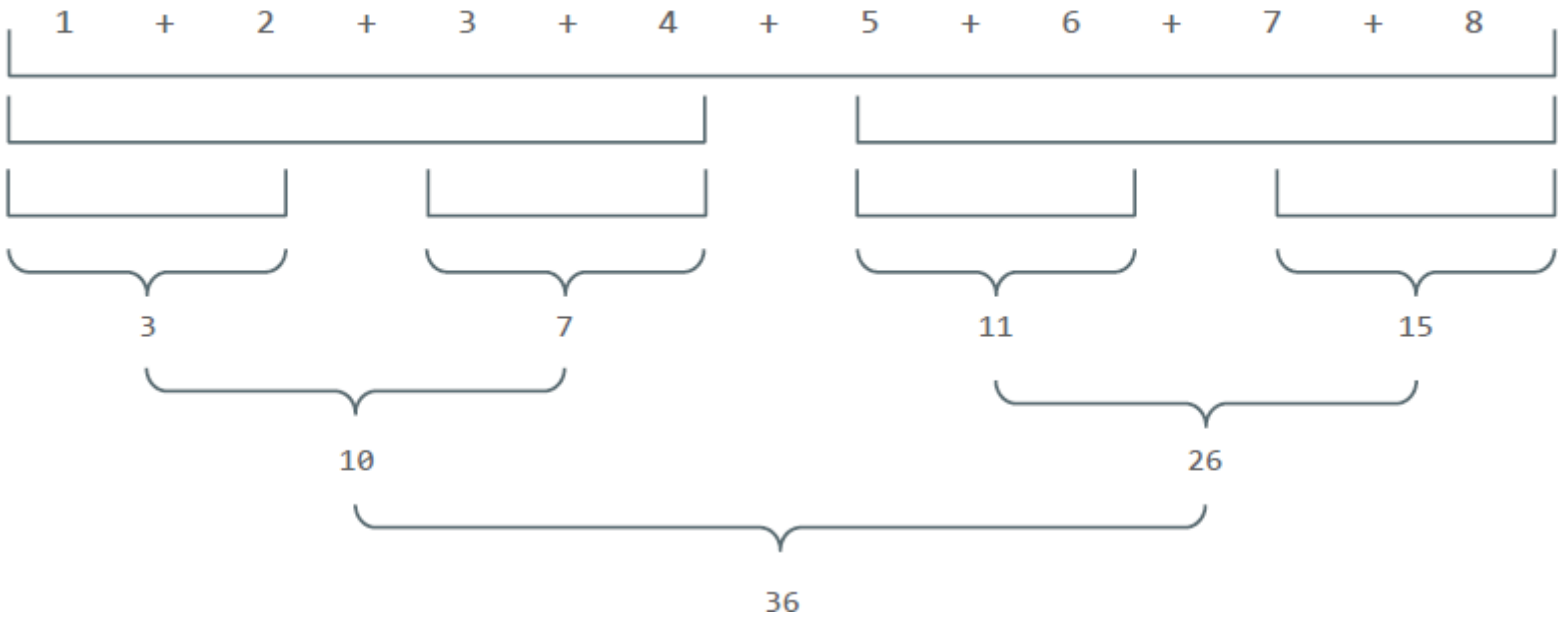
1 2 3 4 5 6 7 8

A Better Way: Reduction

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8

Reduction over addition:
Just put a plus between each value.

