Functional Java

# Functions as Values: first-class functions

store in variable; pass as parameters; return as values;

# Lambda Expression:

Can see nearby variables, but cannot change them!

Data In, Data out:

The lambda expression works much like data processor. It can not do the following:

Access global

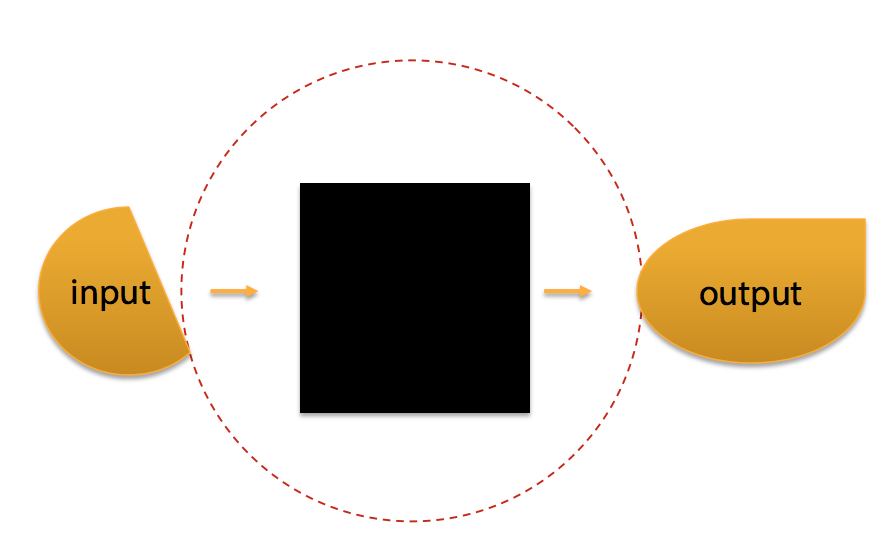
State

Modify

Input

Change the world

It should have no side effects.



External Iteration: Iterable, etc.

Internal Iteration:

Stream

Filter

Map

Reduce

Streams: Coding in a Context

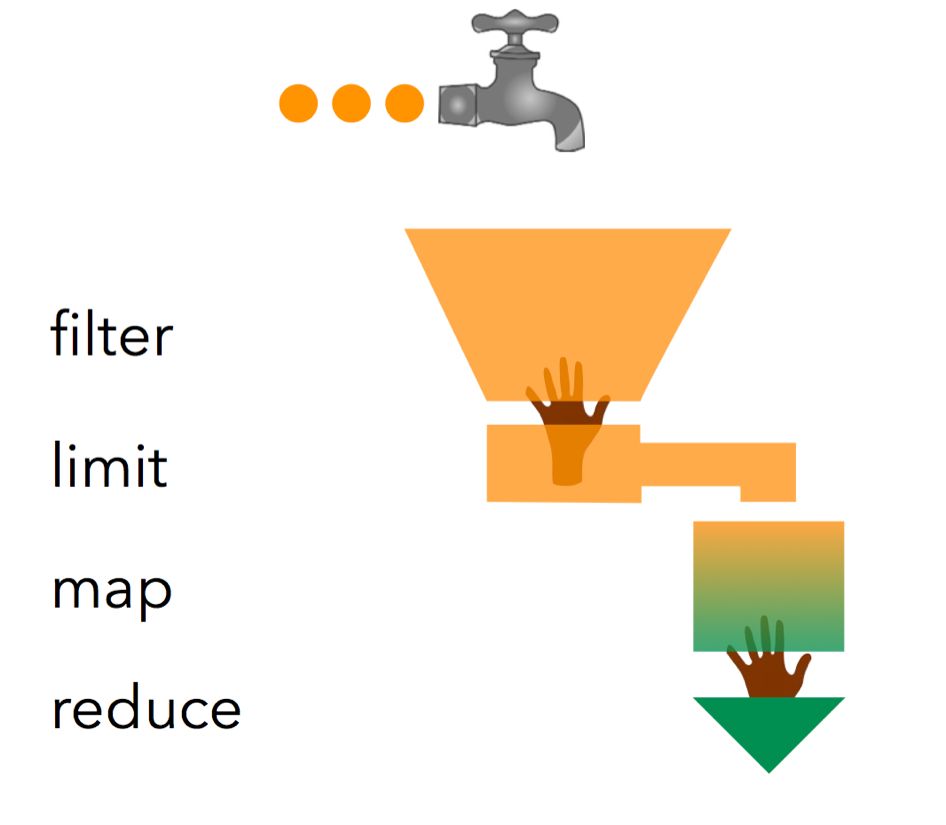
Lazy Streams

Parallel Stream

Stream.map().filter()

How can Streams be more efficient than loops?

