Java 8

GENERICS

Generic method:

```
public static <T> T lastEntry(List<T> list) {
        return list.size() > 0 ? list.get(list.size() - 1) : null;
}
```

Generic class:

```
public class someClass<T> {
        …
}

//somewhere else
someClass<Person> personClass = new someClass<>();
```

ANONYMOUS INNER CLASS

```
Arrays.sort(arrayOfStrings, new Comparator<String>(){
        @Override
        public int compare(String s1, String s2){
                return (s1.length() - s2.length());
        }
});
```

LAMBDAS

```
Arrays.sort(arraysOfStrings, (s1,s2) -> s1.length() - s2.length());
Arrays.sort(arrayOfStrings, (s1, s2) -> lengthCheck(s1,s2));
```

Step 1: Drop interface and method names.
Step 2: Drop parameter type declarations.
Step 3: Use expression instead of block.
Step 4: Omit parens when there is exactly one param.

Normally you think of lambdas as functions, but actually they become an instance of a class that implements whatever is expected.

Variables must be immutable!

SINGLE METHOD INTERFACE / FUNCTIONAL INTERFACE / SINGLE ABSTRACT METHOD (SAM)

```
@FunctionalInterface
public interface BestElement<T> {
	boolean isBetter(T element1, T element2);
}

static public <T> T betterElement(T element1, T element2, BestElement bestElement){
	if(bestElement.isBetter(element1, element2)){
		return element1;
	}
	return element2;
}

betterElement("Seppe", "Thiebault", (s1, s2) -> false);
betterElement("Seppe", "Thiebault", (s1, s2) -> s1.length() > s2.length());
betterElement(1, 2, (n1, n2) -> n1 > n2);
```

This wil return element1 if true, element2 if false.
@FuntionalInterface - If developer later adds a second method, interface will not compile. (Not Required)

BASIC METHOD REFERENCES

Replace:
(args) - > SomeClass.SomeStaticMethod(args);

With:
SomeClass::someStaticMethod;

4 kinds of Method References

Method Reference Type	Example	Lambda Equivalent
SomeClass::staticMethod	Math::Cos	x -> Math.cos(x)
someObject::instanceMethod	someString::toUpperCase	() -> someString.toUpperCase()
SomeClass::InstanceMethod	String::toUpperCase	s -> SomeClass.toUpperCase()
SomeClass::new	Person::new	() -> new Person()

PREDICATE (T IN -> BOOLEAN OUT)

```
public static <T> List<T> allMatches(List<T> list, Predicate<T> predicate) {
	List<T> newList = new ArrayList<>();
	for (T item : list){
		if (predicate.test(item)){
			newList.add(item);
		}
	}
	return newList;
}

List<String> shortWords = allMatches(words, s -> s.length() < 5);
List<String> wordsWithE = allMatches(words, s -> s.contains("e"));
List<String> evenLengthWords = allMatches(words, s -> (s.length() % 2) == 0);
List<Integer> bigNums = allMatches(nums, n -> n>500);
```

FUNCTION (T IN -> R OUT)

```
public static <T,R> List<R> transformedList(List<T> list, Function<T, R> function){
        List<R> newList = new ArrayList<>();
        for (T item : list){
                newList.add(function.apply(item));
        }
        return newList;
}

List<String> excitingWords = transformedList(words, s -> s + "!");
List<String> eyeWords = transformedList(words, s -> s.replace("i", "eye"));
List<String> upperCaseWords = transformedList(words, String::toUpperCase);
List<Integer> wordLengths = transformedList(words, String::length);
```

Other generically typed interfaces:

- *Consumer<T>* = T in -> nothing out
  Will use input, does side effects, like a void
- *Supplier<T>* = Nothing in -> T out;
  Lets you make a no-arg "function" that returns T. It can do so by calling "new", using an existing object, or anything else it wants.
- *BinaryOperator<T>* = Two T's in, one T out

