Streams

Lecture 11 (17 May 2022)

Java – Internal vs. External Iteration (I)

- Till Java 7, collections relied on the concept of *external iteration*
  - By implementing `Iterable`, a collection provides a means to step sequentially through its elements. For example

```
List<String> stringList = Arrays.asList("item1", "item2", "item3");

for ( String item : stringList ) {
   System.out.println( item.toUpperCase() );
}
```

or

```
List<String> stringList = Arrays.asList("item1", "item2", "item3");

Iterator<String> stringListIt = stringList.iterator();
while ( stringListIt.hasNext() ) {
   System.out.println(stringListIt.next().toUpperCase());
}
```

Java – Internal vs. External Iteration (II)

- The alternative to external iteration is *internal iteration*
  - the library handles the iteration; the client only provides the code which must be executed for the elements.

```
List<String> stringList = Arrays.asList("item1", "item2", "item3");

stringList.forEach(s -> System.out.println(s.toUpperCase()));
```

Interface Iterable<T>

forEach

```
default void forEach(Consumer<? super T> action)
```

Interface Consumer<T>

accept

```
void accept(T t)
```

- External iteration mixes the “what” (uppercase) and the “how” (for loop/iterator); internal iteration lets the client to provide only the “what”
  - benefits: client code becomes clearer, can be optimized in the library.

Internal Iteration: removing elements

Interface Collection<E>

removeIf

```
default boolean removeIf(Predicate<? super E> filter)
```

Interface Predicate<T>

test

```
boolean test(T t)
```

asList returns an 'unmodifiable' List

- Example: removing even numbers from a list

```
List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7,8,9);
intList.removeIf( el -> el % 2 == 0 );
intList.forEach(System.out :: println);
```

- Running the example:


```
Exception in thread "main" java.lang.UnsupportedOperationException:
	at java.base/java.util.Iterator.remove(Iterator.java:102)
	at java.base/java.util.Collection.removeIf(Collection.java:
	at lecture11.iteration.IterationMain.intIter2(IterationMain.
	at lecture11.iteration.IterationMain.main(IterationMain.java:14)
```

Java – Internal Iteration

- Fixing the example:

```
List<Integer> intList = new ArrayList (Arrays.asList(1,2,3,4,5,6,7,8,9));
intList.removeIf( el -> el % 2 == 0 );
intList.forEach(System.out :: println);
```

- Output:

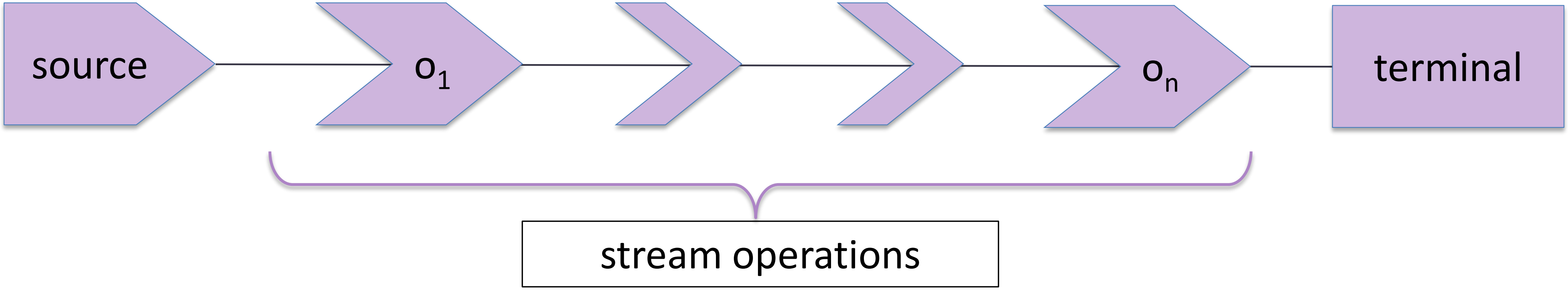
```
run-single:
1
3
5
7
9
BUILD SUCCESSFUL (total time: 0 seconds)
```

Streams

Streams

- What are streams?
  - a stream is a sequence of objects
    like array or list
  - manipulate the stream by using/composing internal iterations
- Why do you want streams?
  - simplifies coding
  - more concise code (compared to looping over lists, arrays, …)
  - improve performance
    - compiler optimizations
    - using multiple cores