As from the chart below, we do the operations for only one iteration. For each element in the stream, we just operate them through filter, limit, map, and then the reduce which is a terminal operation.



Terminal Operators

Only when we get to the terminal operators, trigger processing

And return a value;

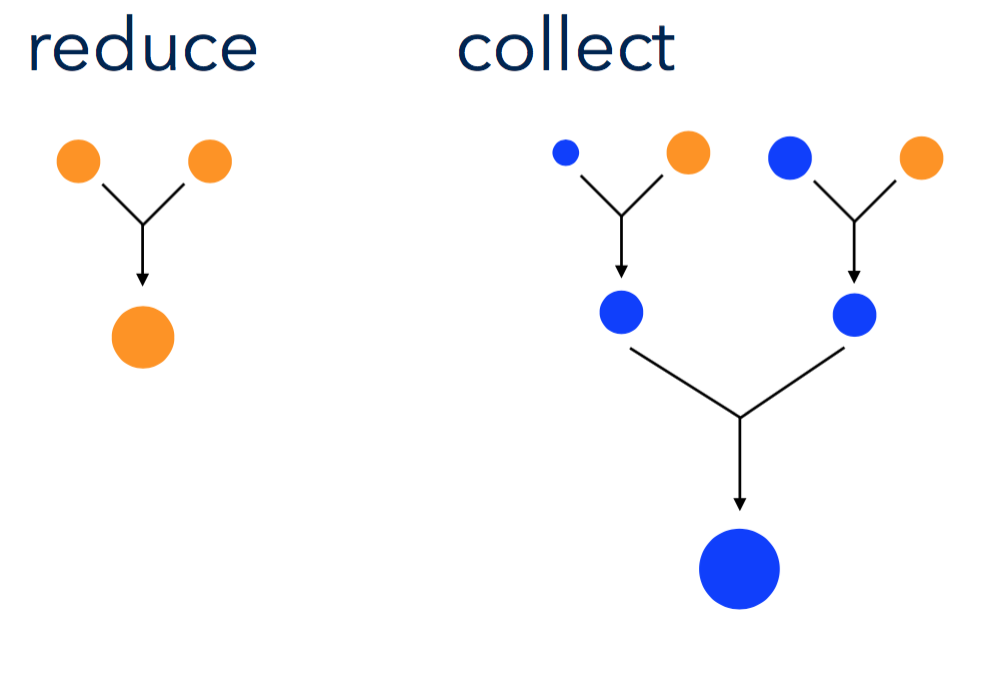
It also terminate the stream instance

Intermediate Operators

Trigger no processing

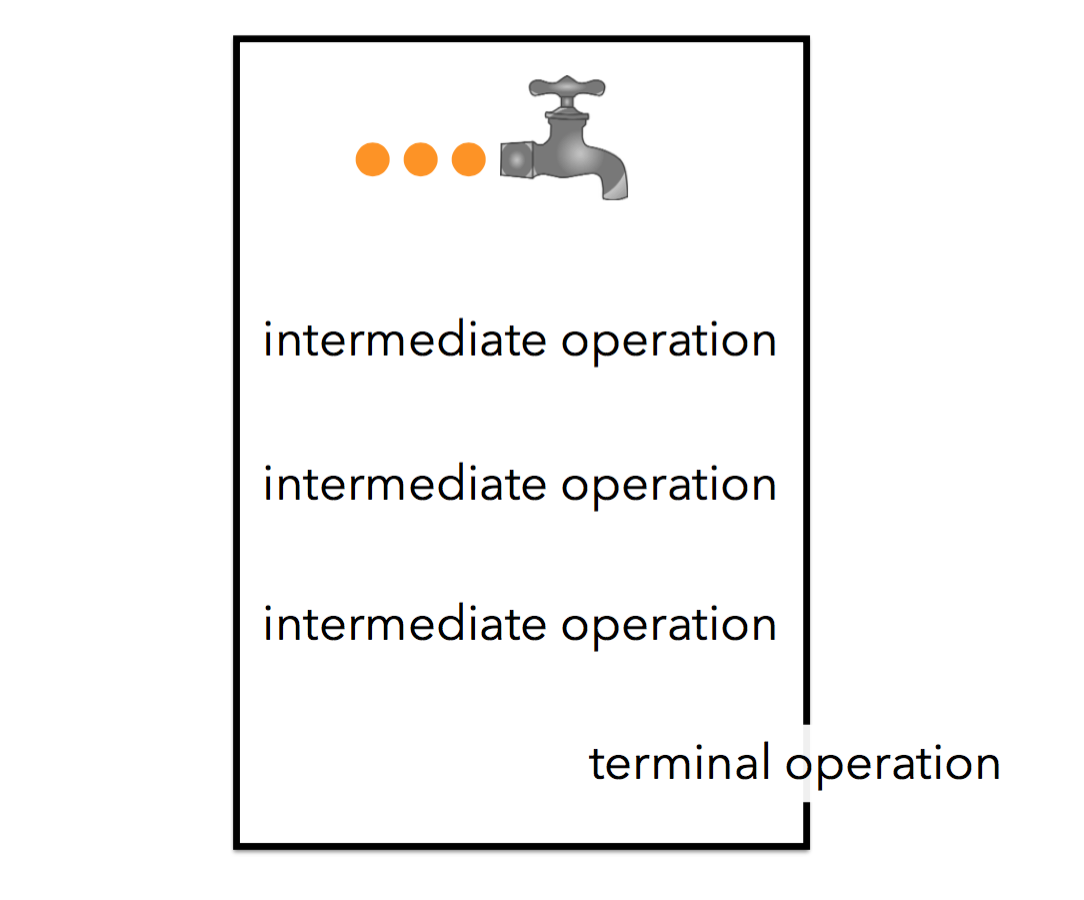
Return a Stream

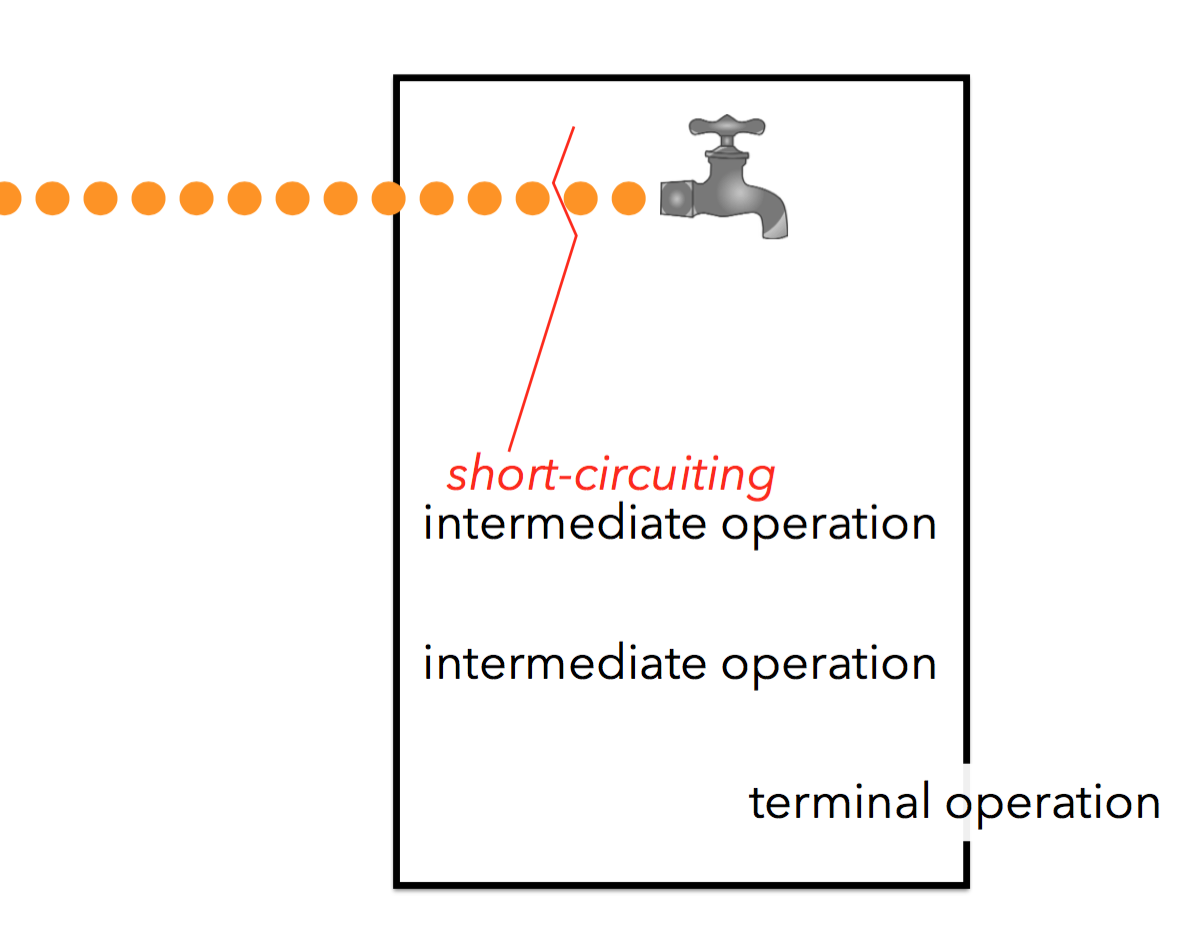
Let you keep writing intermediate operations