HIGHER ORDER FUNCTIONS

Really objects that implement functional interfaces. You can also have a lambda that returns another lambda.

```
public static <T> Predicate<T> allPassPredicate(Predicate<T> ... PassPredicates){
        Predicate<T> result = e -> true;
        for(Predicate<T> predicate: PassPredicates) {
                result = result.and(predicate);
        }
        return result;
}

public static <T> T firstAllMatch(Stream<T> stream, Predicate<T> ... allMatches){
        Predicate<T> combinedTest = allPassPredicate(allMatches);
        return(stream.filter(combinedTest)
                .findFirst()
                .orElse(null));
}

String string = Lambdas4.firstAllMatch(words.stream(),
        word -> word.contains("o"),
        word -> word.length() > 5 );
```

Predicate

- And: True if both are true
- Or: True if one is true
- Negate: Opposite of outcome
- isEqual: True if both are true / false

Function

compose: f1.compose(f2) means first run f2 and pass the result to f1
andThen: f1.andThen(f2) means first run f1 then pass the result to f2

identity: Funtion.identity() creates a function whose apply method just returns the argument unchanged.

Comparator

comparing: Static method that takes function that returns a key and builds a Comparator from it.

Arrays.sort(words, Comparator.comparing(String::length).reversed());

reversed: Default method that imposes the revers ordering

thenComparing: Default method that specifies how to break ties in the initial comparison

Arrays.sort(employees, Comparator.comparing(Employee::getLastName)
.thenComparing(Employee::getFirstName));

STREAMS

Streams are not collections: they do not manage their own data. Instead, they are wrappers around existing data structures. When you make or transform a Stream, it does not copy the underlying data. Instead, it just builds a pipeline of operations. How many times a pipeline will be invoked depends on what you later do with the stream.
Streams are lazy! Most Stream operations are postponed until it knows how much data is eventually needed.
E.g. if you do a 1_-second-per-item operation on a 100-element stream, then select the first entry, it takes 10 seconds, not 1000 seconds.

forEach(Consumer)
Easy way to loop over Stream elements.You supply a function (as a lambda) to forEach, and that function is called on each element of the Stream.

*Stream.of(someArray)**.forEach**(System.out::println);*
*fieldList.stream()**.forEach**(field - > field.setText("…"));*

It can be made parallel with minor effort.

*someStream**.parallel()**.forEach(someLambda);*

map(Function)
Produces a new Stream that is the result of applying a Function to each element of the original stream.

```
Double[] nums = {1.0, 2.0, 3.0, 4.0, 5.0};
Double[] squares = Stream.of(nums).map(n -> n * n).toArray(Double[]::new);

Integer[] ids = {1, 2, 4, 6, 8};
List<Employees> matchingEmployees =
    Stream.of(ids).map(EmployeeUtils::findById).collect(Collectors.toList());
```

Filter(Predicate)
Produces a new Stream that contain only the elements of the original Stream that pass a given test (Predicate)

```
Integer[] nums = {1, 2, 3, 4, 5, 6};
Integer[] evens = Stream.of(nums).filter(n -> n % 2 == 0).toArray(Integer[]::new);

Integer[] evens = Stream.of(nums).filter(n -> n % 2 == 0)
                                 .filter(n -> n > 3).toArray(Integer[]::new);
```

findFirst()
Returns an Optional for the first entry in the Stream. Since Streams are often results of filtering, there might not be a first entry, to the Optional could be empty.

*someStream.map(…)**.findFirst()**.get();*
*someStream.filter(…)**.findFirst()**.orElse(otherValue);*

limit()
Returns a Stream of the first n elements

*someLongStream**.limit**(10);*

skip()
Returns a Stream starting with element n (and throws away the first n elements).

someLongStream**.skip**(5);

sorted()
Sorted with a Comparator works just like Arrays.sort.
Sorted with no arguments works only if the Stream elements implement Comparable.
Sorting Streams is more flexible than sorting arrays because you can do filter an mapping operations before and / or after.

*empStream**.sorted**((e1, e2) -> e1.getSalary() - e2.getSalary());*

min() & max()
It is faster to use min and max than to sort forward of backward, than take first element. Min and max take a Comparator as argument. Both return an Optional.

