FUNCTIONAL PROGRAMMING IN JAVA 8

TALHA OCAKÇI WWW.JAVATHLON.COM

GREATEST UPDATE TO JAVA

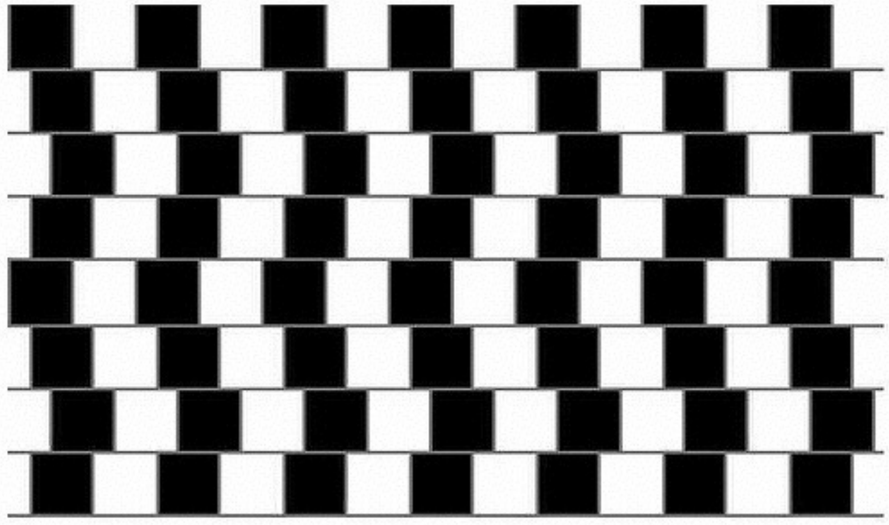
- Inspired by JVM languages such as Scala and Clojure and frameworks like Guava.
- Rise of functional programming

WHAT IS FUNCTIONAL PROGRAMMING?

- A programming style that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- f(x) = y g(f(x)) = z
- Based on lambda calculus. (Some inputs are transformed to some output without modifying the input)
- **No iteration, no for loop, no variable.**

NO ITERATION?

- What? Without a for loop, how I will process all elements of an ArrayList?
- Yes seems weird. You need to change your mindset. **Mind bending at first.**



FUNCTION… YOU MEAN METHOD???

- No, I don’t mean the methods inside a class.
- FUNCTION IS NOT METHOD!
- Methods are not independent program units and they are bound to another context, like an object instance or class.
- Methods may depend upon values other than its arguments
- Methods may change the values of its arguments or some other static values.
- Pure functions are opposite.

WHAT IS A FUNCTION?

- Lives independently
- Does not depend to any value other than its arguments
- Output of a function is a value or another function
- Does not change any argument's value or any other external value. (no side effects)
- Functions may be composed together regardless of the context

WHAT IS FUNCTIONAL PROGRAMMING

- Functions exist independently. Not bound to objects or classes
- Functions can not change the state of any object or value
- **All values are immutable. Hates mutability!**
- No iteration. Operations are done by recursion.

Why so afraid of mutability?

Did you ever wonder why most solutions to program glitches are fixed by rebooting your computer or restarting the offending application?

That's because of state. The program has corrupted its state.

Why so afraid of mutability?

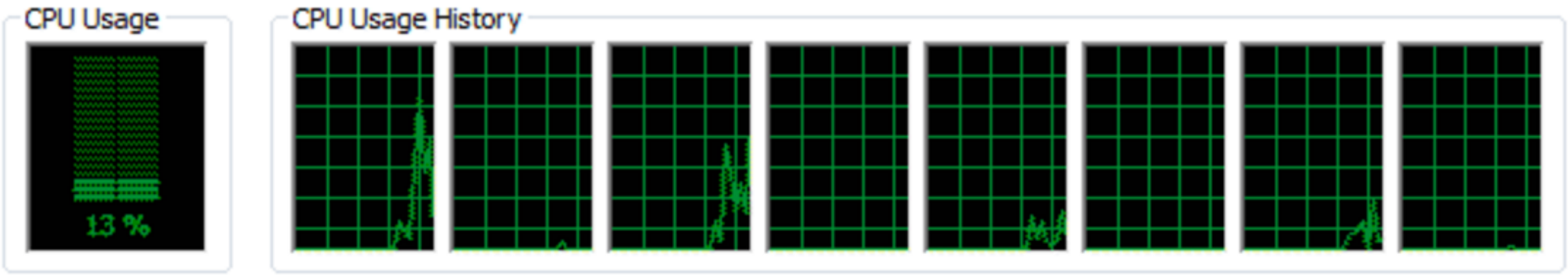
- Do you remember synchronised keyword, Barriers, Semaphores, Locks and their lack of sweetness?
- Have you ever scratched your hair because of deadlocks, thread starving, unexpected values of non-atomic object operations?
- We created the devil, intentionally, ourselves. We! Phew! Why?

How to prevent multithread problems?

- Prevent mutability
- If mutability is important, put the synchronisation block in the object.
  - Use AtomicInteger instead of Integer
  - Use ConcurrentHashMap instead of HashMap…

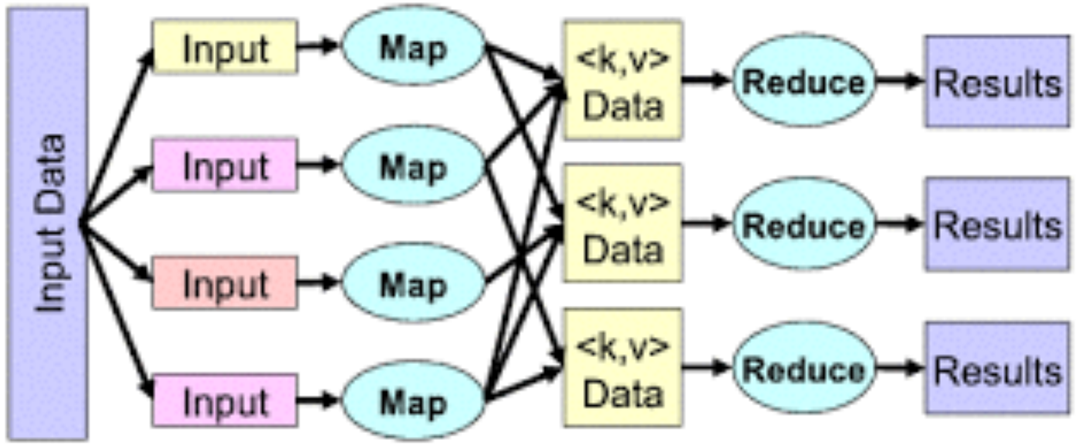
MULTICORE SUPPORT AND THREAD SYNCHRONIZATION IN JAVA

- **Fork/Join** for multicore in Java
- **Thread and ThreadPool** for multi-thread
- Leverage all cores of CPU. Distribute the tasks each core and then combine the results.

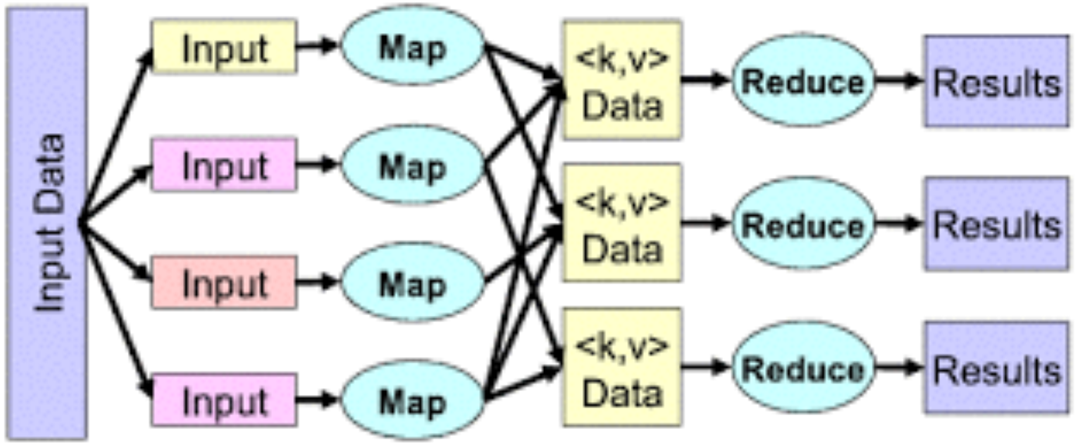


BIG-DATA AND MAP-REDUCE

- Huge data is processed in chunks, each chunk yields an intermediate form of data.
- Intermediate data is combined into single result.
- None of the functions may affect each other but simply yield a single result without a side effect.



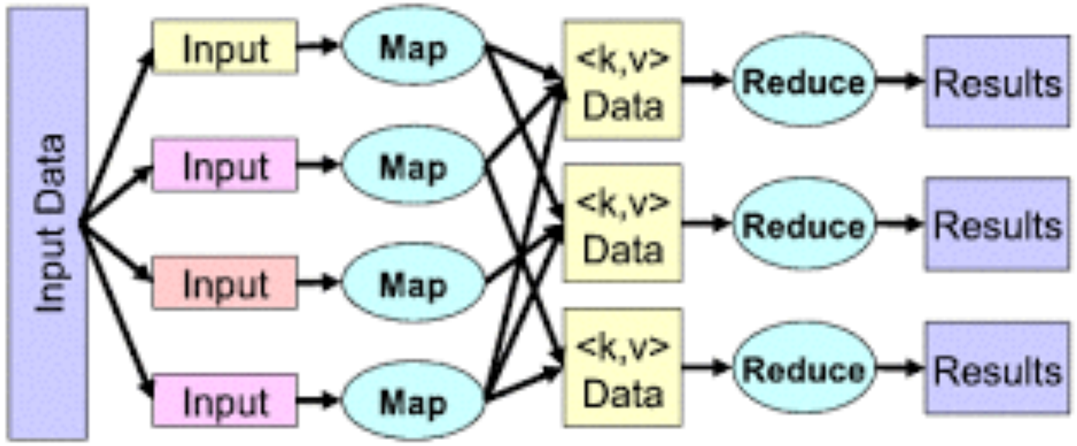
BIGDATA AND MAP-REDUCE



Mapper Combinator

BIGDATA AND MAP-REDUCE

- Each operation must be an independent **pure function**.
- So, functional programming is cut out for these type of problems.



	FUNCTIONAL	OBJECT ORIENTED	JAVA 8
Where is functions?	Independent	Bound to object or classes. Called as method.	Functional interface (Seems independent but bound to object or class again)
Value state	Immutable	Mutable	Immutable if you desire
Function chaining	Yes	Yes if in same instance	Lambda functions may chain
Synchronization and multicore support	Great!!!	Exist but really a cumbersome	So much easy than Java 7.