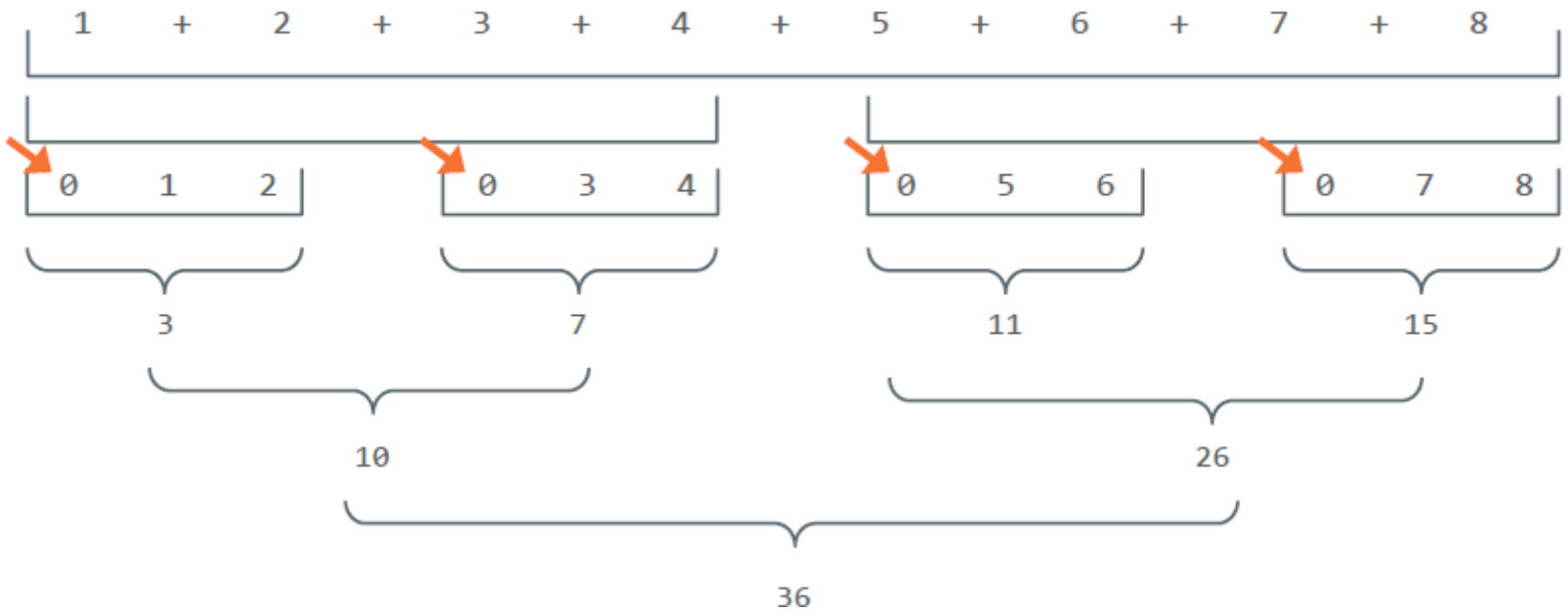
Reduction Implementation



Reduction – Identity

- Identity Value
  - the starting value of each partition of a parallel reduction
  - becomes the result if there are no values in the stream
  - it must be the ***right*** identity value
  - must really be an identity ***value*** (immutable, not mutable)

Reduction Implementation

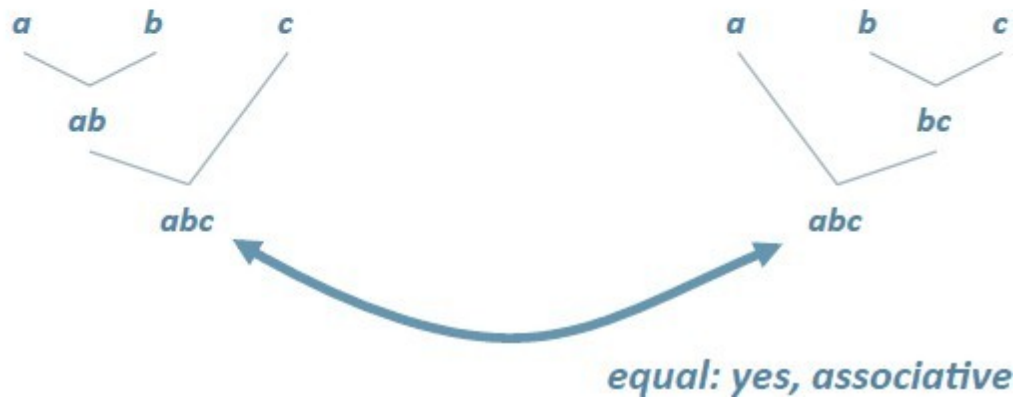


Associativity

- Remember elementary arithmetic?
  - $(a + b) + c = a + (b + c)$?
- Turns out it's quite important
- A function is ***associative*** if different groupings of operands don't affect the result
- Reduction functions must be associative

