Streams

Object-Oriented Programming Lecture 11

IJP (Liang): chapter 30

https://docs.oracle.com/javase/tutorial/collections/streams/index.html

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

May 11, 2021

Radboud University

Java – Internal vs. External Iteration

- Till Java 7, the 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 letter : stringList ) {
   System.out.println(letter.toUpperCase());
}
```

Or

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

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

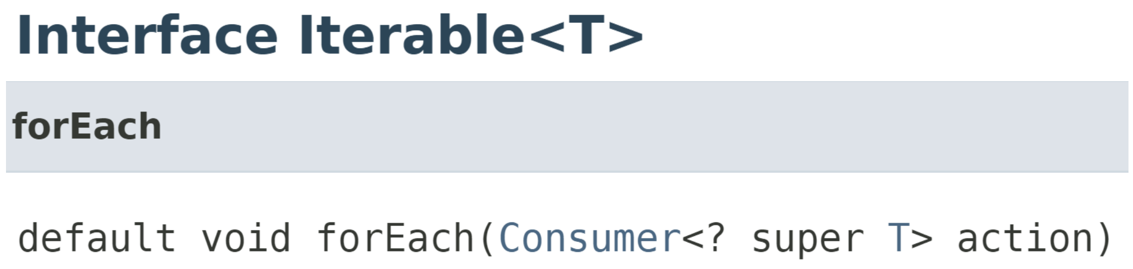
Java – Internal vs. External Iteration

- The alternative to external iteration is *internal iteration*
  - the libraray 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()));
```

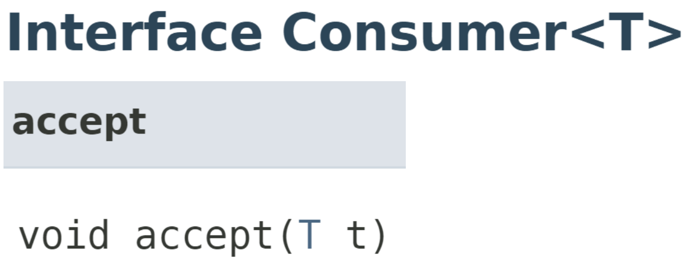
- External iteration mixes the "what" (uppercase) and the "how" (for loop/iterator); internal iteration lets the client to provide only the "what"
  - benefits: client code becomes clearer, can be optimized in the library.



Interface Iterable<T>

forEach

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



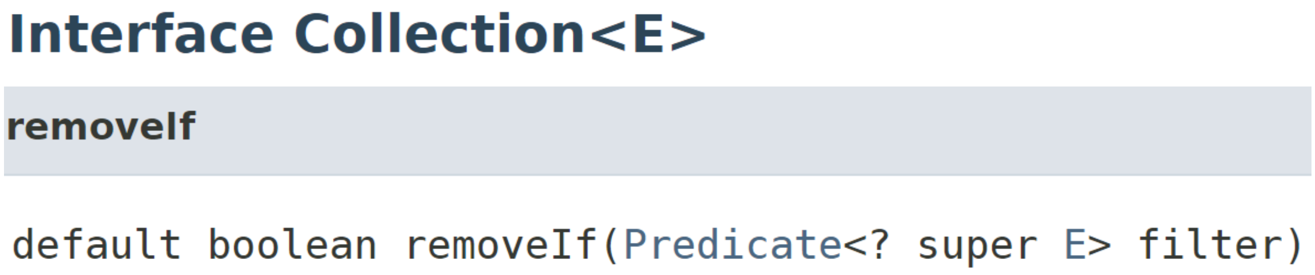
Interface Consumer<T>

accept

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

Java – Internal Iteration

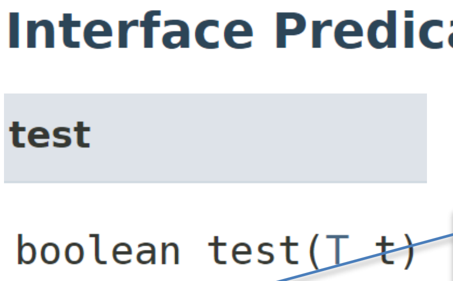
- Removing elements from a collection with *internal iteration*