IMPERATIVE VS FUNCTIONAL

Imperative vs. Functional Separation of Concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l -> l.startsWith("ERROR")
        .limit(40)
        .collect(toList());
```

Green: Error handling

Blue: Stop criteria

Red: IO operations

Yellow: “Business logic”

ELEMENTS OF FUNCTIONAL PROGRAMMING

- Immutable state (yes, get rid of mutable state, the evil.)
- No side effects of function apply. (Input arguments are not modified)
- No iteration. Only recursion
- Function references and function compositions
- Higher order functions(functions may be passed as arguments to other functions) and currying (partial evaluation)
- Lazy evaluation of collections
- No null value
- Closure
- Monad

JAVA 8 PRETENDS TO BE A FUNCTIONAL PROGRAMMING LANGUAGE

Attribute	Java 8 Solution (or pretend to be a solution)
Immutable state	**final** keyword, Immutable collections (**Collections.unmodifiableX**). Both make the objects and the collection itself immutable.
Declarative. Not imperative	Use **Streams**
No iteration. Only recursion	No for loop on collections. Instead use **Streams** and its methods.
Function references and function composition	All **FunctionalInterfaces such as Function, Predicate, Consumer, Supplier** interfaces and **lambda expressions**.

JAVA 8 FUNCTIONAL PROGRAMMING ELEMENTS

Attribute	Java 8 Solution (or pretend to be a solution)
Higher order functions and currying	Manuel implementation, no real built-in solution
Lazy evaluation of collections	take() method of **Stream**
No null value	**Optional** class
Monad	**Stream** and **Optional** …
Function Chaining	Built-in with Function interface
Closure	Only with final variables

SUMMARY

- Java 8 brings functional programming elements into the language not in an academical and formal level but has a goodwill to have them
- Lambda expressions
- Stream
- @FunctionalInterface and some others…

FINAL - IMMUTABLE OBJECTS

- **final** keyword makes what final?
- final Date d = new Date();
- d = new Date() // legal?
- d.setTime(23234234); // legal?

FINAL IMMUTABLE OBJECTS

- **final** only protects the **address changes**. Can not protect state modifications.
- You must explicitly protect attributes with private access modifier and then prevent setter methods.
- **LocalDate** is announced as **immutable Date object in Java 8**.

FUNCTIONAL INTERFACE

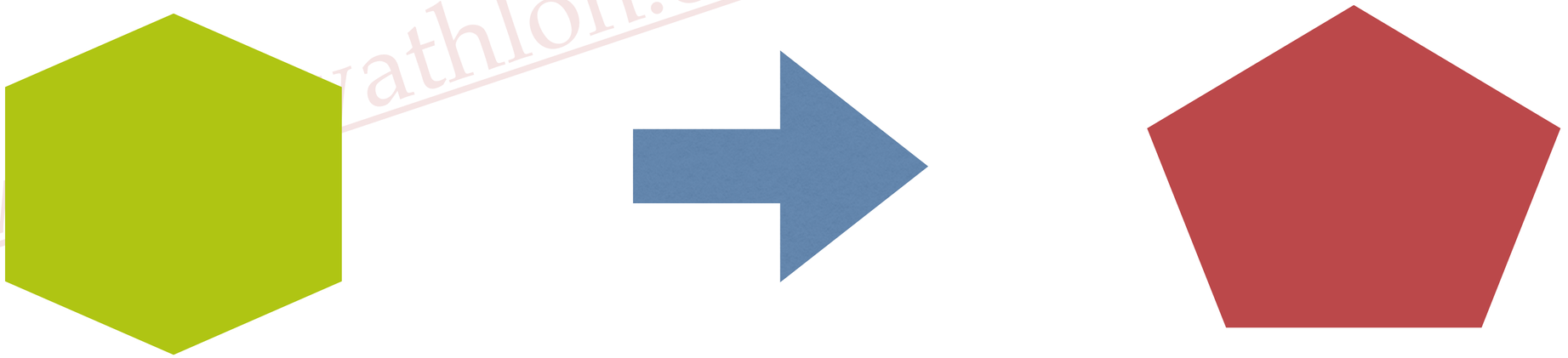
- Formal definition: **Single Abstract Method** interfaces
- An interface with only one abstract method.
- **Function**, **BiFunction**, **Predicate**, **Supplier** and **Consumer** interfaces are function interfaces that come or gain importance with Java 8.
- Known are **Runnable** interface with **run()** method; **Comparable** with **compare()** method.

FUNCTIONAL INTERFACES

Interface	Input	Output	Goal
Function	Single object of any type	Single object of any type	1. Applying a logic 2. Logic chaining
BiFunction	Two objects of any type	Single object of any type	1. Applying a logic 2. Logic chaining
Predicate	**Single object of any type**	boolean	Tests if a value conforms to a logic
Consumer	Single object of any type	None	Using a value and output some side effect
Supplier	None	Single object of any type	Create an object of desired type

FUNCTION INTERFACE

- Used for creating a an output object based on a given input and the logic and possibly chaining with other functions.
- A logic can be packaged as a variable.

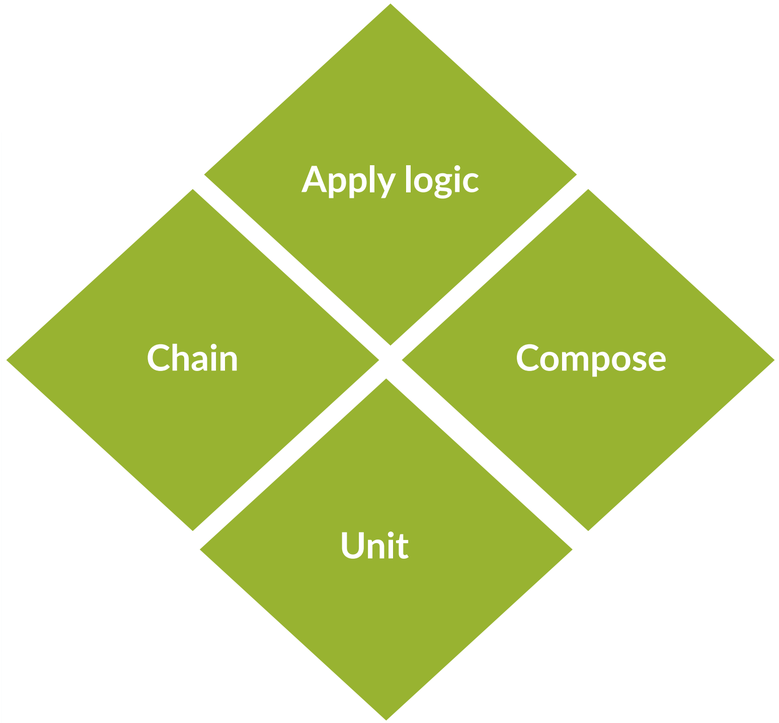


FUNCTION INTERFACE

- Used for creating a an output object based on a given input and the logic and possibly chaining with other functions.

public interface Function<T,R>

R apply(T t)	Applies logic to T and returns an object of R
andThen(Function *after*)	First applies its logic then the logic provided by Function *after*
compose(Function *before*)	First applies before logic then its own logic
identity()	Returns its own input argument



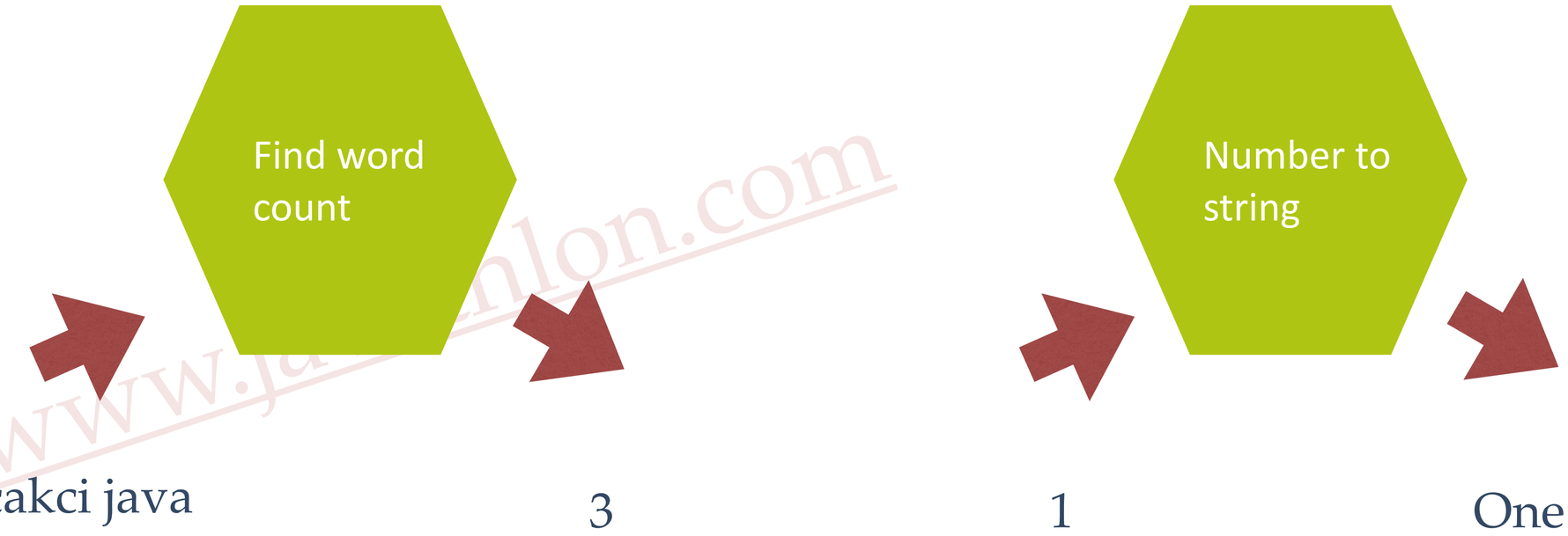
APPLYING A FUNCTION

```
Function<String, Integer> findWordCount = new Function<String, Integer>() {

    @Override
    public Integer apply(String t) {
        return t.split(" ").length;
    }
};
```

```
Integer count = findWordCount.apply("michael phelps broke a new record");
```

CHAINING FUNCTION

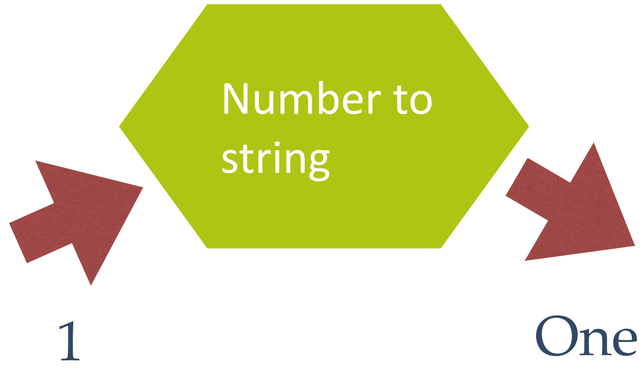
We have 2 functions written before



CHAINING FUNCTION

Compose them as returning the word count in a string as text

CHAINING FUNCTION




```
Function findWordCount = new Function<String, Integer>() {

    @Override
    public Integer apply(String t) {
        return t.split(" ").length;
    }
};
```

```
Function numberToString = new Function<Integer, String>() {

    @Override
    public String apply(Integer t) {
        switch(t.intValue()) {
        case 0 : return "zero";
        case 1 : return "one";
        case 2 : return "two";
        }
        return "Unknown";
    }
};
```

CHAINING FUNCTION

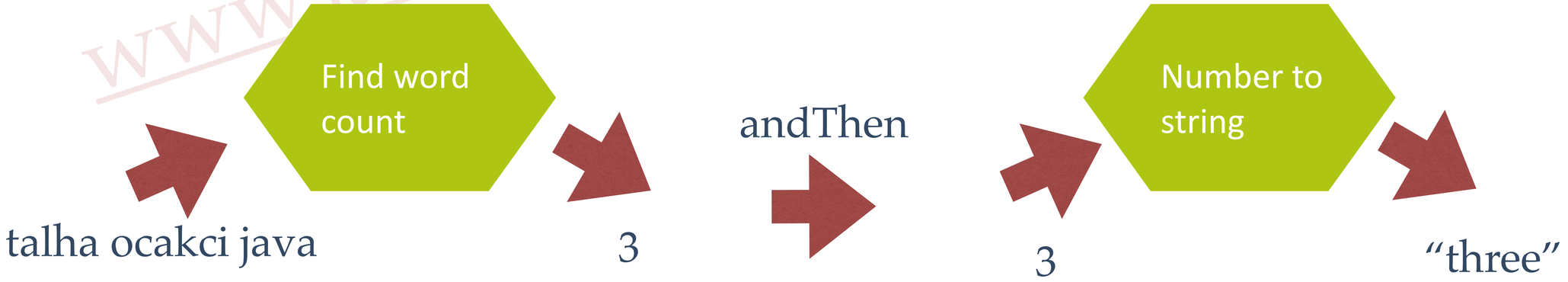
Simple Function invocation:

findWordCount.apply(“talha ocakci java”) => 3

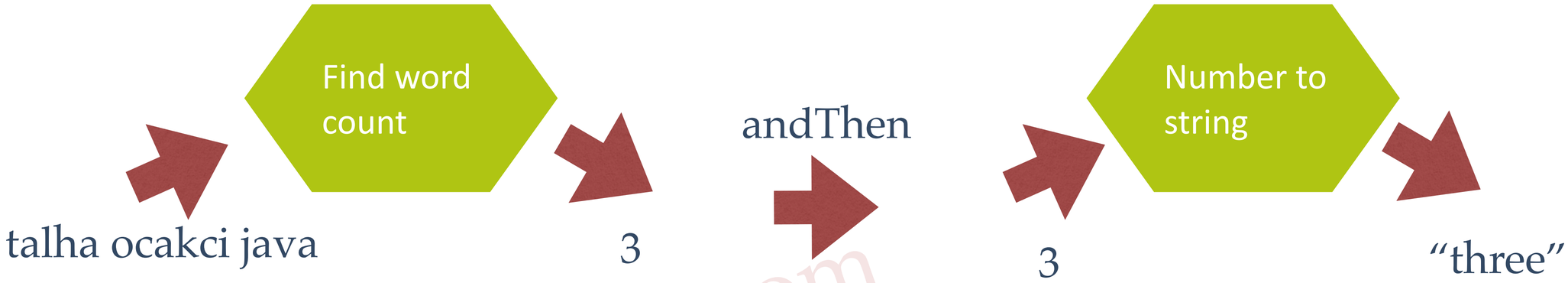
numberToString.apply(2) => “two”

Function Chaining

findWordCount.andThen(numberToString).apply(“talha ocakci java”) => “three”



FUNCTION CHAINING RULE



- Output type of first function must be a super class or the same class of the input of second function’s input

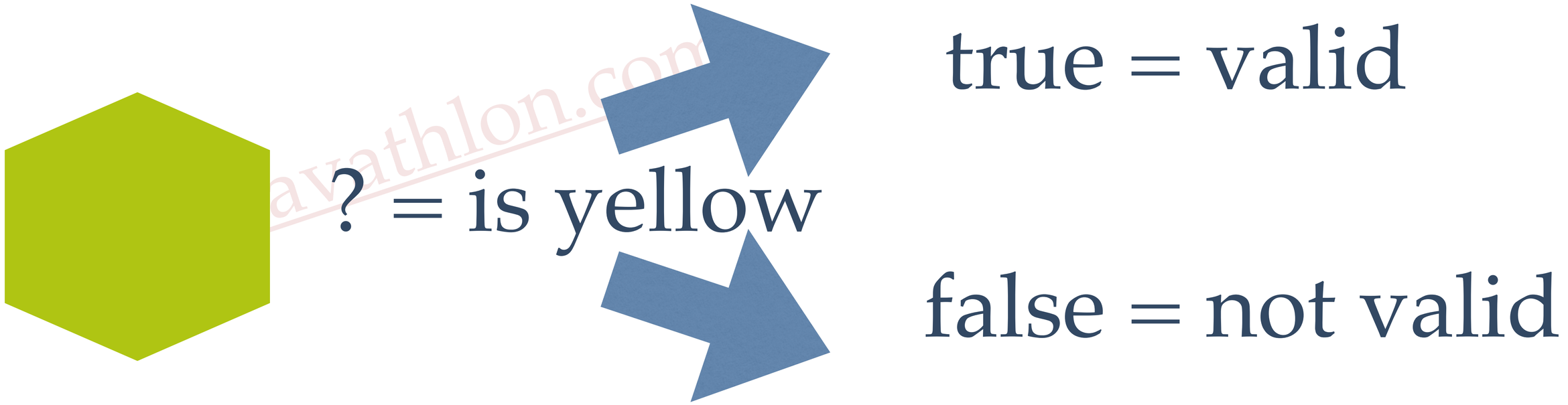
FUNCTION CHAINING RULE

- Otherwise ClassCastException is thrown.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        at com.javathlon.java8.FunctionExample$2.apply(FunctionExample.java:1)
        at java.util.function.Function.lambda$andThen$6(Function.java:88)
        at com.javathlon.java8.FunctionExample.main(FunctionExample.java:30)
```