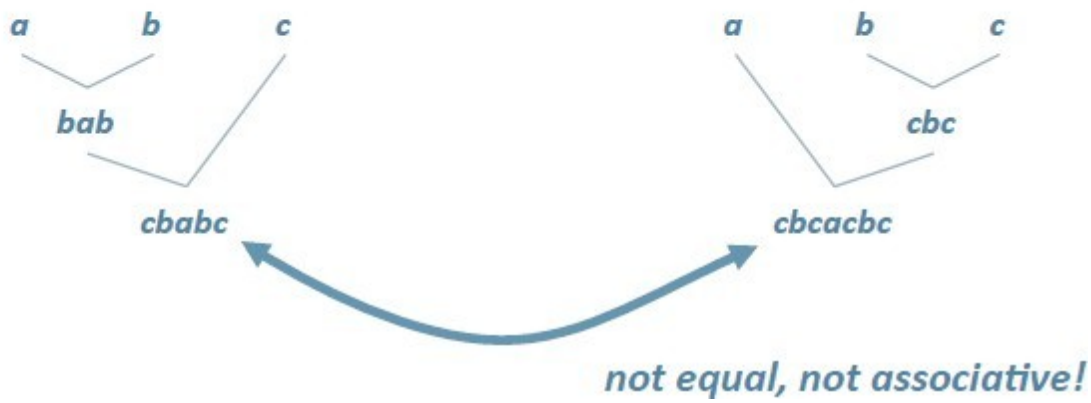
Associativity

```
(String x, String y) -> x + y
```



Associativity

```
(String x, String y) -> y + x + y
```



Reduction: Summary

- Consider reduction in preference to accumulation
- Reduction can be subtle
  - rewriting an accumulation into a reduction can be non-obvious
- Identity must be an immutable value
- Reduction function must be associative

Non-determinism

- Usually we want to get the same result for parallel as sequential
- Consider findFirst()
  - "first" means first in encounter (spatial) order
  - parallel can find *a* matching element quickly
  - but still has to search space to the left to ensure it's first
- Consider findAny()
  - parallel can find a matching element quickly
  - and it's done!

Where are the Threads?

- Parallel stream initiated by parallelStream() or parallel() call
  - who starts the threads? where do they come from?
- Stream workload split and dispatched to the ***common fork-join pool***
- Control over concurrency explicitly opaque in the API
  - allocation of resources should be by administrator/deployer, not programmer
  - common FP pool controlled by system properties; needs to be enhanced
- Policy APIs need development
  - split policy, degree of parallelism, handling blocking tasks

Case study Stream processing

Data setup

```
public class Book {
    private List<Author> authors;
    private String title;
    private int pages;
    private Genre genre;
    private int year;
    private String Isbn;
}
```

```
public class Author {
    private String name;
    private String lastName;
    private String country;
}

public enum Genre {
    NOVEL, SHORT_NOVEL, NON_FICTION;
}
```

rgupta.mtech@gmail.com

Lambda sample

- We need to find the books with more than 400 pages.

```
public List getLongBooks(List books) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (book.getPages() > 400) {
            accumulator.add(book);
        }
    }
    return accumulator;
}
```

- Now the requirements has changed and we also need to filter for genre of the book

```
public List getLongNonFictionBooks(List books) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (book.getPages() > 400 && Genre.NON_FICTION.equals(book.getGenre())) {
            accumulator.add(book);
        }
    }
    return accumulator;
}
```