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 form 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:

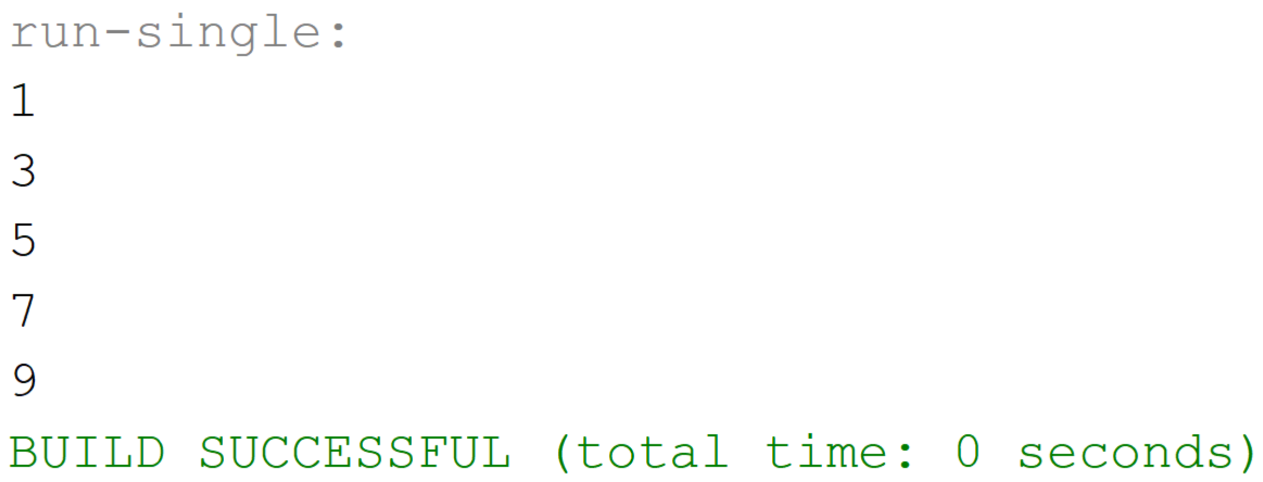
```
Exception in thread "main" java.lang.UnsupportedOperationException: remove
	at java.base/java.util.Iterator.remove(Iterator.java:102)
	at java.base/java.util.Collection.removeIf(Collection.java:577)
	at lecture11.iteration.IterationMain.intIter2(IterationMain.java:43)
	at lecture11.iteration.IterationMain.main(IterationMain.java:14)
```

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

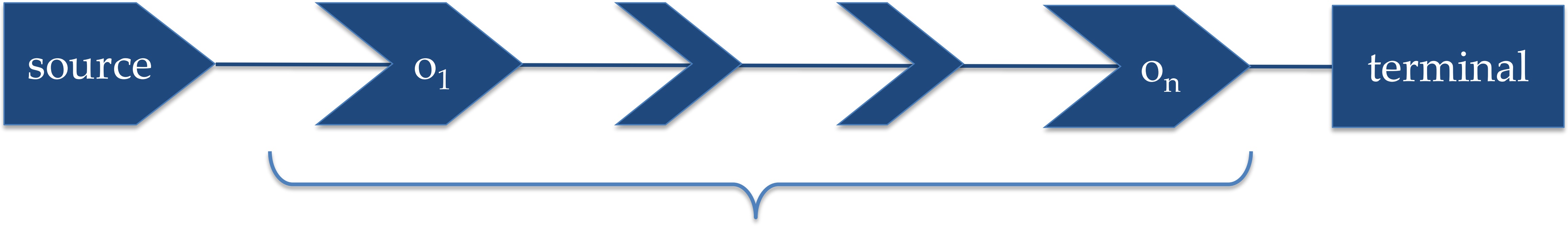
what are streams

- 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



stream operations

`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 {
  protected String name;
  protected Programme programme;
  protected int year;
  protected 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 make a stream: turn array or collection into a stream

array is not a proper class in Java
use utility `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

```
default 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 **don’t** cause stream evaluation themselves

they `return` **other `Streams`**, which means we can **chain** them!

examples:

- `filter` (apply boolean function and only pass element if `True`)
- `map` (apply unary function and pass mutated 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)
```