PREDICATE

- Tests if a data conforms to some value or logic

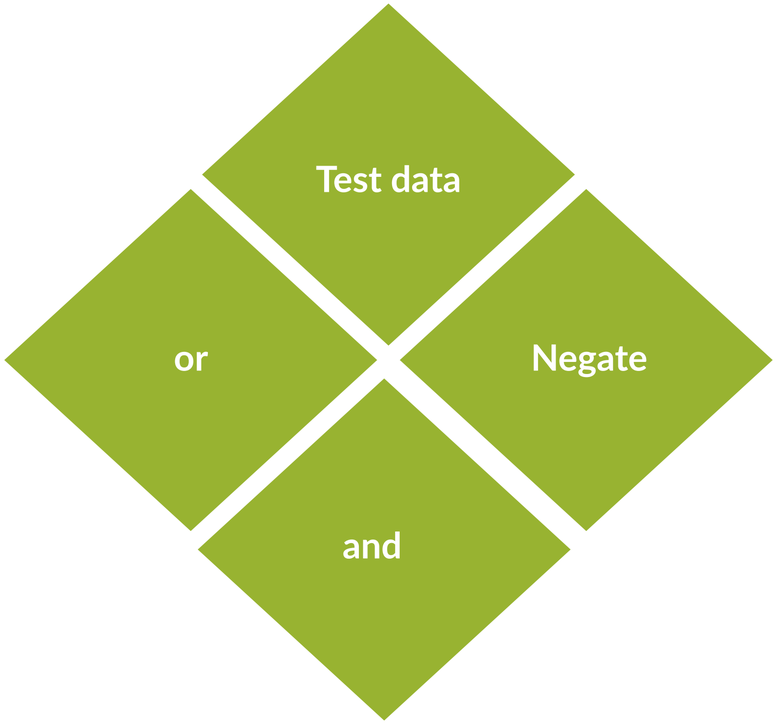


PREDICATE INTERFACE

- Tests if a data conforms to some value or logic
- Logical operations can be used

public interface Function<T,R>

boolean test(T t)	Tests if t conforms to a logic
and(Predicate *otherPredicate*)	Logical and operation with another predicate
or(Predicate *otherPredicate*)	Logical or operation with another predicate
negate()	Logical not operation



USING PREDICATE

```
Predicate<String> specialWordChecker = new Predicate<String>() {
    public boolean test(String t) {
        return t.contains("Download");
    };
};
```

```
Predicate<String> sizeChecker = new Predicate<String>() {
    public boolean test(String t) {
        return t.length() < 50;
    };
};
```

- First predicate tests if given string has word “Download”
- Second predicate tests if text’s length is less than 50.

USING PREDICATE

```
Predicate<String> specialWordChecker = new Predicate<String>() {
    public boolean test(String t) {
        return t.contains("Download");
    };
};
```

- specialWordChecker.test(“Download now”);
- returns true

```
Predicate<String> sizeChecker = new Predicate<String>() {
    public boolean test(String t) {
        return t.length() < 50;
    };
};
```

- sizeChecker.test(“Download now”);
- returns true

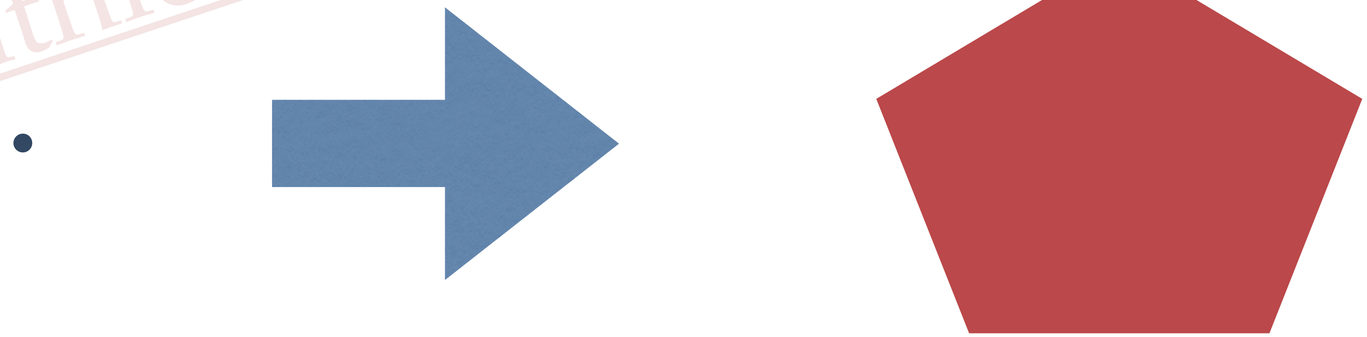
LOGICAL OPERATIONS ON PREDICATE

- Check if the string has less than 50 characters and contains word download by using existing Predicates

- sizeChecker.and(specialWordChecker).test(“Download now”)

SUPPLIER

- Supplies an instance of given type by constructor or some other ways.



SUPPLIER INTERFACE

- Supplies an instance of given type buy constructor or some other ways.

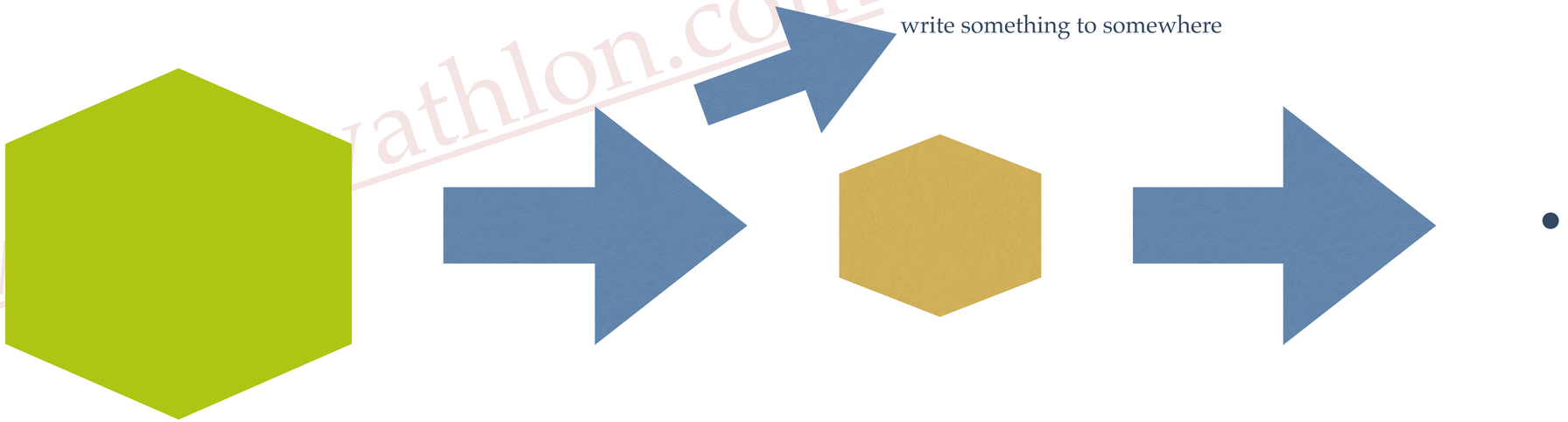
T get()	Supplies you a T instance

USING SUPPLIER

```
Supplier<Calendar> calendarSupplier = new Supplier<Calendar>() {

    @Override
    public Calendar get() {
        return Calendar.getInstance();
    }
};

Calendar c = calendarSupplier.get();
```

CONSUMER

- Gets an object, uses it but does not return a response.
- It possibly side-effects such as changing the value of the object or writing some output to somewhere.



CONSUMER INTERFACE

- Gets an object, uses it but does not return a response.
- It possibly side-effects such as changing the value of the object or writing some output to somewhere.

void accept(T t)	Uses instance t, modifies it if necessary Does something without returning an output
void andThen(Consumer *otherconsumer*)	chains consumers

USING CONSUMER

```
Consumer<Date> oneDayIncrement = new Consumer<Date>() {

    @Override
    public void accept(Date date) {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        calendar.add(Calendar.DAY_OF_YEAR, 1);
        date = calendar.getTime();

    }
};

Consumer<Date> dayPrinter = new Consumer<Date>() {

    @Override
    public void accept(Date date) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy EEEE");
        System.out.println(sdf.format(date));
    }
};

oneDayIncrement.andThen(dayPrinter).accept(new Date());
```

LAMBDA EXPRESSIONS

- Simplifying the usage of Function, Predicate, Consumer, Supplier interfaces lambda expressions are used.
- All methods of these interfaces may be changed with these : **->**
- **This expression may be referred with the proper interface reference.**
- **JVM understands which method to use.**

CONVERTING A DEFINITION TO LAMBDA EXPRESSION

```
Predicate<String> specialWordChecker = new Predicate<String>() {
    public boolean test(String t) {
        return t.contains("Download");
    };
};
```


t -> t.contains(“Download”)

VALID LAMBDA EXPRESSIONS

(parameters) -> expression	(List<String> list) -> list.isEmpty()
(parameters) -> {return … ;}	(int a, String b) -> { **System.out.println(a);** return b +a **; }**
(parameters) -> statements	(int a, int b) -> a * b
(parameters) -> statements	(int a, int b) -> {return a * b;}

LAMBDA EXPRESSIONS

- -> automatically detects the proper interface, according to the return value and then invokes the proper method.
- t -> t.contains(“Download”)
- Since right hand side returns a boolean, JVM understands that it should use a Predicate. So implicitly creates a Predicate and then invoke test() method.

LAMBDA EXPRESSIONS

- -> automatically detects the proper interface, according to the return value and then invokes the proper method.
- t -> System.out.println(“t + “is my string””)
- Since right hand side returns nothing, JVM implicitly creates a Consumer and invokes apply method automatically.

LAMBDA EXPRESSIONS

- -> automatically detects the proper interface, according to the return value and then invokes the proper method.
- t -> t * 3;
- Since right hand side accepts an input and returns a value, JVM implicitly creates a Function instance and invoke apply method automatically.

REFERRING TO A LAMBDA EXPRESSION

```
Predicate<String> wordChecker = s -> s.contains("Download");

boolean containsWord = wordChecker.test("Download e-book");

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

REFERENCING TO LAMBDA EXPRESSION

```
Function<String> consumer = (String s) -> { System.out.println(s); };
```

- This fails because right hand side of lambda returns nothing. Then it is not a Function but Consumer. Thus, below reference is correct.