Stream pipelines



`Arrays.stream(a)` `filter(n -> n > 2)` `distinct()` `count()`

typically written as

```
Arrays.stream(a)            // source: turn array a into a stream
    .filter(n -> n > 2)     // operation 1: remove some objects
    .distinct()             // operation 2: remove duplicate objects
    .count();               // terminal: count the number of objects
```

Stream representation

`interface Stream<T>`

- *source methods* to create a stream
  - `of`, `generate`, `iterate`
- *intermediate methods* to manipulate and select stream elements
  - `filter`, `map`, `distinct`, `sorted`, `limit`, `skip` …
- *terminal methods* to transform the stream to some final result
  - `count`, `reduce`, `forEach`, `toArray`, `collect` …

check the documentation of `Stream<T>` for the methods and their types!
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Stream elements for our examples

```
public enum Programme { AI, CS }
public enum Result    { Fail, Insufficient, Sufficient, Good }

public class Grade { private int assignment; private Result result; …}

public class Student {
  private String       name;
  private Programme    programme;
  private int          year;
  private List<Grade> grades;

  public Student(String name, Programme p, int year, Grade ... grades) {…}
  // getters
```

Some students

```
Student[] students = {
  new Student("Alice", Programme.AI, 2,
         new Grade(1,Result.Good), new Grade(2,Result.Sufficient), new Grade(3,Result.Good)),
  new Student("Bob", Programme.CS, 1,
         new Grade(1, Result.Insufficient), new Grade(2, Result.Fail)),
  new Student("Carol", Programme.CS, 2),
  new Student("Dave", Programme.AI, 3,
         new Grade(1,Result.Good), new Grade(2,Result.Insufficient), new Grade(3,Result.Good)),
  new Student("Eva", Programme.AI, 2,
         new Grade(1, Result.Good), new Grade(2, Result.Good), new Grade(3, Result.Good)),
  new Student("Fred", Programme.CS, 1,
         new Grade(1,Result.Good), new Grade(2,Result.Insufficient), new Grade(3,Result.Good))
  };
List<Student> studentList = Arrays.asList(students);
```

Sources

Stream sources

- usual way to create a stream: turn array or collection into a stream
- array is not a proper class in Java
  - use utility class `Arrays` instead

```
static <T> Stream<T> stream(T[] array)
static <T> Stream<T> stream(T[] array, int from, int to)
```

```
Arrays.stream(students);
```

- Collection interface contains a method to turn it into a stream

```
Stream<E> stream()
```

```
studentList.stream();
```

- make ad-hoc streams by enumerating elements (or iterating a function; see later)

```
static <T> Stream<T> of (T ... values)
```

```
Stream.of(1, 2, 3, 4);
Stream.iterate(1, x -> x + 1);
```

IntStream, LongStream, DoubleStream

- For the basic types `int`, `long` and `double` there are special streams
  - the stream elements are not boxed !
- generators

```
static IntStream of (int... values)
static IntStream range(int startInclusive, int endExclusive)
```

- e.g.

```
IntStream.of(2, 3, 5, 7)
IntStream.range(0, N)
```

- special methods

```
int sum()
```

note: there is a difference between Stream<Integer> and IntStream filtering & manipulating elements in the stream

Intermediate Operations

Building a pipeline of operations

- Intermediate operations define operations that will be applied to each stream element once evaluation happens
- They return other Streams, which means we can chain them!
- examples:
  - `filter` (apply boolean function and only retain element if True)
  - `map` (apply unary function and pass result)
  - `flatMap` (produce a single stream from separate streams from elements)
  - `skip` (skip a certain number of elements)
  - `distinct` (only produce unique elements)

Filter

- Select elements having some property

```
Stream<T> filter(Predicate<? super T> predicate)

interface Predicate<T> {   boolean test(T t)   }
```

- e.g.

```
long result =
            Arrays.stream(students)
                  .filter((Student s) -> s.getProgramme() == Programme.AI)
                  .count();
System.out.println(result);
```