- We need a different method for every filter, while the only change is the if condition

Lambda sample

- We can use a lambda. First we define a functional interface, which is an interface with only one abstract method

```
@FunctionalInterface
public interface BookFilter {

    public boolean test(Book book);
}
```

- Then we can define a generic filter and write as many implementation we want in just one line

```
public static List lambdaFilter(List books, BookFilter bookFilter) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (bookFilter.test(book)) {
            accumulator.add(book);
        }
    }
    return accumulator;
}

// one line filters
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);

BookFilter nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

Functional interfaces

- We don't need to write all the functional interfaces because Java 8 API defines the basic ones in *java.util.function package*

Functional interface	Descriptor	Method name
Predicate<T>	T → boolean	test()
BiPredicate<T, U>	(T, U) → boolean	test()
Consumer<T>	T → void	accept()
BiConsumer<T, U>	(T, U) → void	accept()
Supplier<T>	() → T	get()
Function<T, R>	T → R	apply()
BiFunction<T, U, R>	(T, U) → R	apply()
UnaryOperator<T>	T → T	identity()
BinaryOperator<T>	(T, T) → T	apply()

- So we did not need to write the BookFilter interface, because the Predicate interface has exactly the same descriptor

Lambda sample

- So we can rewrite our code as

```
public static List lambdaFilter(List books, Predicate bookFilter) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (bookFilter.test(book)) {
            accumulator.add(book);
        }
    }
    return accumulator;
}

// one line filters
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);

Predicate nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

Lambdas and existing interfaces

- Since in JDK there are a lot of interfaces with only one abstract method, we can use lambdas also for them:

```
// Runnable interface defines void run() method
Runnable r = () -> System.out.println("I'm running!");
r.run();

// Callable defines T call() method
Callable callable = () -> "This is a callable object";
String result = callable.call();

// Comparator defines the int compare(T t1, T t2) method
Comparator bookLengthComparator = (b1, b2) -> b1.getPages() - b2.getPages();
Comparator bookAgeComparator = (b1, b2) -> b1.getYear() - b2.getYear();
```

Method reference

- Sometimes code is more readable if we refer just to the method name instead of a lambda

Kind of method reference	Example
To a static method	Integer::parseInt
To an instance method of a class	Integer::intValue
To an instance method of an object	n::intValue
To a constructor	Integer::new

- So we can rewrite this lambda

```
Function<String, Integer> lengthCalculator = (String s) -> s.length();
```

- with a method reference

```
Function<String, Integer> lengthCalculator = String::length;
```

Comparators

- In former versions of Java, we had to write an anonymous inner class to speficy the behaviour of a Comparator

```
Collections.sort(users, new Comparator<Author>() {
    public int compare(Author a1, Author a2) {
        return a1.compareTo(a2.id);
    }
});
```

- We can use lambda for making code more readable:

```
// now sort is a oneliner!
Collections.sort(authors, (Author a1, Author a2) -> a1.compareTo(a2));
```

Streams

- The Java Collections framework relies on the concept of external iteration, as in the example below

```
for (Book book: books) {
    book.setYear = 1900;
}
```

- compared to internal iteration, like the example below

```
Books.forEach(b -> book.setYear(1900));
```

- The difference is not only in code readabilty and maintainability, is also related to performance: the runtime can optimize the internal iteration for parallelism, lazyness or reordering the data

Streams

- Let's see again the book filter we wrote with lan

```
public static List lambdaFilter(List books, Predicate bookFilter) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (bookFilter.test(book)) {
            accumulator.add(book);
        }
    }
    return accumulator;
}

// one line filters
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);

Predicate nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

We can rewrite it using streams

```
// stream based filters
List longBooks = books.stream().filter(b -> b.getPages() > 400).collect(toList());

List longNonFictionBooks =
    books.stream().filter(b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre())
    .collect(toList());
```

- The code is much cleaner now, because we don't need the lambdaFilter() method anymore. Let's see how it works

Streams