```
Consumer<String> consumer = (String s) -> { System.out.println(s); };
```

REFERENCING TO A FUNCTION

1

```
Function<String, Integer> findWordCount = new Function<String, Integer>() {

    @Override
    public Integer apply(String t) {
        return t.split(" ").length;
    }
};
```

2

```
Function<String, Integer> function = (String s) -> { return s.split(" ").length;};

function.apply("Some sentence");
```

3

```
Function<String, Integer> function = s -> { return s.split(" ").length;};
```

FUNCTION CURRYING

- Currying is evaluating function arguments one by one, producing a new function with one argument less on each step.

METHOD AND CONSTRUCTOR REFERENCES (:: operators)

- You don’t have to write a function from scratch every time. You may refer to a method or constructor of a class or instance.
- **::** operator is used for accessing static method of a class or an instance method

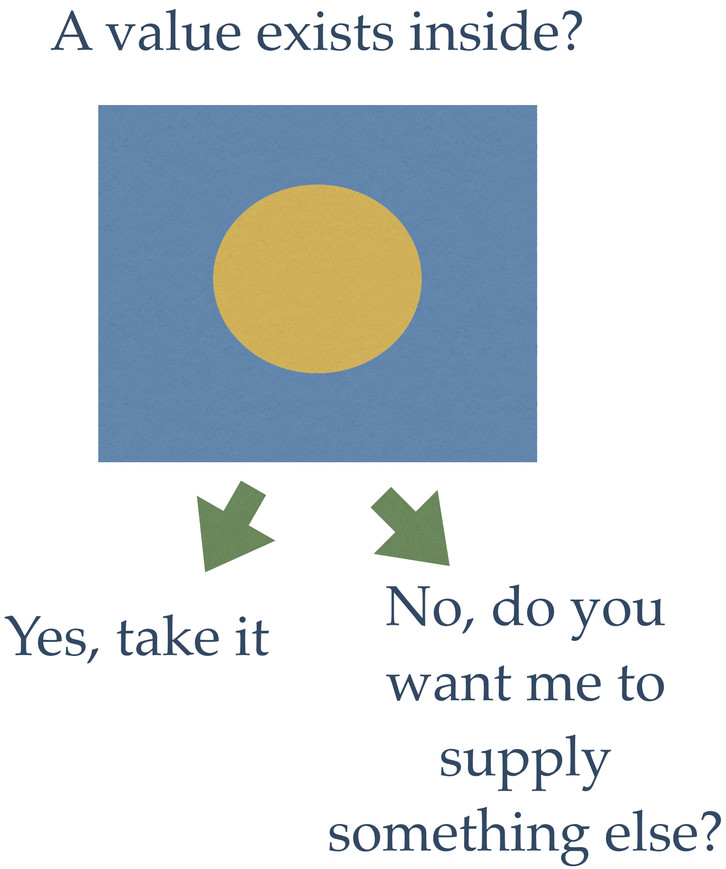
AVOIDING NULL

- Null is a value that is not actually a value.
- **Hard to determine whether the value not present or just has no value**
- Null is mentioned as “millon dollar mistake”
- Functional programming avoids null
- Instead, each language has a structure for determining if a reference holds a real value or not.
- It is **Optional** in Java.

```
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";
}
```

OPTIONAL

- In functional programming, we chain functions. If somewhere in the chain, null is obtained, NullPointerException may be thrown.
- Between the chain items, we should be able to check if a value present.
- This is done by ifPresent() method of an Optional instance.



OPTIONAL

- Optional<String> myOptional = Optional.of(“test”);
- This says, a string may exist inside. get it with get() method
- If a value presents get it, or supply it with something else getOrElse(***Supplier s***)
- Check if a value presents with isPresent()
- Do something if value presents ifPresent(***Consumer consumer***)

STREAM

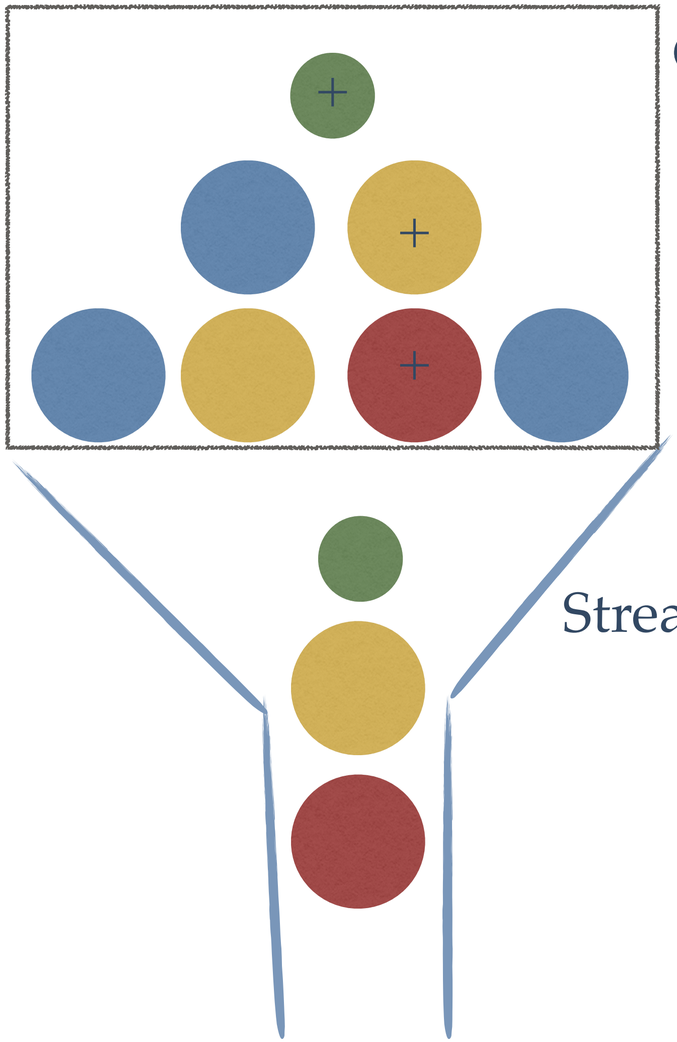
Imperative vs. Functional
Separation of Concerns

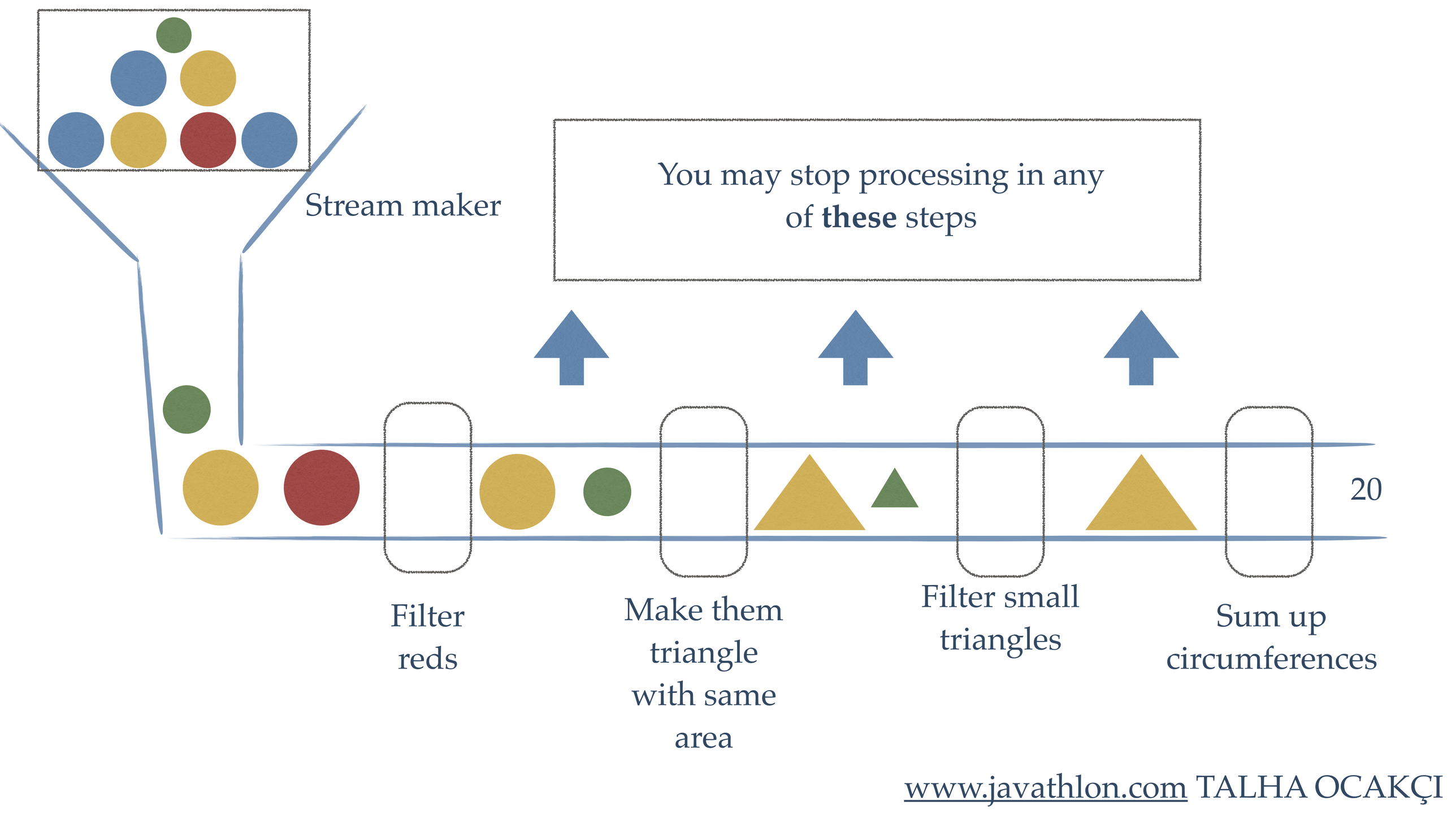
```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
     if (line.startsWith("ERROR")) {
          errors.add(line);
          errorCount++;
     }
     line = file.readLine();
}
```