RUN

3

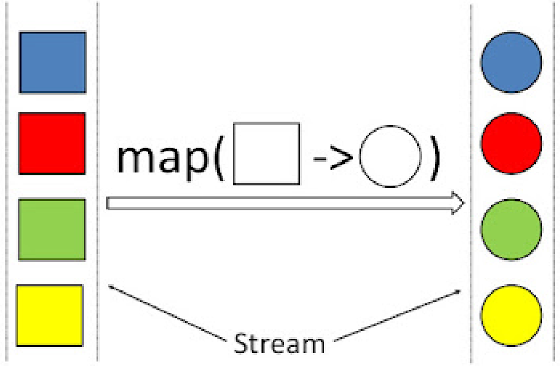
map, mapToInt

- Change the stream elements

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
IntStream mapToInt(ToIntFunction<? super T> mapper)
```

using

```
interface Function<A,R> { R apply(A t) }
interface ToIntFunction<A> { int applyAsInt (A t) }
```



- e.g.

```
int sum = studentList.stream()                        // Stream<Student>
    .filter(s -> s.getProgramme() == Programme.AI)     // Stream<Student>
    .mapToInt(s -> s.getGrades().size())               // IntStream
    .sum();

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

RUN

9

flatMap

omitted <? super T,? extends R>

- Applying a function yielding a stream to the elements of the input stream and then flattens the resulting elements into a new stream

```
<R> Stream<R> map(Function<T, R> mapper)
<R> Stream<R> flatMap(Function<T, Stream<R>> mapper)
```

- e.g. count the number of Fails of all students

```
long nrFails = studentList.stream()                       // Stream<Student>
      .flatMap((Student s) -> s.getGrades().stream())     // Stream<Grade>
      .filter((Grade g) -> g.getResult() == Result.Fail)
      .count();
System.out.println(nrFails);
```

RUN

```
1
```

flatMap + Map

- turn a stream of Students into a stream of Grades (1 to many mapping)
- turn stream of Grades into stream of Results (1 to 1 mapping)
- e.g. count the number of Fails of all students

```
long nrFails = studentList.stream()                             // Stream<Student>
        .flatMap((Student s) -> s.getGrades().stream())  // Stream<Grade>
        .map( Grade :: getResult )                             // Stream<Result>
        .filter(r -> r == Result.Fail)
        .count();
```

shorthand notation for: `(Grade g) -> g.getResult()`

Sidenote: the :: “method reference operator”

- Syntax
  `<Class name>::<method name>`
- Can be used for
  - a static method, e.g. `(Math::abs)`
  - an instance method, e.g. `(Grade::getResult)`
  - a constructor, e.g. `is.mapToObj(Integer::new)`

just a more concise notation

`<U> Stream<U> mapToObj(IntFunction<U> mapper)`

- From lambdas to ::

```
Comparator<Student> c1
      = (Student x, Student y) -> x.getName().compareTo(y.getname());
Comparator<Student> c2 = Comparator.comparing(x -> x.getname());
Comparator<Student> c3 = Comparator.comparing(Student::getName);
```

nested streams

anyMatch: checks if the stream contains at least one element which satisfies the given predicate

- CS students scoring at least one Good

```
studentList
   .stream()                                           // Stream<Student>
   .filter((Student s) -> s.getProgramme() == Programme.CS) // Stream<Student>
   .filter(
      (Student s) -> s.getGrades()                     // List<Grade>
            .stream()                                  // Stream<Grade>
            .anyMatch(g -> g.getResult() == Result.Good) // boolean
         )                                             // Stream<Student>
   .forEach(System.out::println);                      // void
```

shorthand notation for: (s ->System.out.println(s))

RUN

```
Student Fred (CS)
```

Students having only Good

```
studentList
    .stream()
    .filter(s -> s.getGrades()
        .stream()
        .allMatch(g -> g.getResult() == Result.Good))
    .forEach(System.out::println);
```

allMatch: checks all element satisfy the given predicate

!!!