```
List longBooks = books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- What we've done is:
  - calling the stream() method on the collection, for trasforming it into a stream
  - calling the filter() method passing a Predicate, for filtering the elements of the stream dropping those not containing more than 400 pages
  - calling the collect() method with the static import of toList(), for collecting the filtered elements and put them into a List object

Stream operations

Operation	Operation type	Return type
filter(Predicate<T>)	intermediate	Stream<T>
map(Function <T, R>)	intermediate	Stream<R>
flatMap(Function <T, R>)	intermediate	Stream<R>
distinct()	intermediate	Stream<T>
sorted(Comparator<T>)	intermediate	Stream<T>
peek(Consumer<T>)	intermediate	Stream<T>
limit(int n)	intermediate	Stream<T>
skip(int n)	intermediate	Stream<T>
reduce(BinaryOperator<T>)	terminal	Optional<T>
collect(Collector<T, A, R>)	terminal	R
forEach(Consumer<T>)	terminal	void
min(Comparator<T>)	terminal	Optional<T>
max(Comparator<T>)	terminal	Optional<T>
count()	terminal	long
anyMatch(Predicate<T>)	terminal	boolean
allMatch(Predicate<T>)	terminal	boolean
noneMatch(Predicate<T>)	terminal	boolean
findFirst()	terminal	Optional<T>
findAny()	terminal	Optional<T>

Optionals

- Let's start with an example: ISBN in 2007 has changed from 10 to 13 characters. To check which version of ISBN a book has we have to write

```
boolean isPre2007 = book.getIsbn().length() > 10;
```

- What if a book was published before 1970, when ISBN did not exist and the property ISBN is null? Without a proper check, Null Pointer Exception will be thrown at run time! Java 8 has introduced the java.utill. Optional class. The code of our Book class can be now written as

```
public class Book {
    private List<Author> authors;
    private String title;
    private int pages;
    private Optional<String> Isbn;
    private Genre genre;
    private int year;
}
```

Optionals

- We can set the value with:

```
book.setIsbn(Optional.of("978000000"));
```

- Or, if the book was published before 1970:

```
book.setIsbn(Optional.empty());
```

- Or, if we don't know the value in advance:

```
book.setIsbn(Optional.ofNullable(value));
```

(in case value is null an empty Optional will be set)

- We can now get the value with

```
Optional<String> isbn = book.getIsbn();
System.out.println("Isbn: " + isbn.orElse("NOT PRESENT");
```

- If the Optional contains an ISBN it will be returned,otherwise the string "NOT PRESENT" will be returned

Case Study: Book Application Lets start…

rgupta.mtech@gmail.com

Data setup

```
public class Book {
    private List<Author> authors;
    private String title;
    private int pages;
    private Genre genre;
    private int year;
    private String Isbn;
}
```

```
public class Author {
    private String name;
    private String lastName;
    private String country;
}

public enum Genre {
    NOVEL, SHORT_NOVEL, NON_FICTION;
}
```

rgupta.mtech@gmail.com

Streams Problems

- We need all the books with more than 400 pages

Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

```
List<Book> top3LongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

```
List<Book> top3LongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

Here's how

Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

```
List<Book> top3LongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

- We need from the fourth to the last longest books. Here's how

```
List<Book> fromFourthLongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).skip(3).collect(toList());
```

Streams Problems

- We need to get all the publishing years

Streams Problems

- We need to get all the publishing years

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

Streams Problems

- We need to get all the publishing years

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors

Streams Problems

- We need to get all the publishing years
-

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

We need all the authors

```
Set<Author> authors =
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```

Streams Problems

- We need to get all the publishing years

```
Set<Author> authors =
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```

- We need all the authors

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the origin countries of the authors

Streams Problems

- We need to get all the publishing years

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors

```
Set<Author> authors =
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```

- We need all the origin countries of the authors

```
Set<String> countries =
    books.stream().flatMap(b -> b.getAuthors().stream())
    .map(author -> author.getCountry()).distinct().collect(toSet());
```

Streams Problems(Hint Optional)

- We want the most recent published book.

Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =
        books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =
        books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

- We want one of the books written by more than one author.

Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =
        books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

- We want one of the books written by more than one author.

```
Optional<Book> multiAuthorBook =
        books.stream().filter((b -> b.getAuthors().size() > 1)).findAny();
```

Streams Problems (Hint reduction)

- We want the total number of pages published.

Streams Problems

- We want the total number of pages published.

```
Integer totalPages =
        books.stream().map(Book::getPages).reduce(0, (b1, b2) -> b1 + b2);
```

- Or

```
Optional<Integer> totalPages =
        books.stream().map(Book::getPages).reduce(Integer::sum);
```

- We want to know how many pages the longest book has.

Streams Problems

- We want the total number of pages published.

```
Integer totalPages =
        books.stream().map(Book::getPages).reduce(0, (b1, b2) -> b1 + b2);
```

- Or

```
Optional<Integer> totalPages =
        books.stream().map(Book::getPages).reduce(Integer::sum);
```

- We want to know how many pages the longest book has.

```
Optional<Integer> longestBook =
     books.stream().map(Book::getPages).reduce(Integer::max);
```

The Collector interface

- The Collector interface was introduce to give developers a set of methods for reduction operations

Method	Return type
toList()	List<T>
toSet()	Set<t>
toCollection()	Collection<T>
counting()	Long
summingInt()	Long
averagingInt()	Double
joining()	String
maxBy()	Optional<T>
minBy()	Optional<T>
reducing()	...
groupingBy()	Map<K, List<T>>
partioningBy()	Map<Boolean, List<T>>

Collector Streams Problems

- We want the average number of pages of the books.

Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =
    books.stream().collect(averagingInt(Book::getPages));
```

Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books

Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books

```
String allTitles =
    books.stream().map(Book::getTitle).collect(joining(", "));
```

- We want the book with the higher number of authors?

Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books

```
String allTitles =
    books.stream().map(Book::getTitle).collect(joining(", "));
```

- We want the book with the higher number of authors?

```
Optional<Book> higherNumberOfAuthorsBook =
       books.stream().collect(maxBy(comparing(b -> b.getAuthors().size())));
```

Stream grouping Problems

- We want a Map of book per year.

Stream grouping Problems

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =
        Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a Map of how many books are published per year per genre.

Stream grouping Problems

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =
        Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a Map of how many books are published per year per genre.

```
Map<Integer, Map<Genre, List<Book>>> booksPerYearPerGenre =
    Setup.books.stream().collect(groupingBy(Book::getYear, groupingBy(Book::getGenre)));
```

- We want to count how many books are published per year.

Stream grouping Problems

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =
        Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a Map of how many books are published per year per genre.

```
Map<Integer, Map<Genre, List<Book>>> booksPerYearPerGenre =
    Setup.books.stream().collect(groupingBy(Book::getYear, groupingBy(Book::getGenre)));
```

- We want to count how many books are published per year.

```
Map<Integer, Long> bookCountPerYear =
    Setup.books.stream().collect(groupingBy(Book::getYear, counting()));
```

Stream partitioning Problems

- We want to classify book by hardcover.

Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =
        books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.

Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =
        books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.

```
Map<Boolean, Map<Genre, List<Book>>> hardCoverBooksByGenre =
    books.stream().collect(partitioningBy(Book::hasHardCover,groupingBy(Book::getGenre)));
```

- We want to count books with/without hardcover

Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =
        books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.

```
Map<Boolean, Map<Genre, List<Book>>> hardCoverBooksByGenre =
    books.stream().collect(partitioningBy(Book::hasHardCover,groupingBy(Book::getGenre)));
```

- We want to count books with/without hardcover

```
Map<Boolean, Long> count =
        books.stream().collect(partitioningBy(Book::hasHardCover, counting()));
```

Any questions?