```
List<String> errors =
          Files.lines(Paths.get(fileName))
               .filter(l -> l.startsWith("ERROR")
               .limit(40)
               .collect(toList());
```

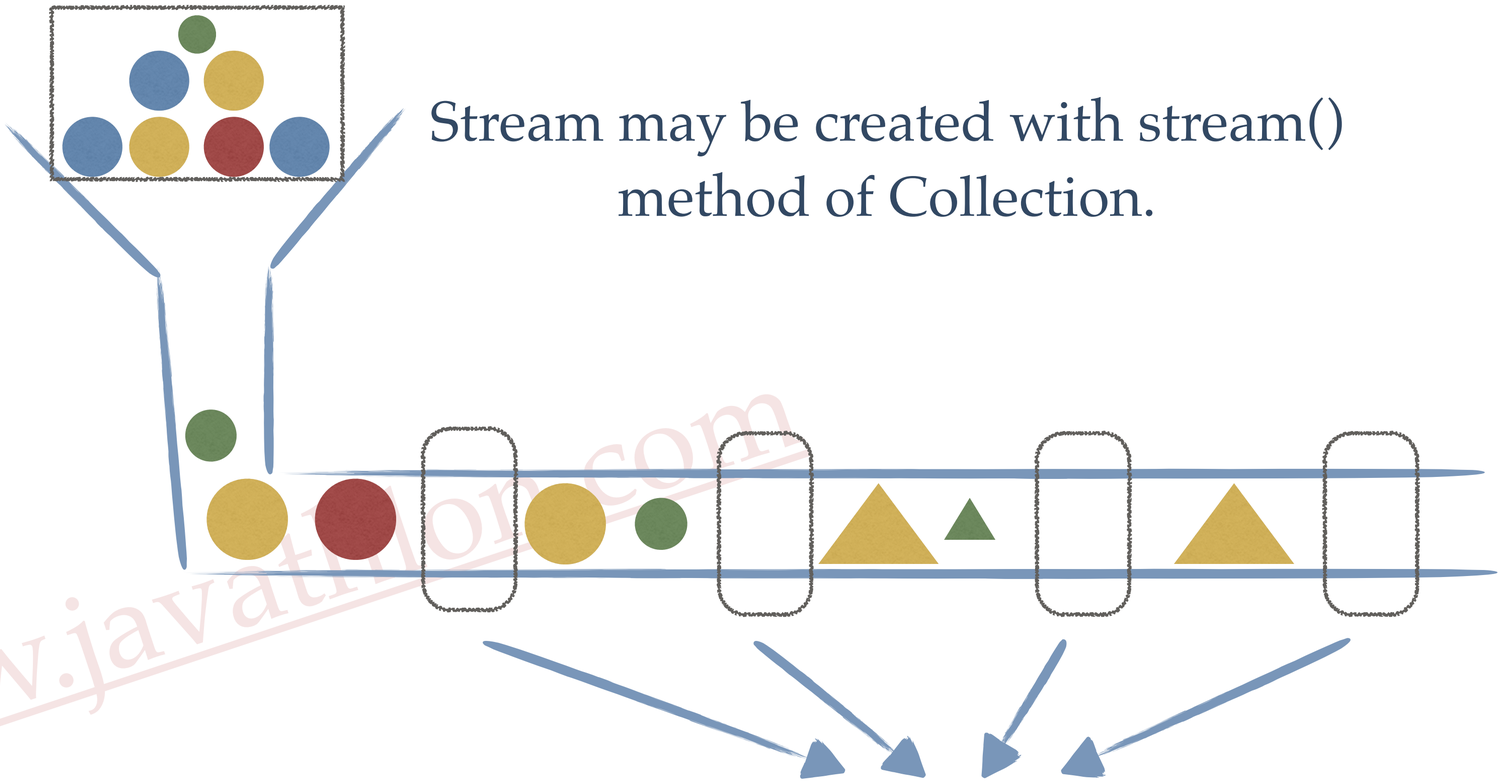
STREAM

- Stream is the structure for processing a collection in functional style.
- Original collection is not modified.
- Stream may be processed only once. After getting an output you can not use it again.




Stream maker
You may stop processing in any of **these** steps
20
Filter reds
Make them triangle with same area
Filter small triangles
Sum up circumferences

STREAM IN JAVA 8

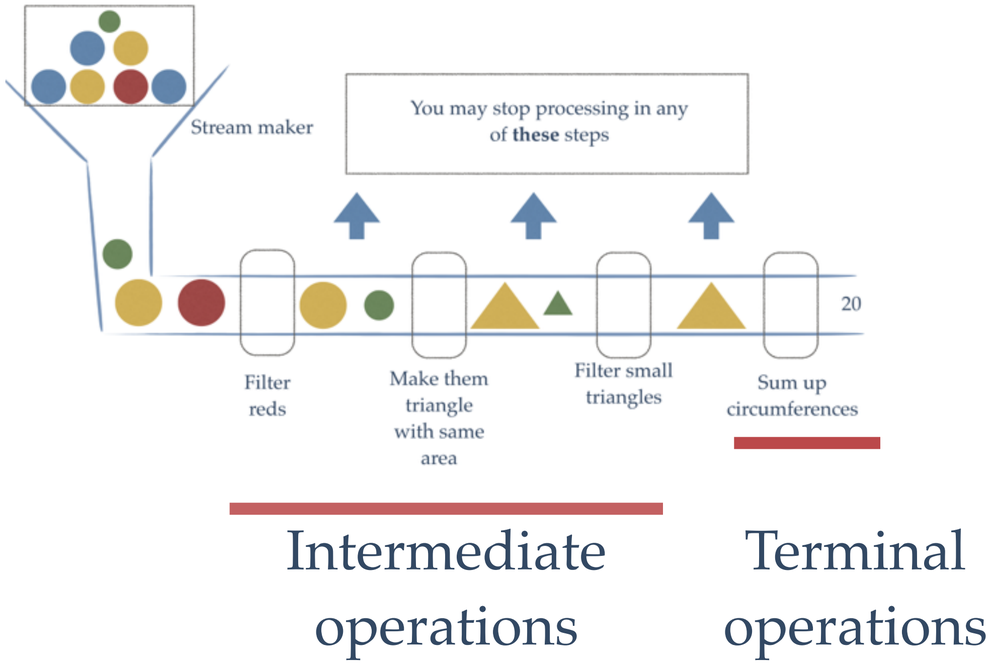


Stream may be created with stream() method of Collection.

Each of these boxes is an functional interface:
Function, Mapper, Supplier, Consumer, Predicate

INTERMEDIATE AND TERMINAL OPERATIONS

- Intermediate operations yield a new Stream and you may pipeline(chain) the operations
- Terminal operations yield a result other than a stream: A list, Integer etc…
- **After a terminal operation, you can not use the stream again and you can not pipeline another function.**



INTERMEDIATE OPERATIONS

Stream Operation	Goal	Input
filter	Filter items according to a given predicate	Predicate
map	Processes items and transforms	Function
limit	Limit the results	int
sorted	Sort items inside stream	Comparator
distinct	Remove duplicate items according to equals method of the given type	

TERMINAL OPERATIONS

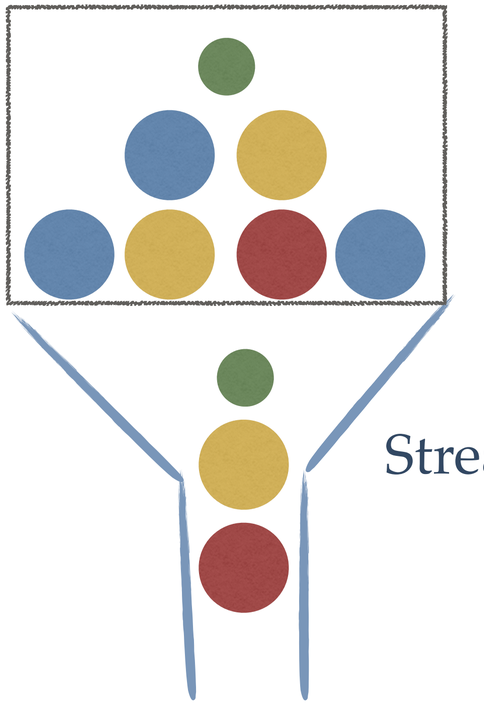
STREAM OPERATION	GOAL	INPUT
forEach	For every item, outputs something	Consumer
count	Counts current items	
collect	Reduces the stream into a desired collection	

INITIALIZE A STREAM

- You may initialise a stream by using
  - An existing ***Collection***'s stream() method
  - From scratch

CONVERTING COLLECTIONS TO STREAM

- You may use stream() method of any collection.
- Original collection is not effected during the process
- You can’t process a stream after you get an output.



CONVERTING COLLECTIONS TO STREAM

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(4);
list.add(5);

Stream<Integer> stream = list.stream();

stream.filter(t->t >=4)
      .forEach(i -> System.out.println(i * i));
```

CREATE STREAM FROM SCRATCH

```
Stream<Integer> s = Stream.of(3, 4, 5);

s.filter(t->t >=4)
  .forEach(i -> System.out.println(i * i));
```

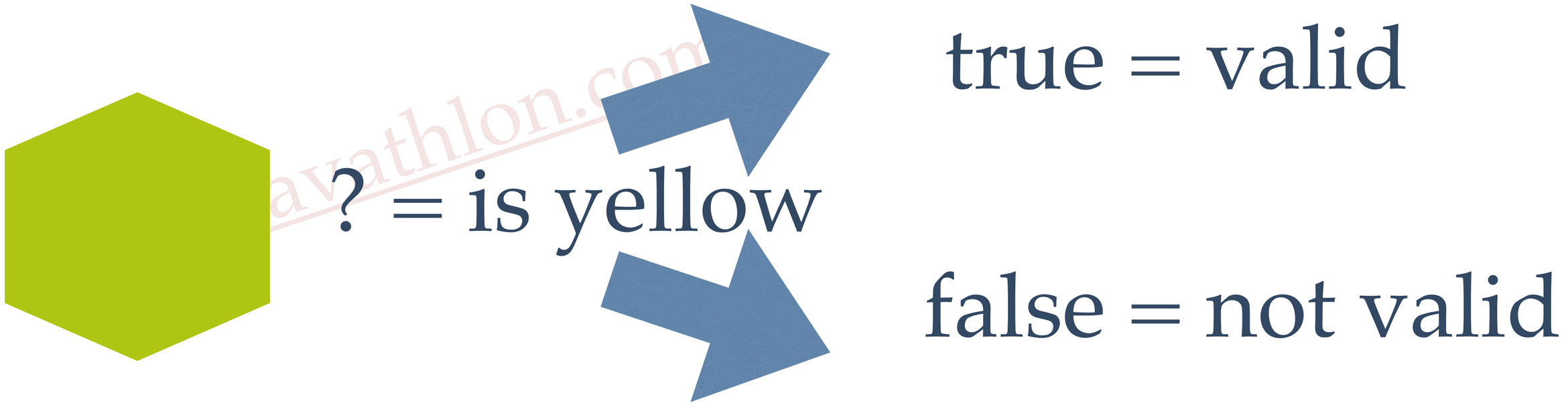
- You may use of() method of stream. Put the values as comma separated.

```
IntStream intStream = IntStream.range(1, 5);

IntStream intStream2 = IntStream.range(10, 50).skip(2);

intStream.forEach(i -> System.out.println(i));

intStream2.forEach(i -> System.out.println(i));
```

- Use IntStream, DoubleStream, LongStream and range() method of them.

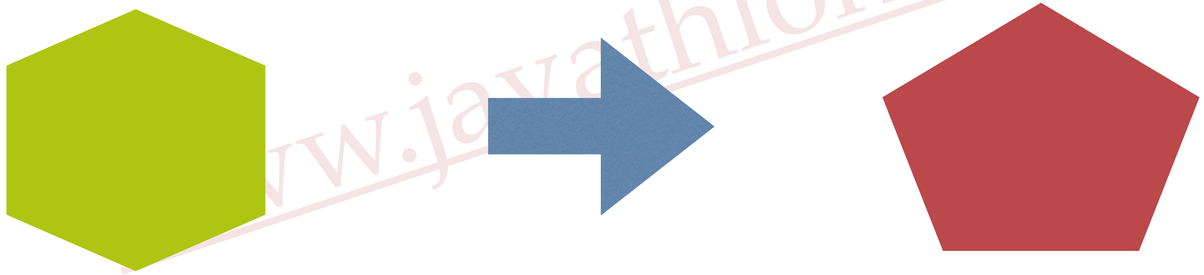
FILTER

- Includes the items that conform to given Predicate and creates a new stream.



MAP

- All items are converted to a new state or a new type.
- Logic is assigned with a **Function**.

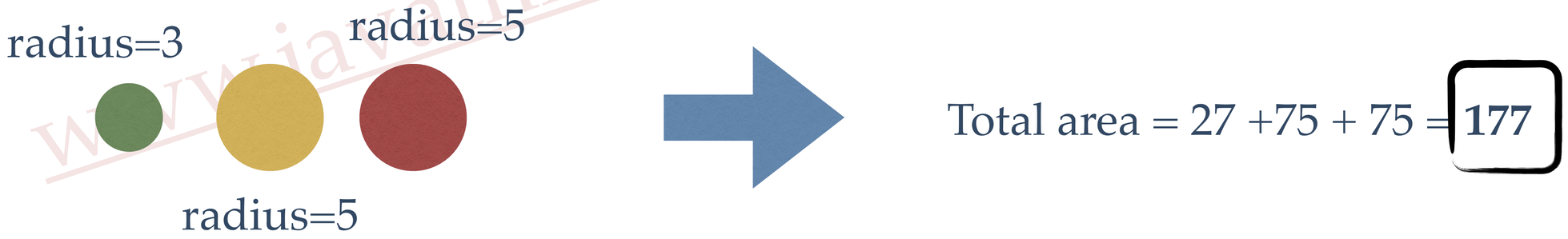


REDUCTION

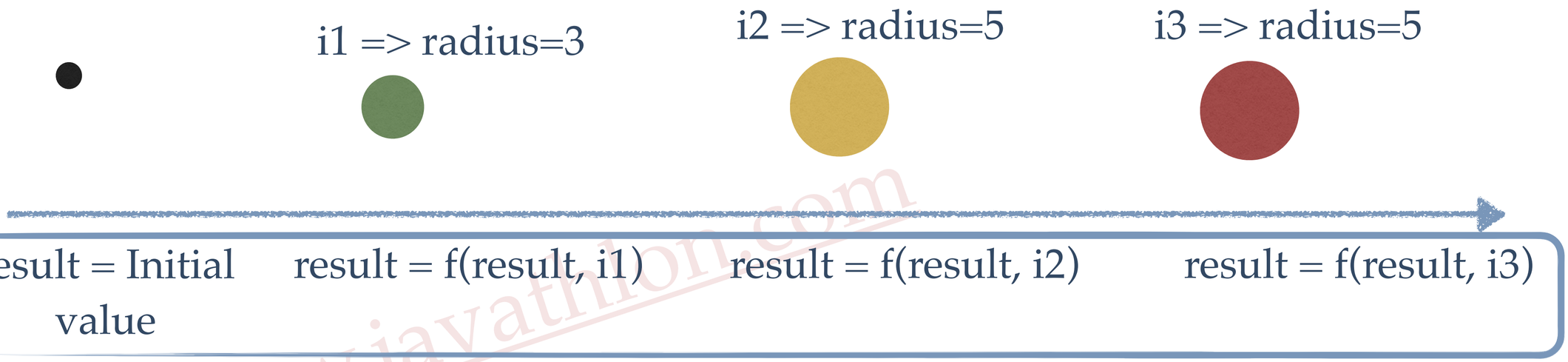
Combining items according to a logic

WHAT IS REDUCTION?

- A **reduction** operation (also called as **fold**) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list.



REDUCING THE STREAM



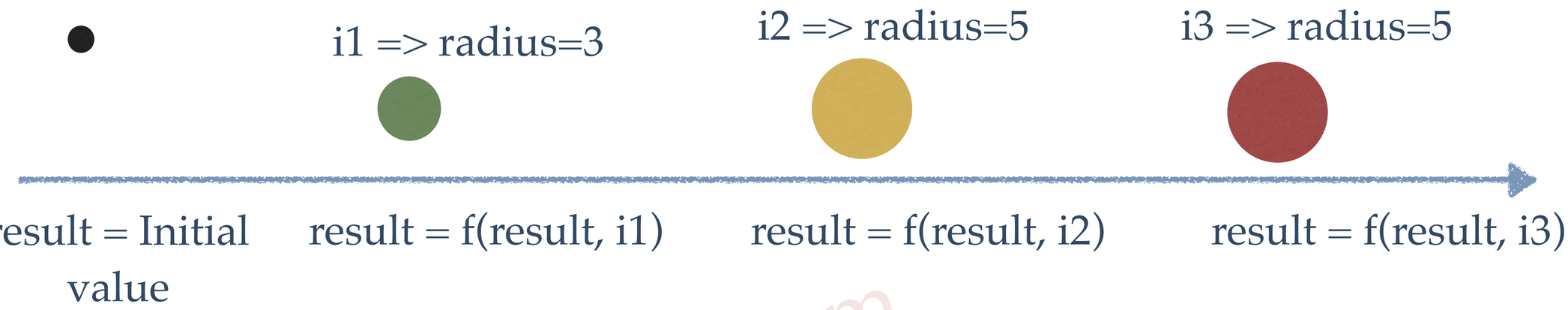
f(a, b) = a + (b.radius)*(b.radius)*PI

f(0, i1) = 0 + 3 *3 * 3 = 27

f(27, i2) = 27 + 5* 5*3 = 102

f(102, i3) = 102 + 5*5*3 = 177

REDUCING THE STREAM



stream. reduce(initial_value, f(a, b))

```
stream. reduce(0, (a,b) -> a + b * b * PI)
```

This would be enough if operations were sequential only. But streams must be processable in parallel.