RUN

```
Student Carol (CS)
Student Eva (AI)
```

```
Student [] students = {
  new Student("Carol", Study.CS, 2),
  new Student("Eva", Study.AI, 2,
     new Grade(1, Result.Good),
     new Grade(2, Result.Good),
     new Grade(3, Result.Good)),
  ..
```

evaluating a stream & computing a final result

Terminal Operations

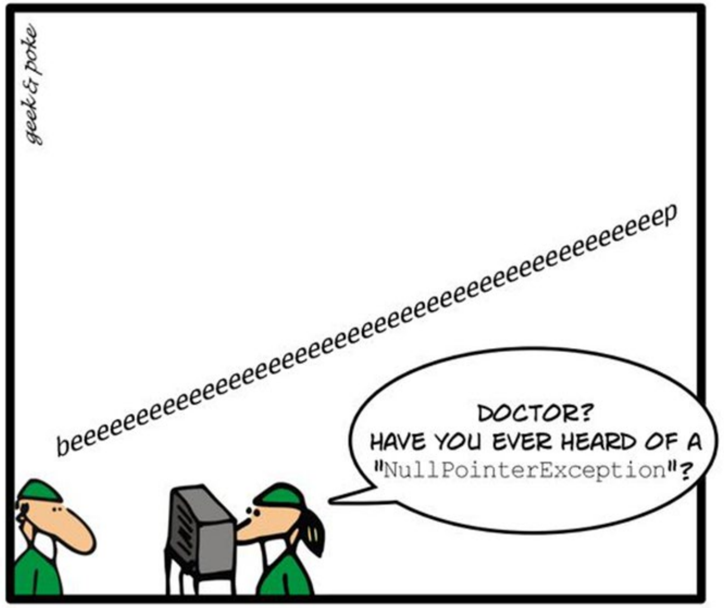
Stream terminals

```
interface Consumer<T> {
  void accept(T t)
}
```

```
long count()                                // number of elements
Optional<T> max ( Comparator<T> )           // maximum element if any
Optional<T> min ( Comparator<T> )
Optional<T> findFirst()                     // first element if any
Optional<T> findAny()                       // some element if any
void forEach(Consumer<T> action)
void forEachOrdered(Consumer<T> action)
boolean anyMatch(Predicate<T> predicate)
boolean allMatch(Predicate<T> predicate)
boolean noneMatch(Predicate<T> predicate)
```

Sitenote: type Optional (I)

- All of us must have encountered `NullPointerException`
- This exception happens when you try to use an object reference which has not been initialized or initialized to `null`.
  - `null` simply means ‘absence of a value’.
- “I call it my billion-dollar mistake.” – Sir C. A. R. Hoare, on his invention of the null reference.
- Optional is a way of replacing null pointers with a non-null values.
  - An `Optional<T>` may either contain a non-null T reference (in which case we say the reference is *present*), or it may contain nothing (in which case we say the reference is *absent*).
- You can view Optional as a single-value container which may or may not contain a value.



RECENTLY IN THE OPERATING ROOM

Sitenote: type Optional (II)

```
public class Optional<T>{
  static <T> Optional<T> empty();
  static <T> Optional<T> of(T value);
  void ifPresent(Consumer<T> action);
  boolean isEmpty();
  boolean isPresent();
  Stream<T> stream();
}
```

And many other methods

Sitenote: type Optional (III)

```
Optional<Integer> optional1 = Optional.of(42);
optional1.isPresent();    // returns true
optional1.get();          // returns 42

Optional<Integer> optional2 = Optional.empty();
optional1.isPresent();    // returns false
```

- So

    `Optional<T>` **`max`** `( Comparator<T> )`

  returns an `Optional<T>` containing the maximum element of this stream, or an empty optional if this stream is empty.

First student without grades

```
Optional<Student> optFirst = studentList
    .stream()
    .filter(s -> s.getGrades().isEmpty())
    .findFirst();
System.out.println(optFirst);
```