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

e.g.

```
    assertEquals("number of AI students", 3,
      Arrays
        .stream(students)
        .filter(s -> s.getProgramme() == Programme.AI)
        .count());
```

map, mapToInt ..

change the stream elements

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

using

```
public interface Function<T,R> { R apply(T t) }
```

e.g.

```
assertEquals("number of AI grades", 9,
  studentList
    .stream()                                         // Stream<Student>
    .filter(s -> s.getProgramme() == Programme.AI)     // Stream<Student>
    .mapToInt(s -> s.getGrades().size())               // IntStream
    .sum());
```

flatMap

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

```
    assertEquals("fails", 1,
        studentList
         .stream()                           // Stream<Student>
         .flatMap(s -> s.getGrades().stream()) // Stream<Grade>
         .filter(g -> g.getResult() == Result.Fail)
         .count()
      );
```

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

```
assertEquals("fails", 1,
  studentList
   .stream()                                    // Stream<Student>
   .flatMap(s -> s.getGrades().stream())         // Stream<Grade>
   .map(Grade::getResult)                        // Stream<Result>
   .filter(r -> r == Result.Fail)
   .count()
);
```

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

sidenote: the :: "method reference operator"

syntax

`<Class name>::<method name>`

just a more concise notation

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)`

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 filters

CS students scoring at least one `Good`

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

```
Student Fred (CS)
```

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

students having only **Good**

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

```
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

```
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<? super T> action)
void forEachOrdered(Consumer<? super T> action)
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

`collect & reduce` (types given later)

first student without grades

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

```
studentList
   .stream()
   .filter(s -> s.getGrades().isEmpty())
   .findFirst()
   .ifPresent(System.out::println);
```

doesn’t print unless there is such a student

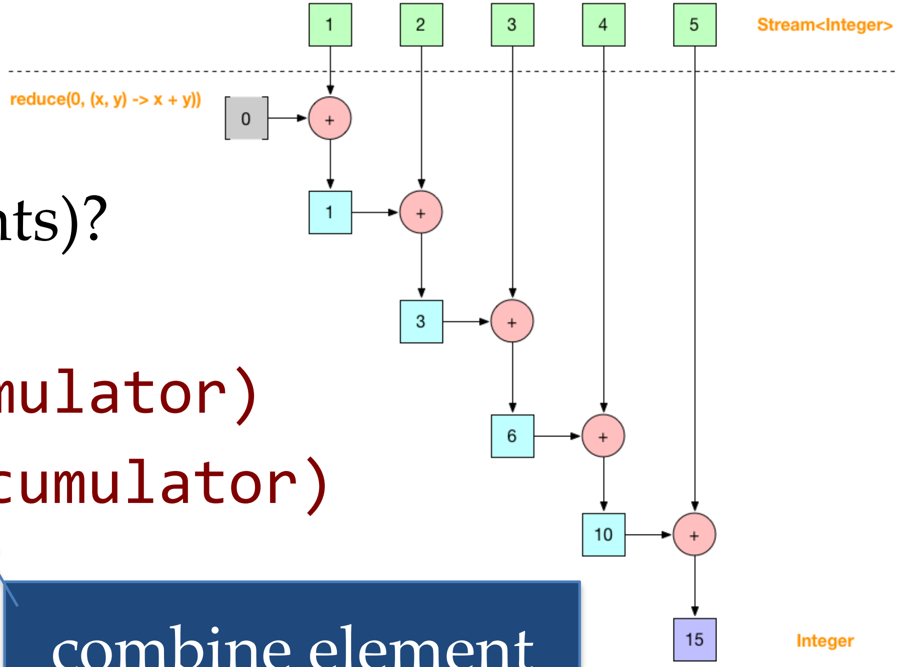
```
Student Carol (CS)
```

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

combine element with reduced rest

for `IntStream`:
`sum() ≡ reduce(0, (x,y)->x+y)`

e.g.

```
 assertEquals("fac 4 = 24", 24,
   IntStream
   .rangeClosed(1, 4)
   .reduce(1, (n,m) -> n * m));
```