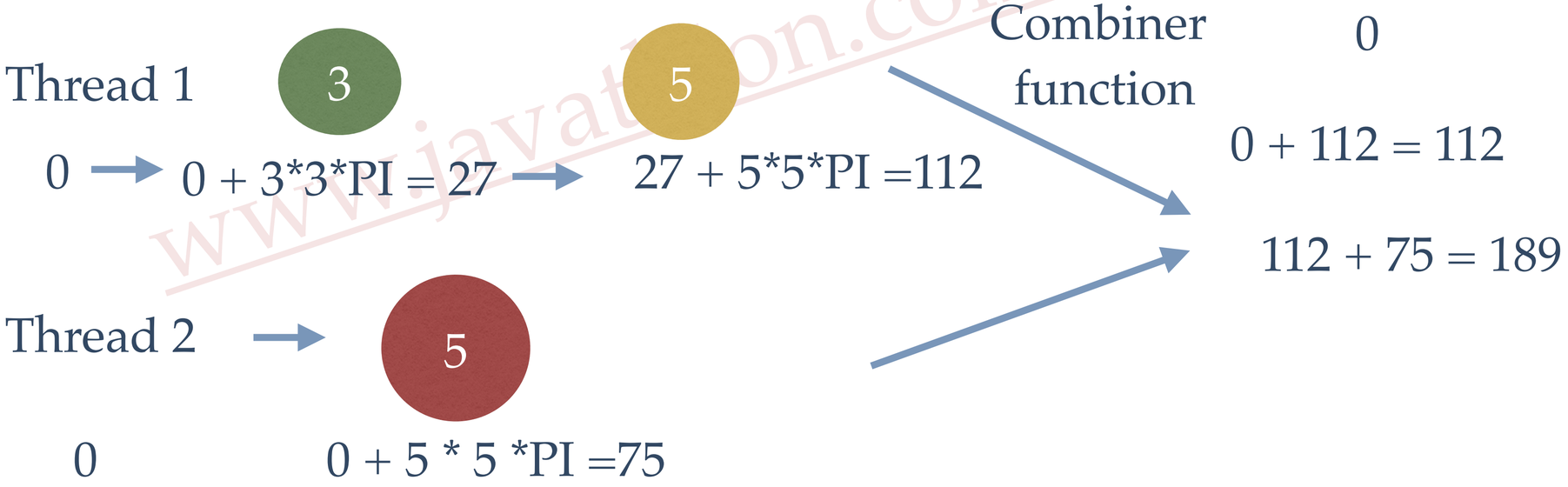
PARALEL REDUCING OF STREAM

- **You must implement the reducing operation as parallel-ready.**
- That's why, reduce() method has a third parameter for combining the results of parallel operations.
- Below code does not compile because JVM does not know what to do if stream is a parallel stream. (a, b) -> a + b defines only the sequential reduce operation.


```
double areaSum = stream.parallel().reduce(0.0,
    (a,b) -> a + b.radius * b.radius * Math.PI);
```

PARALLEL REDUCE STREAM

- The third parameter is the accumulator and defines how to combine the intermediate results of several threads.



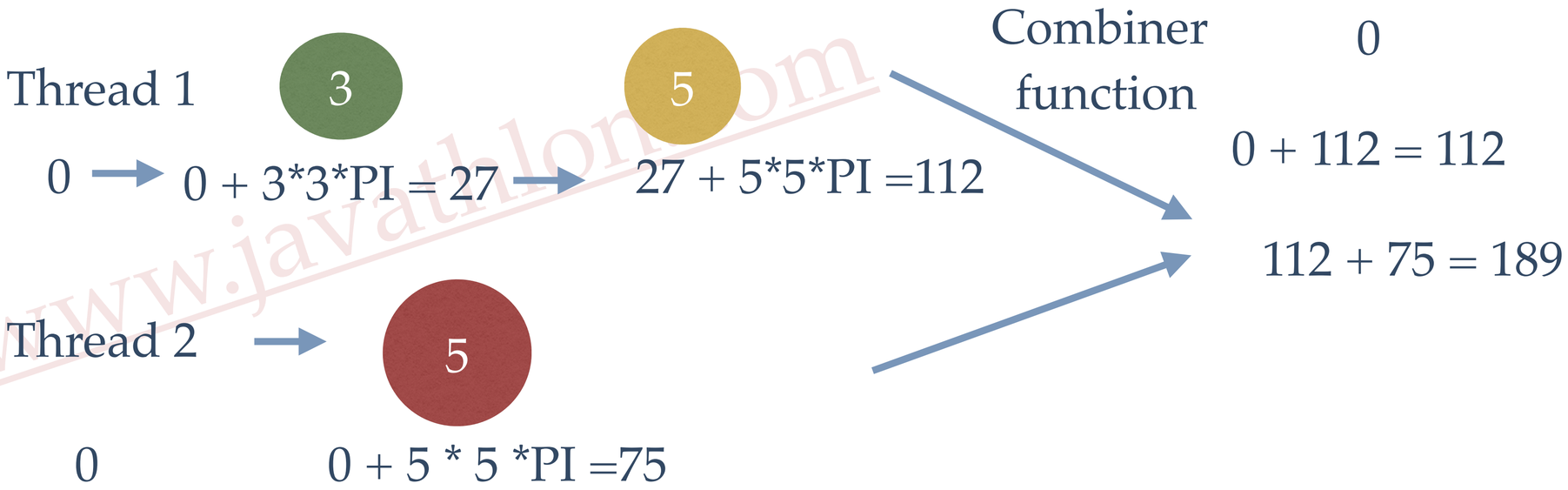
PARALLEL REDUCE STREAM

```
double areaSum = stream.parallel().reduce(0.0,
    (a,b) -> a + b.radius * b.radius * Math.PI,
    (a, b) -> (a + b));
```

Initial seed for both functions

Accumulator function

Combiner function



OTHER OPERATIONS

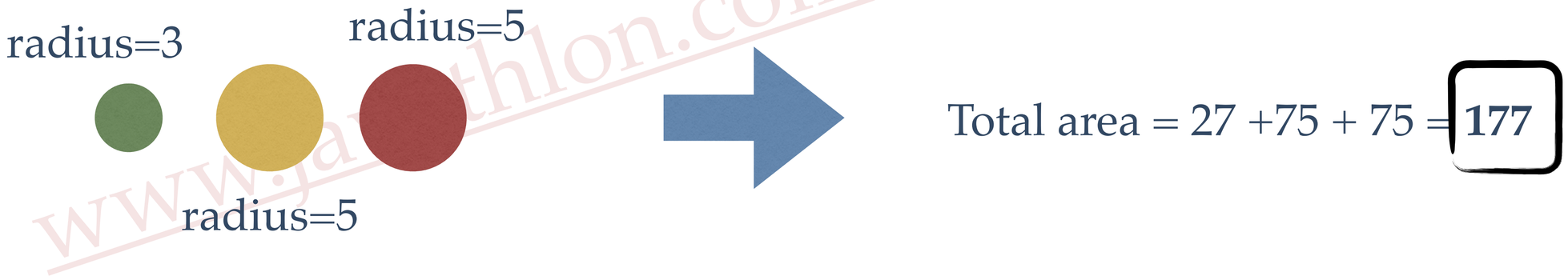
distinct()	Using equals() method of a class, filters duplicate items
sorted(Comparator c)	Using given Comparator, sorts the stream
count()	Find the item count in stream

COLLECTOR

- Collector is a **reducer** operation.
- Reducer: takes a sequence of input elements and combines them into a single summary result
- Result may be **one single collection or any type of one object instance**

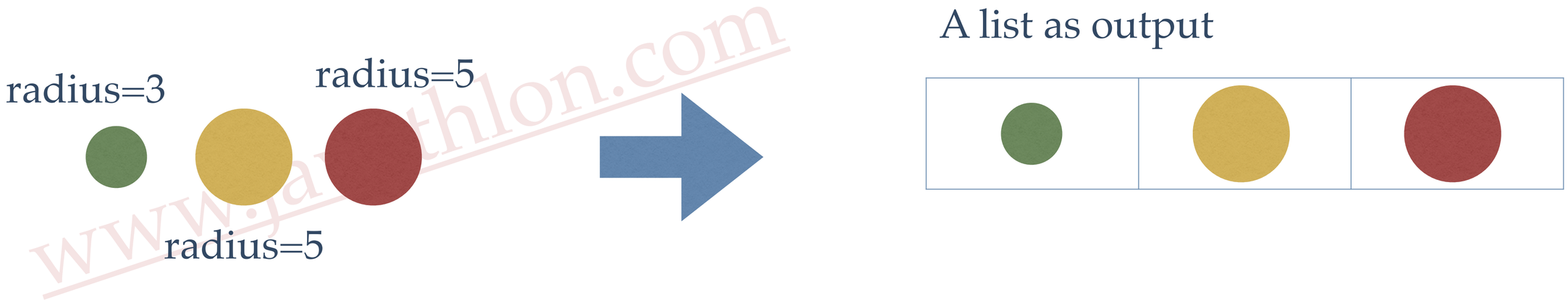
WHAT COLLECTOR DOES

- Gets the desired values of each item, process and returns a single result



WHAT COLLECTOR DOES

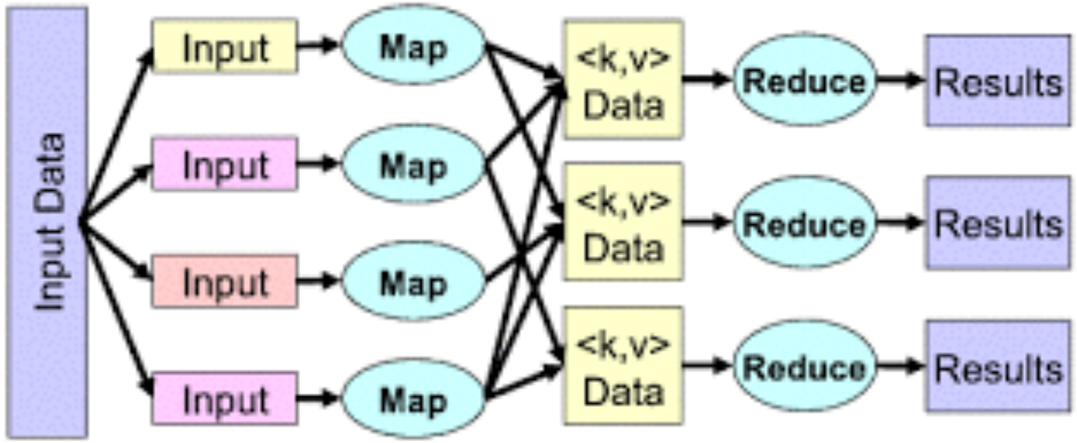
- Converts the stream into a collection (List, Set, Map).



COLLECTORS

```
interface Collector<T,A,R> {
    Supplier<A>          supplier()
    BiConsumer<A,T>      accumulator()
    BinaryOperator<A>    combiner()
    Function<A,R>        finisher()
    Set<Characteristics> characteristics()
}
```

supplier: starting object
accumulator: - mutable data structure that we will use to accumulate input elements of type T.
combiner: used to join two accumulators together into one. used in parallel mode.
finisher: takes an accumulator A and turns it into a result value



COLLECTORS.TOLIST()

- Collectors.toList() collector
  - Starts with an empty ArrayList
  - Append each item to this ArrayList in sequential mode
  - Add each intermediate ArrayList to total ArrayList while combining the results


```
new CollectorImpl<>((Supplier<List<T>>) ArrayList::new,
    List::add,
   (left, right) -> { left.addAll(right); return left; });
```

