Specialized Streams

IntStream	for `int` elements
DoubleStream	for `double` elements
LongStream	for `long` elements

It has better performance to use these specialized streams when using numeric data types, because there is no boxing/unboxing

Suppress elements

limit	`.limit(5)` will limit the result to the first 5 elements
skip	`.skip(5)` will skip the first 5 elements
filter	`.filter(e -> e.getS alary() > 200000)` will keep the elements that satisfy the given predicate. In this case, all elements that have salary above 200000

Comparing elements

distinct	`.disti nct()` will compare the elements in the stream using equals() and eliminate duplicates
sorted	`.sorte d((e1, e2) -> e1.get Nam e().co mpa reT o(e 2.g et N ame()))` will sort the elements with the given comparator. Elements must be Comparable.
min	Similar to sorted, but it will find the min element according to the given comparator
max	Similar to sorted, but it will find the max element according to the given comparator

Apply a function to each element

map	`.map(e mpl oye eRe pos ito ry: :fi nd ById)` will apply the given function and substitute the elements in the stream for new elements. In this case, it received a stream of employee IDs and returned a stream of Employee objects
mapToDouble mapToInt mapToLong	similar to *map*, but the function converts the element to the specified primitive type, resulting in a specialized stream IntStream, DoubleStream or LongStream

Apply a function to each element (cont)

flatMap	similar to map, but the number of elements resulting may be different. It's normally used to convert a List of List into a single list with all the elements
peek	`.peek(e -> e.sala ryI ncr eme nt( 10.0))` will apply the give function to all elements in the list, but doesn't substitute the elements in the list

Reduce elements to single value

reduce	`.reduc e(0.0, Double ::sum)` will return a single value. It starts with the identity value (0.0) and applies the given function to each element in the array. In this case it's summing all elements, one by one
allMatch	`.allMa tch(i -> i % 2 == 0);` will check if all elements match the given condition. If so, returns true, else returns false
anyMatch	`.anyMa tch(i -> i % 2 == 0);` will check if one of the elements match the given condition. If so, returns true, else returns false
noneMatch	`.noneM atch(i -> i % 2 == 0);` will check if no elements match the given condition. If so, returns true, else returns false
findFirst	`.findF irst()` will return an Optional with the first element in the stream
forEach	`forEach(e -> e.sala ryI ncr eme nt( 10.0 ))` will apply the given function to each element in the stream, but it's a terminal operation and returns void
count	`.count()` outputs the number of elements in the stream

Collect elements

toList	`collec t(C oll ect ors.to List())` gather all elements in the stream into a List
toSet	`collec t(C oll ect ors.to Set())` gather all elements in the stream into a Set
toCollection	`collec t(C oll ect ors.to Col lec tio n(V ect or: :new))` gather all elements in the list in an arbitrary Collection
joining	`collec t(C oll ect ors.jo ini ng( ", " )).t oS tring()` will join String elements with the given separator and return the aggregated String
summarizingDouble	
summaryStatistics	
partitioningBy	`.colle ct( Col lec tor s.p art iti oni ngBy(s -> s.getG rade() >= PASS_T - HRE SHOLD))` will partition the data into 2 categories based on the given condition. The result will be a Map<Boolean, List<Student>>
groupingBy	`.colle ct( Col lec tor s.g rou pin gBy (Em plo yee ::g etD epa rtm ent));` will group the elements into categories based on the function. The result will be a Map<Department, List<Employee>>
mapping	`mappin g(P ers on: :ge tLa stName, toSet())` it receives a function to be applied to all elements and way of collecting downstream the elements. In this case, it will get the last name of all persons and add them to a set
reducing	