map with type conversion & reduce in one

```
<U> U reduce(U identity,                              // initial
             BiFunction<U,? super T,U> accumulator, // (U u,T t) -> U
             BinaryOperator<U> combiner)            // (U x,U y) -> U
```

combine reduced parts of the stream

reduce parts of the stream

e.g.

```
int graded = studentList
  .stream()
  .reduce(0,
          (count, student) -> count + student.getGrades().size(),
          Integer::sum);
assertEquals("graded", 14, graded);
```

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

number of grades could also be computed with flatMap and count

many reductions of this form can be represented more simply by an explicit combination of map() and reduce() operations!

map with type conversion & reduce in one

many reductions of this form can be represented more simply by an explicit combination of map() and reduce() operations!

e.g.

```
int graded = studentList
   .stream()
   .map(student -> student.getGrades().size())
   .reduce(0, Integer::sum);
assertEquals("graded", 14, graded);
```

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

There are two variants of collect

1.
```
<R> R collect(
              Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner)
```
2.
```
<R, A> R collect( Collector<? super T, A, R> collector )
```

collect (variant 1)

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

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator, // R.add(T)
              BiConsumer<R, R> combiner);           // combine Rs
```

```
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

```
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
```

```
List<String> slist = s                          // Stream<Object>
                      .map(Object::toString)    // Stream<String>
                      .collect(ArrayList::new,
                               ArrayList::add,  // no conversion
                               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..