→ Initial seed for both functions

→ Accumulator function

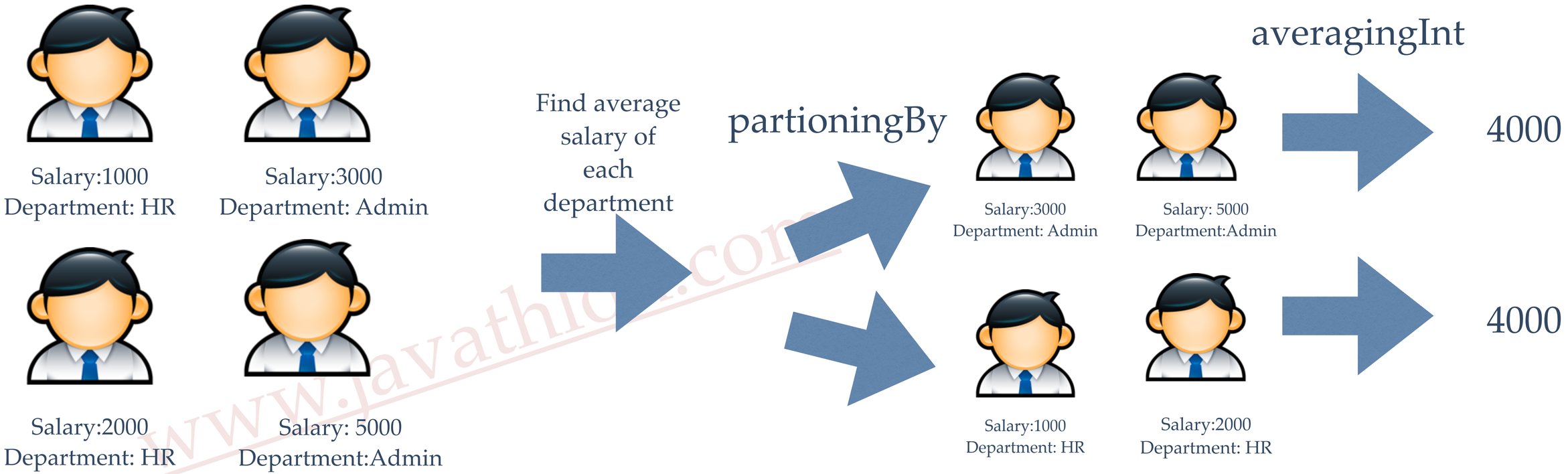
→ Combiner function

COLLECTORS

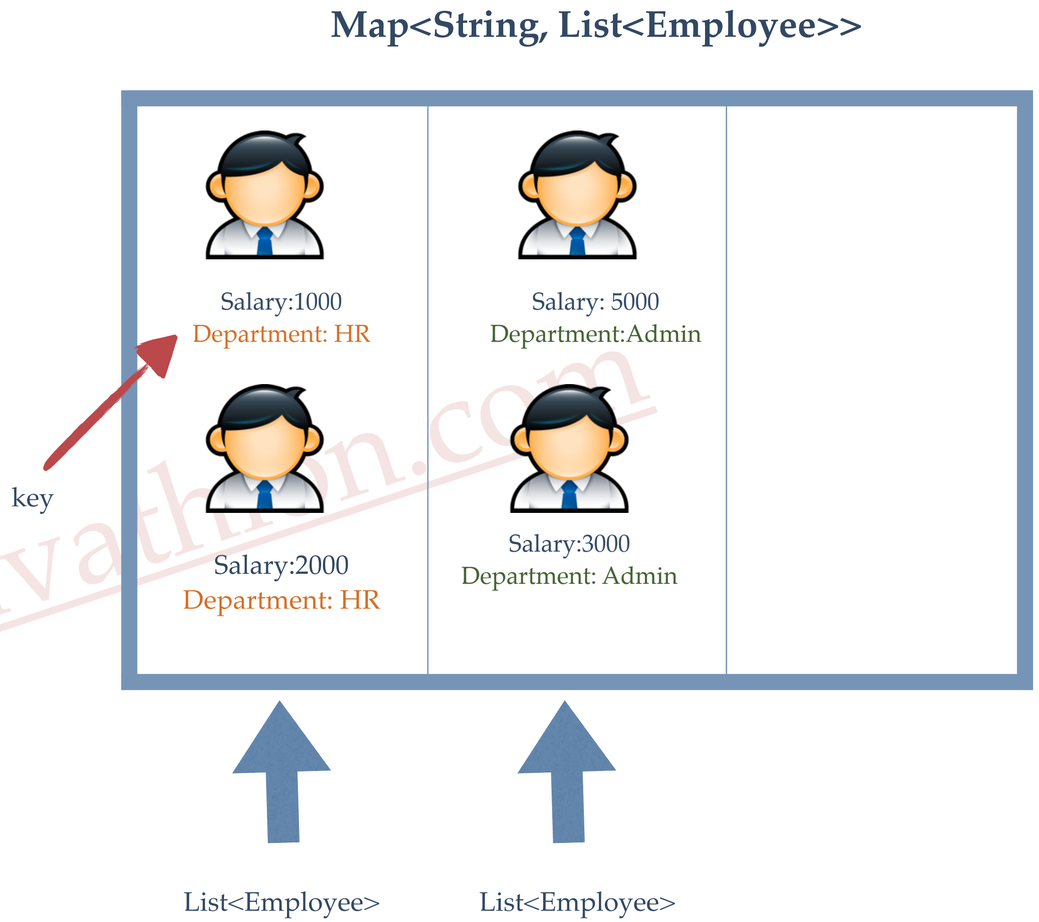
- Prebuilt **Collector** implementations that perform common tasks:

Collectors.toList()	Puts items into a list
Collectors.toCollection(*TreeSet::new*)	Puts items into a desired container. Container is supplied with a supplier
Collectors.joining(", “)	Joins multiple items into a single item by concatenating them
Collectors.summingInt(*item::getAge*)	Sums the values of each item with given supplier
Collectors.groupingBy()	Groups the items with given classifier and mapper
Collectors.partitioningBy()	Partitions the items with given predicate and mapper

EXAMPLE

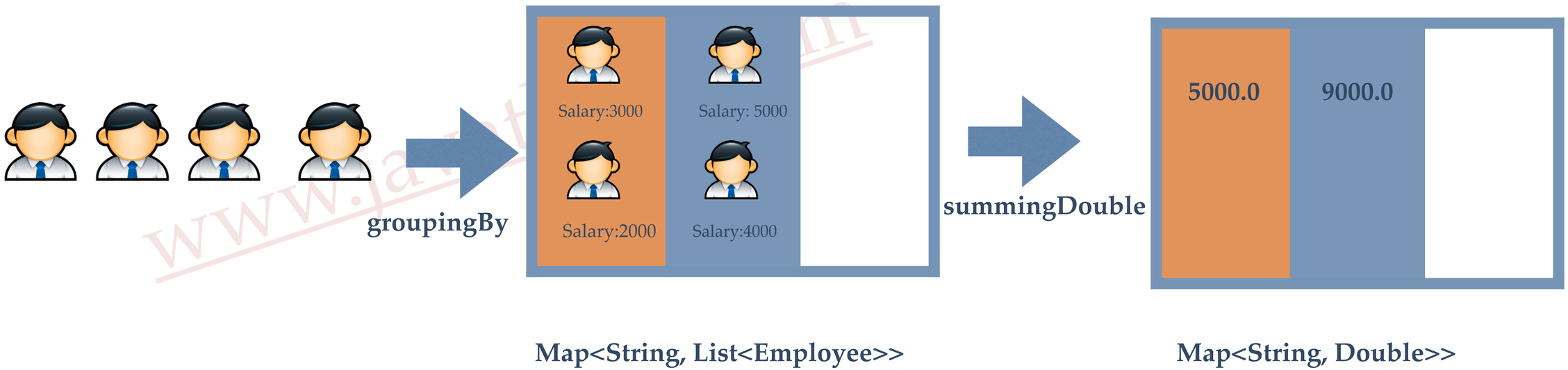


RESULT OF GROUPING BY

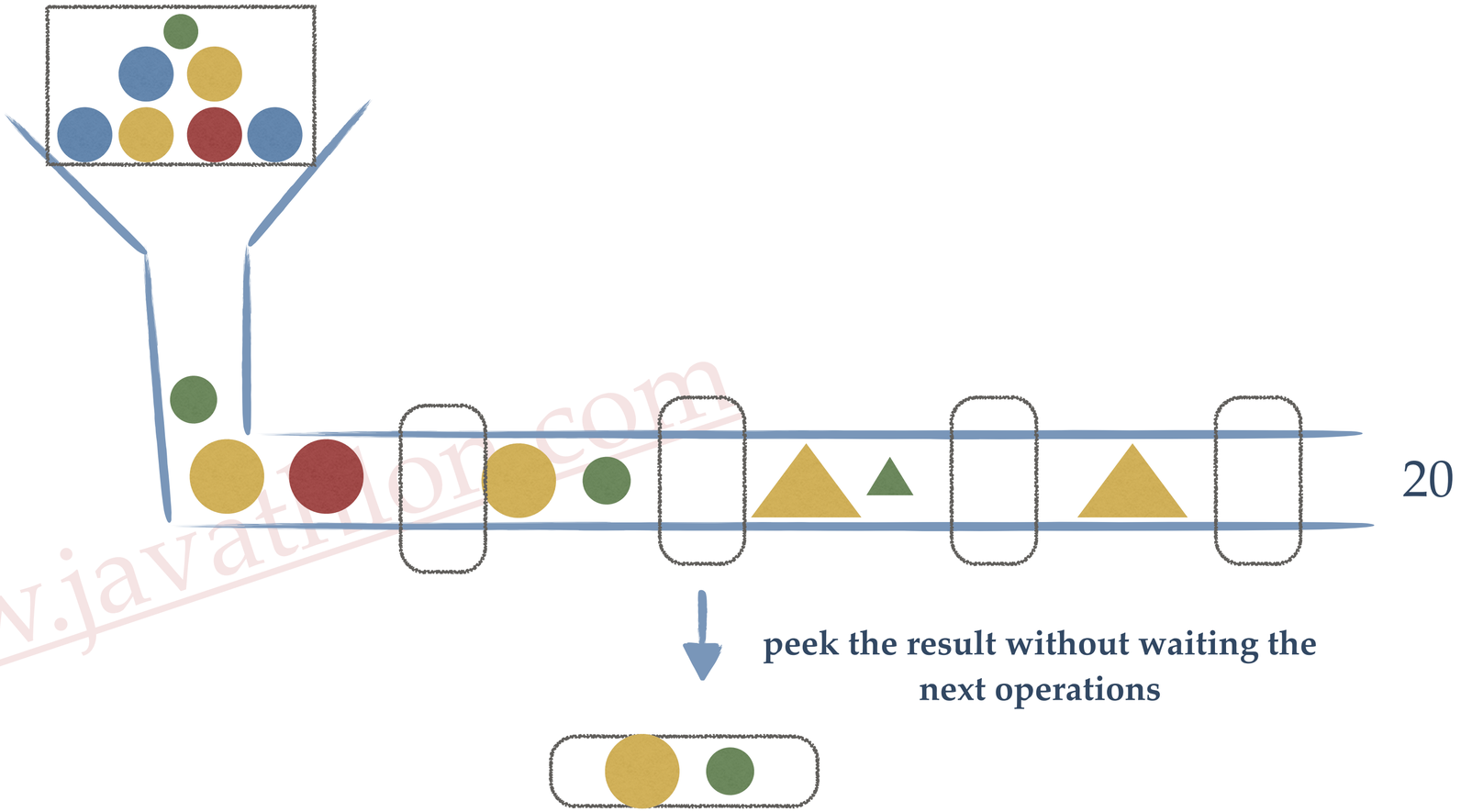


PIPELINING REDUCERS

```
Map<String, Double> salarymap = employeeList.stream()
 .collect(Collectors.groupingBy(Employee::getDepartment,
          Collectors.summingDouble(Employee::getSalary)));
```



PEEKING INTERMEDIATE RESULTS



PEEKING ELEMENTS

- In any intermediate operation, you may get the intermediate result.
- Below code, adds each item in stream after they are multiplied by two.

```
AtomicInteger atomicX = new AtomicInteger(0);

IntStream.range(0, 1000)
    .map(t -> t * 2).peek(t -> atomicX.addAndGet(t))
    .summaryStatistics().getSum();
```

CLOSURE

- Closure means that, if a method contains another method; outer variables should be accessible to the inner method.

```
outer = function() {
  var a = 1;
  var inner = function() {
    alert(a);
  }
  return inner; // this returns a function
}

var fnc = outer(); // execute outer to get inner
fnc();
```

Image is from:
http://stackoverflow.com/questions/36636/what-is-a-closure

CLOSURE IN JAVA 8

- A lambda expression is a function. So A lambda expression should be able to access the variables in outer scope

```
int x = 0;

long sum = IntStream.range(0, 1000)
    .map(t -> t * 2).peek(t -> x = x + t)
    .summaryStatistics().getSum();
```

- We may access but only if this variable is final (read-only)

