**JAVA 8 - Cheat Sheet**

**Lambda Expression**

(int a) -> a \* 2; // Calculate the double of a

a -> a \* 2; // or simply without type

(a, b) -> a + b; // Sum of 2 parameters

If the lambda is more than one expression we can use { } and return

(x, y) -> {

int sum = x + y;

int avg = sum / 2;

return avg;

}

A lambda expression cannot stand alone in Java, it need to be associated to a functional interface.

interface MyMath {

int getDoubleOf(int a);

}

MyMath d = a -> a \* 2; // associated to the interface

d.getDoubleOf(4); // is 8

All examples with "list" use :

List<String> list = [Bohr, Darwin, Galilei, Tesla, Einstein, Newton]

**Collections**

**sort** sort(list, comparator)

list.sort((a, b) -> a.length() - b.length())

list.sort(Comparator.comparing(n -> n.length())); // same

list.sort(Comparator.comparing(String::length)); // same

//> [Bohr, Tesla, Darwin, Newton, Galilei, Einstein]

**removeIf**

list.removeIf(w -> w.length() < 6);

//> [Darwin, Galilei, Einstein, Newton]

**merge** merge(key, value, remappingFunction)

Map<String, String> names = new HashMap<>();

names.put("Albert", "Ein?");

names.put("Marie", "Curie");

names.put("Max", "Plank");

// Value "Albert" exists

// {Marie=Curie, Max=Plank, Albert=Einstein}

names.merge("Albert", "stein", (old, val) -> old.substring(0, 3) + val);

// Value "Newname" don't exists

// {Marie=Curie, Newname=stein, Max=Plank, Albert=Einstein}

names.merge("Newname", "stein", (old, val) -> old.substring(0, 3) + val);

**Method Expressions Class::staticMethod**

Allows to reference methods (and constructors) without executing them

// Lambda Form:

getPrimes(numbers, a -> StaticMethod.isPrime(a));

// Method Reference:

getPrimes(numbers, StaticMethod::isPrime);

| **Method Reference** | **Lambda Form** |
| --- | --- |
| StaticMethod::isPrime | n -> StaticMethod.isPrime(n) |
| String::toUpperCase | (String w) -> w.toUpperCase() |
| String::compareTo | (String s, String t) -> s.compareTo(t) |
| System.out::println | x -> System.out.println(x) |
| Double::new | n -> new Double(n) |
| String[]::new | (int n) -> new String[n] |

**Streams**

Similar to collections, but

* They don't store their own data
* The data comes from elsewhere (collection, file, db, web, ...)
* *immutable* (produce new streams)
* *lazy* (only computes what is necessary !)

// Will compute just 3 "filter"

Stream<String> longNames = list

.filter(n -> n.length() > 8)

.limit(3);

**Create a new stream**

Stream<Integer> stream = Stream.of(1, 2, 3, 5, 7, 11);

Stream<String> stream = Stream.of("Jazz", "Blues", "Rock");

Stream<String> stream = Stream.of(myArray); // or from an array

list.stream(); // or from a list

// Infinit stream [0; inf[

Stream<Integer> integers = Stream.iterate(0, n -> n + 1);

**Collecting results**

// Collect into an array (::new is the constructor reference)

String[] myArray = stream.toArray(String[]::new);

// Collect into a List or Set

List<String> myList = stream.collect(Collectors.toList());

Set<String> mySet = stream.collect(Collectors.toSet());

// Collect into a String

String str = list.collect(Collectors.joining(", "));

**map** map(mapper)  
Applying a function to each element

// Apply "toLowerCase" for each element

res = stream.map(w -> w.toLowerCase());

res = stream.map(String::toLowerCase);

//> bohr darwin galilei tesla einstein newton

res = Stream.of(1,2,3,4,5).map(x -> x + 1);

//> 2 3 4 5 6

**filter** filter(predicate)  
Retains elements that match the predicate

// Filter elements that begin with "E"

res = stream.filter(n -> n.substring(0, 1).equals("E"));

//> Einstein

res = Stream.of(1,2,3,4,5).filter(x -> x < 3);