```
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**

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

map collector

map names to grade-lists

make key

make value

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

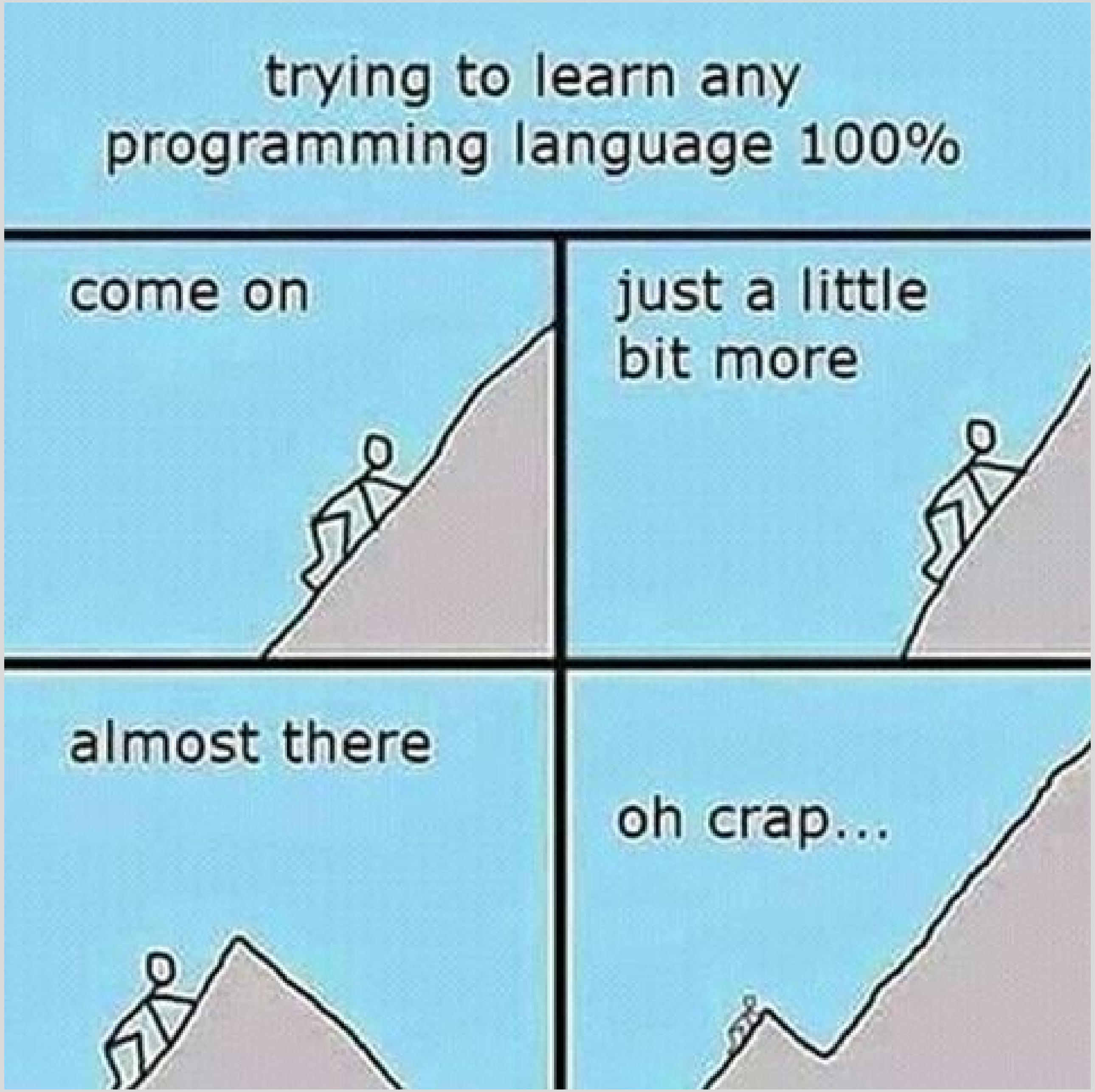
```
{ Eva=[Grade 1, Good, Grade 2, Good, Grade 3, Good]
, Bob=[Grade 1, Insufficient, Grade 2, Fail]
, Alice=[Grade 1, Good, Grade 2, Sufficient, Grade 3, Good]
, Fred=[Grade 1, Good, Grade 2, Insufficient, Grade 3, Good]
, Carol=[]
, Dave=[Grade 1, Good, Grade 2, Insufficient, Grade 3, Good]
}
```

making groups

```
Map<Programme, List<Student>> groups =
  studentList
    .stream()
    .collect(Collectors.groupingBy(Student::getProgramme));

groups
    .forEach((prog, list) -> System.out.format("group %s: %s\n"
       ,prog
       ,list
         .stream()
         .map(Student::getName)
         .collect(Collectors.toList())));
```

```
group CS: [Bob, Carol, Fred]
group AI: [Alice, Dave, Eva]
```


trying to learn any programming language 100%
come on
just a little bit more
almost there
oh crap...

MORE STREAM MANIPULATIONS

sorting

```
Optional<Student> bestStudent =
   studentList
      .stream()
      .sorted((x,y) -> points(y) - points(x))
      .findFirst();
```

Sorts low-to-high;
reverse order for best first

```
Student Eva (AI)
```

using

```
private int points (Student s) {
  return s.getGrades()
    .stream()
    .map(g -> g.getResult().ordinal())
    .reduce(0, (n, m) -> n + m);
}
```

or

```
return s.getGrades()
  .stream()
  .mapToInt(g->g.getResult().ordinal())
  .sum();
```

number in enum type

concat: glue two stream together

```
List<String> problems =
   Stream
    .concat(
       studentList
        .stream()
        .filter(s -> s.getGrades().size() < N)
    , studentList
        .stream()
        .filter(s -> s.getGrades()
           .stream()
           .anyMatch(g -> g.getResult() == Result.Fail))
    )
    .distinct()
    .map(Student::getName)
    .collect(Collectors.toList());
```

students have a problem if:

- number of results is < N (3)
- one or more `Fails`

Bob matches both conditions

```
[Bob, Carol]
```

creating new list of perfectly scoring students

```
List<Student> perfectStudents =
  Stream
    .of("Liye", "Ike")
    .map(n -> new Student(n, Programme.CS, 10,
      IntStream
        .rangeClosed(1, N)
        .boxed()
        .map(i -> new Grade(i, Result.Good))
        .collect(Collectors.toList()))))
    .collect(Collectors.toList());
```

map for `IntStream` yields an `IntStream`

- `boxed()` yields `Stream<Integer>`
- or use `mapToObj(i->new Grade(i,Result.Good))`

list of N Good grades

```
[Student Liye (CS), Student Ike (CS)]
```

order of operations

the order of operations in a stream can change the efficiency

- the result is usually independent of the order of operations
- rule of thumb: filter as soon as possible

```
assertEquals("sort first", 3
  studentList
  .stream()
  .sorted()
  .filter(s -> s.getYear() == 1)
  .count());
```

```
assertEquals("sort last", 3,
  studentList
  .stream()
  .filter(s -> s.getYear() == 1)
  .sorted()
  .count());
```

sorts also the year 2 & 3 students that will be removed in the next step

sorts only year 1 students

infinite streams: iterate

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);
```

`n, f(n), f(f(n)), f(f(f(n)..`

repeat this method forever with `number` as start value

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

```
[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 this method forever

```
0.37238726115093057
0.4527012376585603
0.00414289561216839
0.13991206174351822
0.07948602794734327
```

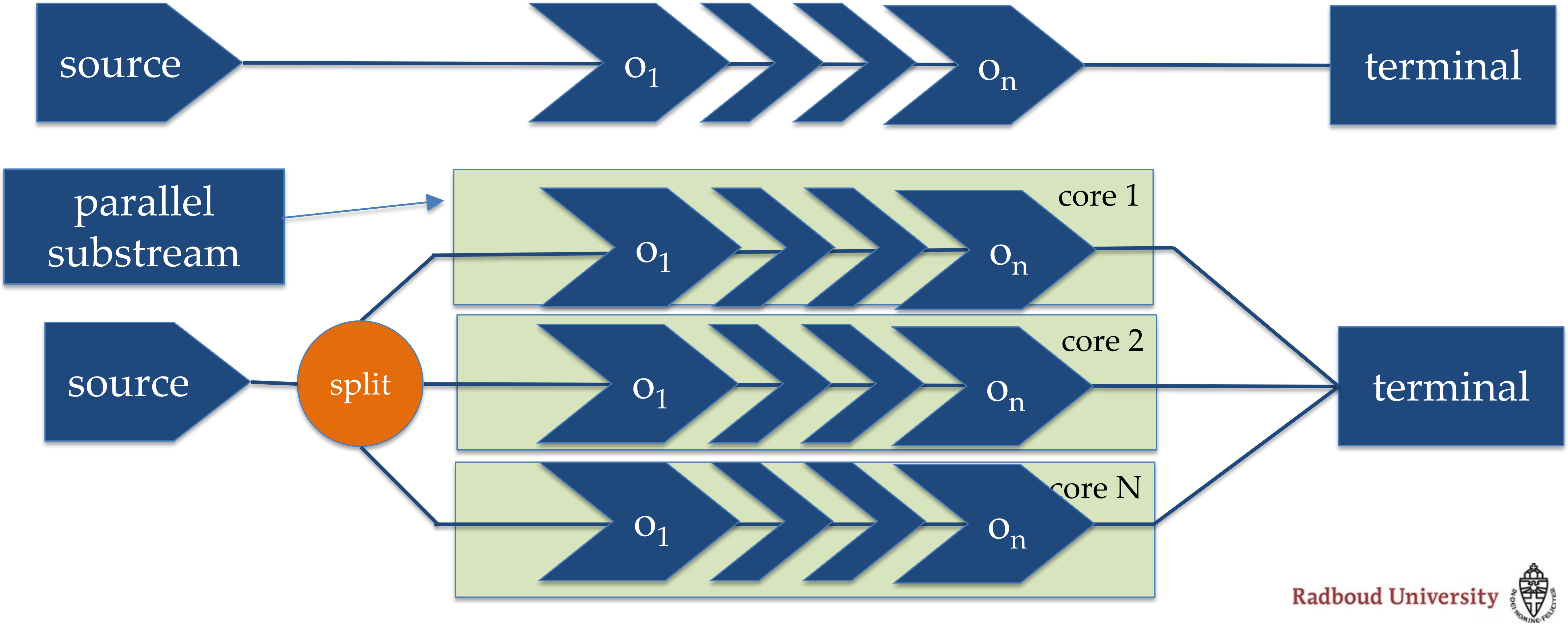
typically we use a method of a stateful object using all cores in your machine