RUN

```
Optional[Student Carol (CS)]
```

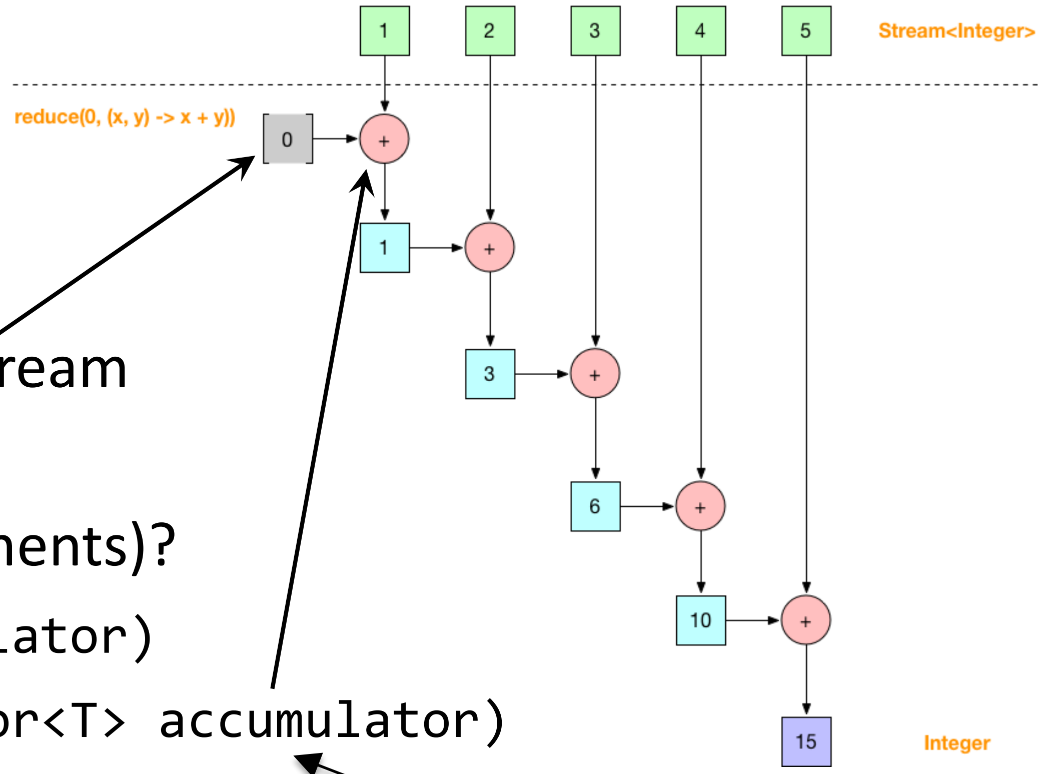
Or (slightly) better

```
Optional<Student> optFirst = studentList
    .stream()
    .filter(s -> s.getGrades().isEmpty())
    .findFirst();
optFirst.ifPresent(System.out::println);
```

RUN

```
Student Carol (CS)
```

Stream reduction



- Predefined methods for simple operations:
  - we can count the number of elements in a stream
  - for `IntStream` we can sum the elements
- What about other operations (like product of elements)?

```
Optional<T> reduce(BinaryOperator<T> accumulator)
          T reduce(T identity, BinaryOperator<T> accumulator)
```

initial value (of the of the intermediate result)

combines intermediate result with next element

- Example: computing 5!

```
int fac5 = IntStream
    .rangeClosed(1, 5)
    .reduce(1, (n,m) -> n * m));
```

Collect: mutable stream reduction

- Suppose we want to concatenate all elements from Stream<String> s
  - `String concatenated = s.reduce("", String::concat);`
- Horrible performance: $O(n^2)$ in the number of characters
- Idea: “reduce into a StringBuilder”
- collect: collects together the desired results into a mutable result container

Collect: two variants

R: the container type, e.g. List

- There are two variants of collect
  1. ```
     <R>    R collect( Supplier<R> supplier,
                       BiConsumer<R, T> accumulator,
                       BiConsumer<R, R> combiner )
     ```
  2. ```
     <R, A> R collect( Collector<T, A, R> collector )
     ```

Collect (variant 1)

```
interface Supplier<R> {
  R get()
}
```

```
<R> R collect( Supplier<R> supplier,
               BiConsumer<R,T> accumulator,
               BiConsumer<R,R> combiner )
```

```
interface BiConsumer<T,U> {
  void accept(T t, U u)
}
```

This variant requires three argument functions:

- **supplier**: construct instances of the result container
- **accumulator**: put input element into a result container
- **combiner**: merge one result container with another (if you have parallel streams; see later)

```
ArrayList<String> slist = s.collect( // Stream<Object> s
    ()       -> new ArrayList<>(),   // empty list for chunk
    (c, e)   -> c.add(e.toString()), // add element to list
    (c1, c2) -> c1.addAll(c2));      // combine lists
```

Making the map explicit

- Now we do it in two steps: first the conversion to string and then the collection

```
List<String> slist = s                              // Stream<Object>
                     .map(Object::toString)         // Stream<String>
                     .collect(ArrayList::new,
                              ArrayList::add,
                              ArrayList::addAll);
```

Collect (variant 2)

- This variant requires just one argument of type:

```
interface Collector<T,A,R>
```