//> 1 2

**reduce**  
Reduce the elements to a single value

String reduced = stream

.reduce("", (acc, el) -> acc + "|" + el);

//> |Bohr|Darwin|Galilei|Tesla|Einstein|Newton

**limit** limit(maxSize) The n first elements

res = stream.limit(3);

//> Bohr Darwin Galilei

**skip** Discarding the first n elements

res = strem.skip(2); // skip Bohr and Darwin

//> Galilei Tesla Einstein Newton

**distinct** Remove duplicated elemetns

res = Stream.of(1,0,0,1,0,1).distinct();

//> 1 0

**sorted** Sort elements (must be *Comparable*)

res = stream.sorted();

//> Bohr Darwin Einstein Galilei Newton Tesla

**allMatch**

// Check if there is a "e" in each elements

boolean res = words.allMatch(n -> n.contains("e"));

anyMatch: Check if there is a "e" in an element  
noneMatch: Check if there is no "e" in elements

**parallel** Returns an equivalent stream that is parallel

**findAny** faster than findFirst on parallel streams

**Primitive-Type Streams**

Wrappers (like Stream) are inefficients. It requires a lot of unboxing and boxing for each element. Better to use IntStream, DoubleStream, etc.

**Creation**

IntStream stream = IntStream.of(1, 2, 3, 5, 7);

stream = IntStream.of(myArray); // from an array

stream = IntStream.range(5, 80); // range from 5 to 80

Random gen = new Random();

IntStream rand = gen(1, 9); // stream of randoms

Use *mapToX* (mapToObj, mapToDouble, etc.) if the function yields Object, double, etc. values.

**Grouping Results**

**Collectors.groupingBy**

// Groupe by length

Map<Integer, List<String>> groups = stream

.collect(Collectors.groupingBy(w -> w.length()));

//> 4=[Bohr], 5=[Tesla], 6=[Darwin, Newton], ...

**Collectors.toSet**

// Same as before but with Set

... Collectors.groupingBy(

w -> w.substring(0, 1), Collectors.toSet()) ...

**Collectors.counting** Count the number of values in a group

**Collectors.summing\_\_** summingInt, summingLong, summingDouble to sum group values

**Collectors.averaging\_\_** averagingInt, averagingLong, ...

// Average length of each element of a group

Collectors.averagingInt(String::length)

*PS*: Don't forget Optional (like Map<T, Optional<T>>) with some Collection methods (like Collectors.maxBy).

**Parallel Streams**

**Creation**

Stream<String> parStream = list.parallelStream();

Stream<String> parStream = Stream.of(myArray).parallel();

**unordered** Can speed up the limit or distinct

stream.parallelStream().unordered().distinct();

*PS*: Work with the streams library. Eg. use filter(x -> x.length() < 9) instead of a forEach with an if.

**Optional**

In Java, it is common to use null to denote absence of result. Problems when no checks: NullPointerException.

// Optional<String> contains a string or nothing

Optional<String> res = stream

.filter(w -> w.length() > 10)

.findFirst();

// length of the value or "" if nothing

int length = res.orElse("").length();

// run the lambda if there is a value

res.ifPresent(v -> results.add(v));

Return an Optional

Optional<Double> squareRoot(double x) {

if (x >= 0) { return Optional.of(Math.sqrt(x)); }

else { return Optional.empty(); }

}

**Note on inferance limitations**

interface Pair<A, B> {

A first();

B second();

}

A steam of type Stream<Pair<String, Long>> :

* stream.sorted(Comparator.comparing(Pair::first)) // ok
* stream.sorted(Comparator.comparing(Pair::first).thenComparing(Pair::second)) // dont work

Java cannot infer type for the .comparing(Pair::first) part and fallback to Object, on which Pair::first cannot be applied.

The required type for the whole expression cannot be propagated through the method call (.thenComparing) and used to infer type of the first part.

Type *must* be given explicitly.

stream.sorted(

Comparator.<Pair<String, Long>, String>comparing(Pair::first)

.thenComparing(Pair::second)

) // ok