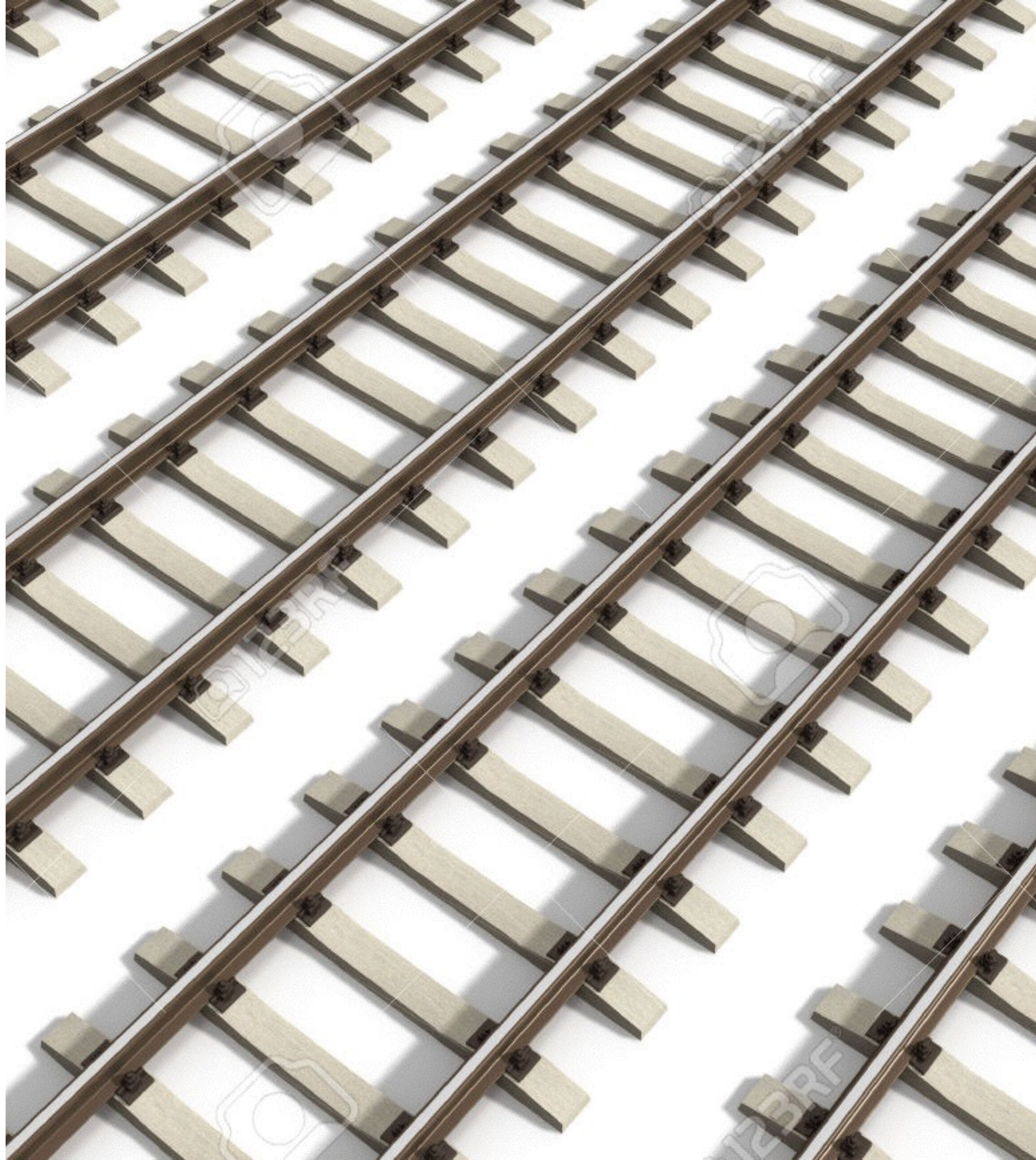
CLOSURE IN JAVA 8

- What if I really want to modify the local variable?
- You may use Atomic classes from Java 7.


```
AtomicInteger atomicX = new AtomicInteger(0);

IntStream.range(0, 1000)
    .map(t -> t * 2).peek(t -> atomicX.addAndGet(t))
    .summaryStatistics().getSum();
```

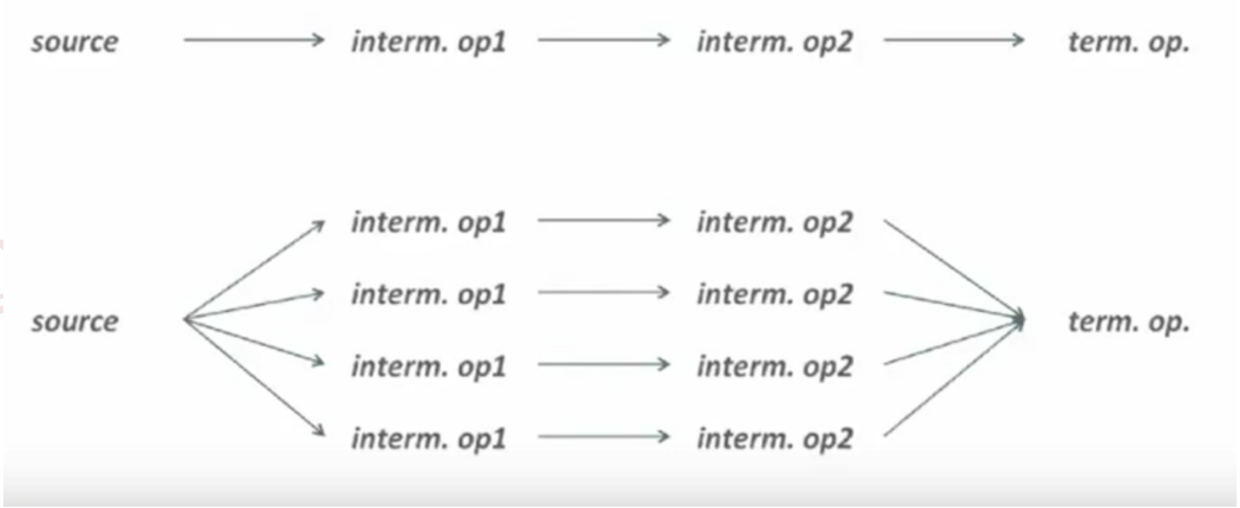
PARALLELISM

Automatic usage of fork-join framework



PARALLEL STREAMS

- Parallel streams automatically distribute the tasks to several cores and combine the results by using fork-join framework.



PARALLEL STREAMS BENCHMARKING

- Benchmarking is done by JMH library.
- JMH library warms up the JVM with 10 iterations and then 10 real iteration.
- Calculates in one second, how many times this method can be completed. Because I have selected the output type as “throughput”.
- The only difference in compared codes is the sequential stream processing and parallel stream processing.

PARALLEL PROCESSING

```
LongStream stream = LongStream.rangeClosed(1, 2_000_000_000);
return stream.parallel().map(t -> t +1).sum();
```

JMH result	Type	Score	Error	Units
Sequential	throughput	0.711	± 0.011	ops/sec
Parallel	throughput	2.260	± 0.424	ops/sec

Parallel stream operated 3 times faster with 2 billion items.

PARALLEL PROCESSING

```
LongStream stream = LongStream.rangeClosed(1, 2_000_000_000);
return stream.parallel().map(t -> t +1).max().getAsLong();
```

JMH result	Type	Score	Error	Units
Sequential	throughput	0.526	± 0.006	ops/sec
Parallel	throughput	1.845	± 0.054	ops/sec

max() operation also operated 3 times faster in parallel mode with 2 billion items.

IS IT ALWAYS THAT GOOD?

```
LongStream stream = LongStream.rangeClosed(1, 2_000);
return stream.parallel().map(t -> t +1).sum();
```

JMH result	Type	Score	Error	Units
Sequential	throughput	**106058.484**	± 1128.029	ops/sec
Parallel	throughput	**33520.489**	± 2358.957	ops/sec

- If item count is not that big, overhead of parallel processing kick offs will decrease the overall performance. **In this code, parallel processing is 3 times slower than the sequential one with only 2000 items.**

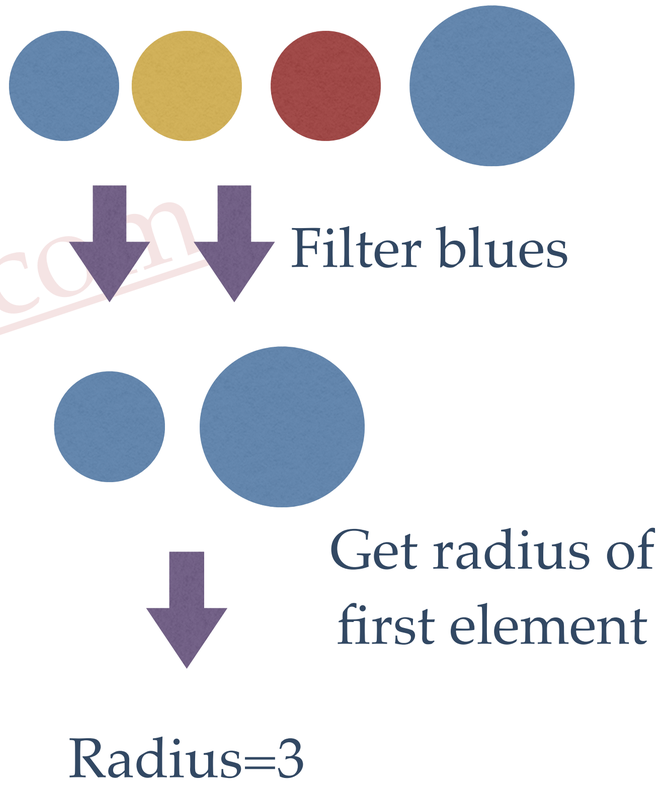
.parallel() miracle?

- You invoke parallel() method on the stream and expected a true and faster process. If so, you will be miserable.
- You must first
  - transform your code into thread-safe structure.
  - reduce blocking parts of the code
  - obey the rules of monads in your code

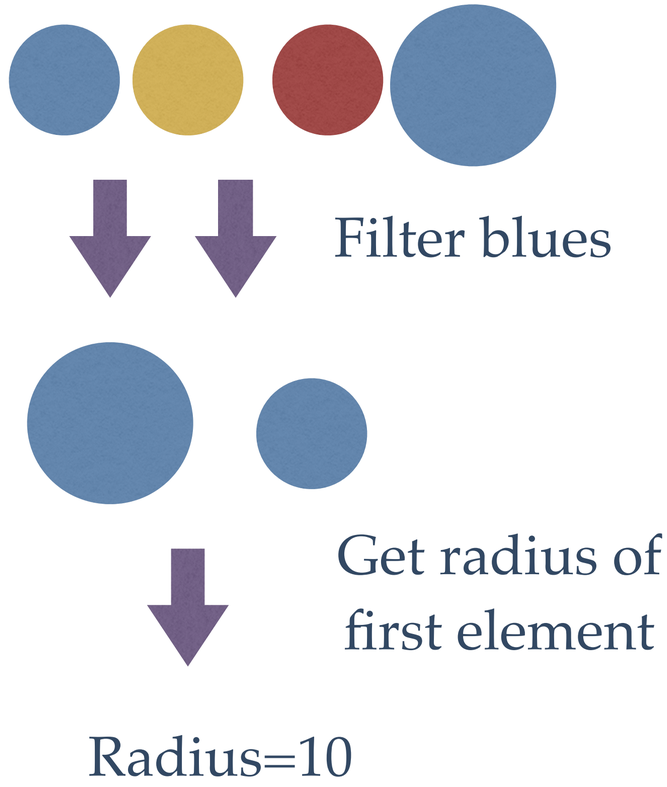
PARALLEL PROCESSING and NON-DETERMINISM

- Encounter order and output order of an item may differ from time to time if we are processing in parallel.
- Stream structure guarantees the latest output's order but not the intermediate results

- First execution



- Second execution



PARALLEL STREAM BENCHMARKING

	Type	Score	Error	Units
Sequential	throughput	**0.090**	± 0.007	ops / sec
Parallel	throughput	**0.025**	± 0.003	ops / sec


```
String fileName = "/Users/ocakcit/Downloads/Crimes_-_2001_to_present.csv";
try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
    return stream.parallel()
        .map(t -> t.split(",")[0]).collect(Collectors.toList());

} catch (IOException e) {
    e.printStackTrace();
}
```

- **Parallel processing is 4 times slower unexpectedly.** Because underlying fork-join framework has an overhead when starting the threads. And the operation on each item costs a negligible time. That's why overhead slows the process down.

PARALLELISM ON CPU-BOUND OPERATIONS

	Score	Error	Units
Sequential	**0.085**	± 0.003	ops/sec
Parallel	**0.258**	± 0.008	ops/sec

```
String fileName = "/Users/ocakcit/Downloads/Crimes_-_2001_to_present.csv";
BufferedReader bufferedReader = new BufferedReader(new FileReader(new File(fileName)));
Stream<String> stream = bufferedReader.lines().parallel();
stream.forEach(t ->
{
    String s = t.split(",")[0];
    Blackhole.consumeCPU(300);
    return;
});
bufferedReader.close();
return stream;
```

Now, parallel processing is faster.

Because the process time on each item is much more than fork-join overhead.

CONCLUSION - WHEN TO GO PARALLEL?

- If our process is not IO-bound.
- If process for each item is CPU-bound and
- If **duration of each process * item count > fork-join overhead**
- **Benchmark your operations before going parallel.**

PARALLELISM ON STATEFUL OPERATIONS

	Score	Error	Units
Sequential	**0.008**	± 0.001	ops/sec
Parallel	**0.008**	± 0.002	ops/sec

```
String fileName = "/Users/ocakcit/Downloads/Crimes_-_2001_to_present.csv";
BufferedReader bufferedReader = new BufferedReader(new FileReader(new File(fileName)));
Stream<String> stream = bufferedReader.lines().parallel();
stream.sorted().forEach(t ->
{
    String s = t.split(",")[0];
    return;
});;
bufferedReader.close();
return stream;
```

Stateful operations such as distinct and sort harms parallelism.

CONCLUSION - WHEN TO GO PARALLEL?

- If most of the operations are stateless such as map and filter. If sort, distinct methods are used, multiple passes or data caching will be required. Bad for parallelism because laziness can not be leveraged.

- When source collection is efficiently splittable such as **ArrayList** or **Array**.