  - T - the type of input elements to the reduction operation
  - A - the mutable accumulation type of the reduction operation
  - R - the result type of the reduction operation

- Often used with standard collectors from the **Collectors** class:

```
class Collectors {
  static <T> Collector<T,?,List<T>> toList();
  static <T> Collector<T,?,Set<T>>  toSet();
```

…

List collector

```
List<Student> aiStudents =
  Arrays
    .stream(students)
    .filter(s -> s.getProgramme() == Programme.AI)
    .collect(Collectors.toList());
System.out.println("AI students " + aiStudents);
```

we have no control of the type of list:
`ArrayList, LinkedList..`

RUN

```
AI students [Student Alice (AI), Student Dave (AI), Student Eva (AI)]
```

Linked List collector

```
List<Student> aiStudents =
  Arrays
    .stream(students)
    .filter(s -> s.getProgramme() == Programme.AI)
    .collect(Collectors.toCollection( LinkedList::new ));
System.out.println("AI students " + aiStudents);
```

Here we specify a `LinkedList` collector supplier

RUN

```
AI students [Student Alice (AI), Student Dave (AI), Student Eva (AI)]
```

Map collector

map names to grade-lists

get key

get value

```
Map<String, List<Grade>> map =
   studentList
   .stream() // Stream<Student>
   .collect(Collectors.toMap(Student::getName, Student::getGrades));
```

```
static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<T,K> keyM, Function<T,U> valM)
```

More stream manipulations

Sort students bij their average grade

```
Stream<Pair<Student,Double>> studentsWithAverageGrades =
        studentList.stream()
                .map((Student s) -> new Pair<>(s,
                        s.getGrades()
                            .stream()
                            .mapToInt((Grade g) -> g.result().ordinal())
                            .average().orElse(0)));
studentsWithAverageGrades
        .sorted(Comparator.comparing(Pair::second, Comparator.reverseOrder()))
        .forEach(System.out::println);
```

ordinal: position in enum type

orElse: Optional method

RUN

```
Pair[first=Student Eva (AI), second=3.0]
Pair[first=Student Alice (AI), second=2.6666666666666665]
Pair[first=Student Dave (AI), second=2.3333333333333335]
Pair[first=Student Fred (CS), second=2.3333333333333335]
Pair[first=Student Bob (CS), second=0.5]
Pair[first=Student Carol (CS), second=0.0]
```

Infinite streams: iterate

- Due to lazy evaluation (streams are only evaluated if absolutely necessary) streams can be infinite
- Two ways to make infinite streams: `generate` and `iterate`

```
int number = 1234567;
List<Integer> list =
    IntStream
        .iterate(number, n -> n / 10)
        .takeWhile(n -> n > 0)
        .map(n -> n % 10)
        .boxed()
        .collect(Collectors.toList());
System.out.println(list);.
```

`iterate(N,f)` produces: N, f(N), f(f(N)), f(f(f(N))),...

1234567, 123456, 12345, 1234, 123, 12, 1, 0, 0 ,0,...