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
  .stream()
  .reduce(0,
        (count, student) -> count + student.getGrades().size(),
        Integer::sum);
```

to merge sub-streams

parallel

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

```
int N = 50_000;
long startTime = System.currentTimeMillis();
assertEquals("sequential", N,
  IntStream
    .range(0, N)
    .filter(i -> useless(i))
    .count());
long doneTime = System.currentTimeMillis();
System.out.println("sequential time " + (doneTime - startTime));
startTime = System.currentTimeMillis();
assertEquals("parallel", N,
  IntStream
    .range(0, N)
    .parallel()
    .filter(i -> useless(i))
    .count());
doneTime = System.currentTimeMillis();
System.out.println("parallel time " + (doneTime – startTime));
```

```
private 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;
}
```

```
sequential time 5254
parallel time 3081
```

split stream

using 4 cores

parallelStream

in principle we can replace any `stream()` by `parallelStream()`

```
studentList
  .parallelStream()
  .sorted((x,y) -> points(y) - points(x))
  .findFirst();
```

sometimes it gives more speed

collect respects the encounter order of stream elements

```
List<Integer> list = Arrays.asList(0,1,2,3,4,5,6,7,8,9);
List<Integer> r1 =
    list
    .stream()
    .collect(Collectors.toList());
List<Integer> r2 =
    list
    .parallelStream()
    .collect(Collectors.toList());
assertEquals("parallelTest", r1, r2);
```

successful test

some streams do not *have* a defined encounter order!

- streams based on e.g. HashSets
- streams on which `unordered()` has been called

good for parallelism!

bad for stable ordering.

`foreach` does *not* respect the encounter order!

```
List<Integer> list = Arrays.asList(0,1,2,3,4,5,6,7,8,9);
System.out.print("stream          ");
list
  .stream()
  .forEach(i -> System.out.print(i + " "));

System.out.print("\nparallelStream ");


list
  .parallelStream()
  .forEach(i -> System.out.print(i + " "));
```

```
stream         0 1 2 3 4 5 6 7 8 9
parallelStream 7 2 8 9 4 1 3 0 5 6
```

order can be different in each run

findFirst() vs findAny()

`Optional<T> findAny()`

- returns an `Optional` describing **some element** of the stream, or an `empty Optional` if the stream is empty
- especially in a parallelStream this is not necessarily the first stream element

`Optional<T> findFirst()`

- returns an `Optional` describing **the first element** of the stream, or an `empty Optional` if the stream is empty

often fails

```
assertEquals("findAny s", 1,
    IntStream
      .range(1, 1000)
      .findAny()
      .getAsInt());
```

typically succeeds, no guarantees

```
assertEquals("findAny p", 1,
    IntStream
      .range(1, 1000)
      .parallel()
      .findAny()
      .getAsInt());
```

going wild ?

this is an easy way to use multiple cores
should we parallelize any stream ?

NO

parallelization of streams and merging is also work
this takes time and slows your program

- sometimes there is a speed gain by using multiple cores

when you notice your stream processing program is slow

- try to optimize the order of operations
- parallelize the top-level of your streams
  measure if this gives the desired effect

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
- no `null`: avoid the billion dollar mistake

lazy evaluation is great

- automatically only compute what you really need

can be parallelized very easily

- no guarantees for speed improvements
- we will see more options for parallelization in the remainder of the course

read https://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html

NEXT WEEK

Concurrency (I)