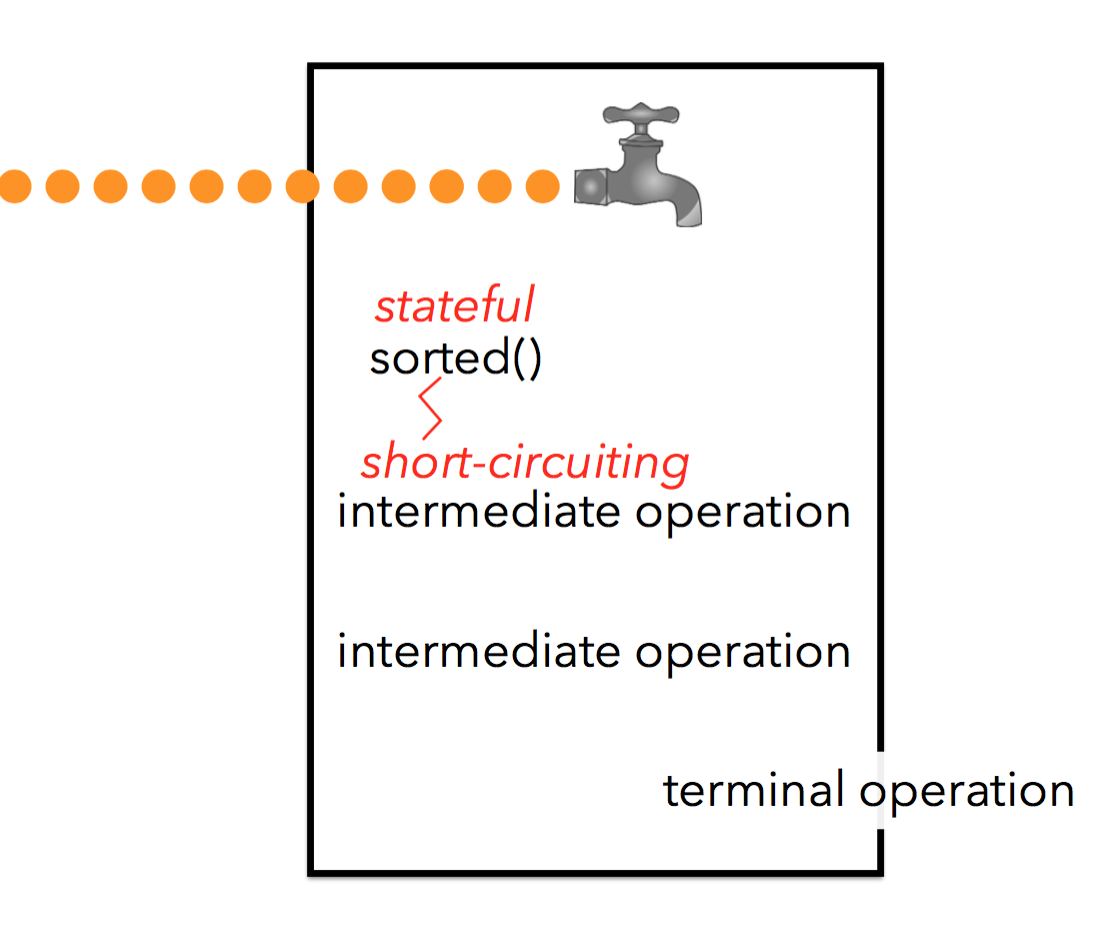


Infinite streams need short-circuiting operations

Optional







There are many function interfaces in Java standard api:

IntToDoubleFunction

Function<T, R>

BiFunction<T, U, R>

BinaryOperator<T>

Stream<E>:

filter

map

reduce

anyMatch(Predicate<? super T>)

allMatch(Predicate<? super T>)

findFirst

mapToDouble

sum

count

distinct

parallel

sorted

forEach

generate

limit

infiniteStream

ToDoubleFunction

Predicate

Optional<T>:

orElese

of

empty

Supplier<T>

Consumer<T>