1234567, 123456, 12345, 1234, 123, 12, 1

7, 6, 5, 4, 3, 2, 1

infinite streams: generate

- Very similar to iterate, but no value passed between calls

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

repeat calling this method forever

`limit(N)`: returns first N elements

- Typically we use a method of a stateful object

RUN

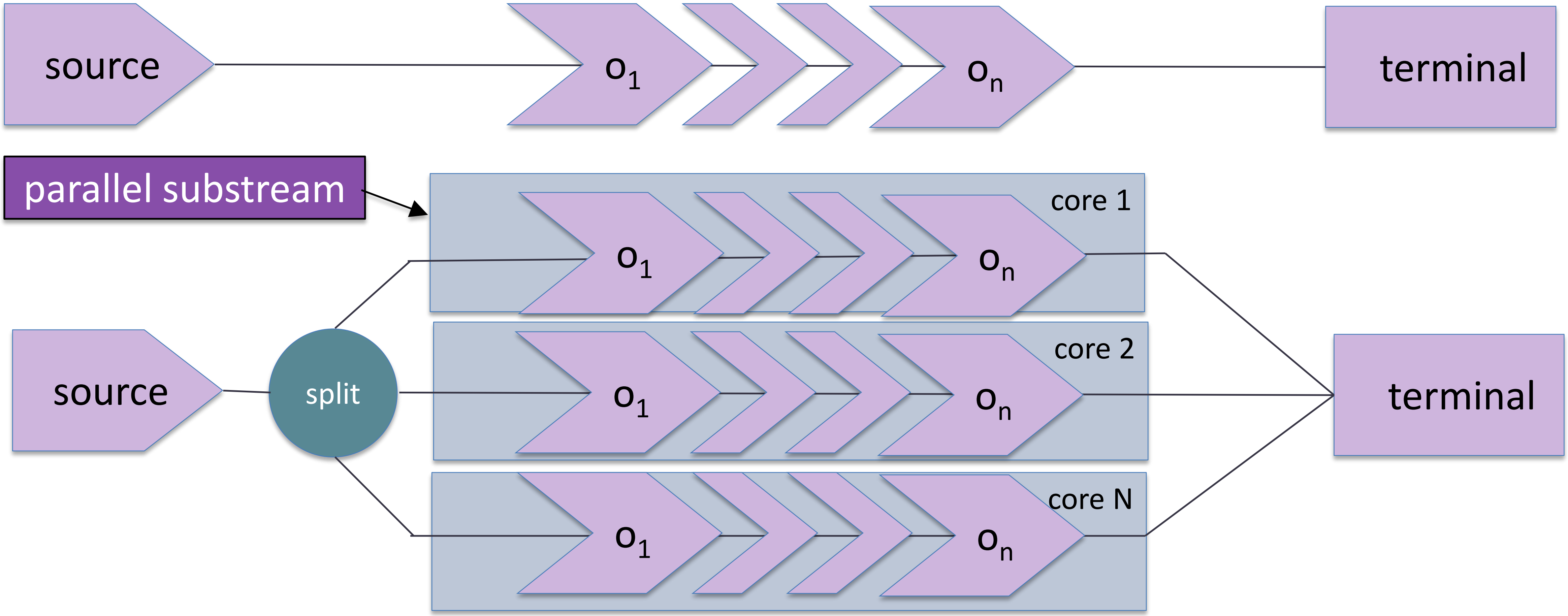
```
0.37238726115093057
0.4527012376585603
0.00414289561216839
0.13991206174351822
0.07948602794734327
```

using all cores in your machine

Parallel streams

multicore machines are everywhere

- Most modern computers have multiple cores
  - each core can execute its own (part of your) program
  - it requires hard work of the programmer to use this efficiently **and safely**



Parallel streams

- A stream can be turned into a (potentially) parallel stream by `parallel()`
  - there is no guarantee that this becomes really parallel
- Instead of using `stream()` we can use `parallelStream()`
  - there is no guarantee that this becomes really parallel
- Merging in a terminal is implicit
  - this explains the structure of reduce with combiners:

```
int graded = studentList
  .parallelStream()
  .reduce(0,
        (count, student) -> count + student.getGrades().size(),
         Integer::sum);
```

a parallel stream is created

to combine sub-streams

Parallel

This doesn't do any real work but tricks the compiler into believing the program has to do the computations

```java
private static boolean useless(int n) {
  for (int i = 0; i < n; i += 1) {
    for (int j = i; j < n; j += 1) {
      if (i + j < 0) {
        return false;
      }
    }
  }
  return true;
}
```

```java
public static void run() {
  int N = 50_000;
  long startTime = System.currentTimeMillis();
  long seqN = IntStream
          .range(0, N)
          .filter(i -> useless(i))
          .count();
  long doneTime = System.currentTimeMillis();
  System.out.printf("result: %d, sequential time: %d\n", seqN, doneTime - startTime);
  startTime = System.currentTimeMillis();
  long parN = IntStream
          .range(0, N)
          .parallel()
          .filter(i -> useless(i))
          .count();
  doneTime = System.currentTimeMillis();
  System.out.printf("result: %d, parallel time: %d\n", parN, doneTime - startTime);
}
```

split stream

RUN

```
result: 50000, sequential time: 3477
result: 50000, parallel time: 981
```

Recap

- Streams yield concise and efficient programs
  - although everything can be done with arrays, lists and loops,
    this yields longer and more error prone programs that are often slower

- Can be parallelized very easily
  - no guarantees for speed improvements
  - we will see more options for parallelization in the remainder of the course

Lecture 12: Concurrency