*empStream**.max**((e1, e2) -> e1.getSalary() -e2.getSalary()).get();*

distinct()
Distinct uses equals as its comparison. Is used to eliminate duplicates.

stringStream**.distinct**();

allMatch(), anyMatch() & noneMatch()
They stop processing once an anewer can be determined.
E.g. If the first element fails the predicate, allMatch would immediately return false an skip checking other elements.

*employeeStream**.anyMatch**(e -> e.getSalary() > 500000);*

count()
count simply returns the number of elements.

*employeeStream.filter(somePredicate)**.count**();*

IntStream()
A specialization of Stream that makes it easier to deal with ints. Does not extend Stream, but instead extends BaseStream, on which Stream is also built.
Can make IntStream form int[], whereas Integer[] needed to make Stream<Integer>
Methods like min and max take no arguments, unlike Stream versions that need a Comparator

*double totalCost = carList.stream()**mapToDouble**(Car::getPrice)**.sum**();*
*int population = countryList.stream().filter(Utils::inRegion)**.mapToInt**(Country::getPopulation).sum();*
***IntStream**.of(1, 2, 3, 4, 5);*

reduce()
Reduction operators take a Stream<T>, and combine or compare the entries to produce a single value of type T.
findFirst().orElse, findAny.orElse, min(), max(), sum(), average()

You start with a seed value, combine this value with the first entry of the Stream, combine the result with the second entry of the stream, and so forth.

reduce(starter, binaryOperator), Takes starter value and BinaryOperator. Returns result directly reduce(binaryOperator), *Takes BinaryOperator, with no starter. It starts by combining the first 2 values with each other. Returns an Optional;*

```
nums.stream().reduce(Double.MIN_VALUE, Double::max);
nums.stream().reduce(1, (n1, n2) -> n1 * n2);
letters.stream().Reduce("", String::concat);
```

Find "biggest" employee:

```
Employee poorest = new Employee("None", "None", -1, -1);
BinaryOperator<Employee> richer = (e1, e2) -> {
        return (e1.getSalary() >= e2.getSalary() ? e1 : e2);
};
Employee richest = employees.stream().reduce(poorest, richer);
```

LAZY EVALUATION
"I'm not lazy, I'm just highly motivated to do nothing."

Streams defer doing most operations until you actually need the results.
Operations that appear to traverse Stream multiple times actually traverse it only once. Due to "short-circuit" methods, operations that appear to traverse entire stream can stom much earlier.

stream.map(someOp).filter(someTest).findFirst().get();
Does map and filter operations one element at a time. Continues only until first match on the filter test.

stream.map(…).filter(…).filter(…).allMatch(someTest);
Does the map, the two filters and the allMatch one element at a time. The first time it gets false for the allMatch test, it stops.

Intermediate methods
These are methods that produce other streams. These methods don't get processed until there is some terminal method called.

map, filter, distinct, sorted, peek, limit, skip, parallel, sequential, unordered

Terminal method
After on of these methods is invoked, the Stream is considered consumed and no more operations can be preformed on it.

forEach, forEachOrdered, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator

Short-circuit methods
These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

anyMatch, allMatch, noneMatch, findFirst, findAny, limit, skip

PARALLEL STREAMS

By designating that a Stream be parallel, the operations are automatically done in parallel, whiteout any need for explicit fork/join or threading code.

anyStream.parallel();
anyList.parallelStream();
Shortcut for anyList.stream().parallel();

Make sure the result of a serial and a parallel stream are the same. I they differ, do you care?

INFINITE / UNBOUNDED / ON-THE-FLY STREAMS

generate(valueGenerator)
Stream.generate lets you specify a Supplier. This Supplier is invoked each time the system needs a Stream element.

Supplier<Double> random = Math::Random;
*Stream**.generate**(random)limit(2).forEach(System.out::println);*

iterate(initialValue, valueTransformer)
Stream.iterate lest you specify a seed and an UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) becomes the third element, etc.

*List<Integer> powerOfTwo = Stream.iterate(1, n -> n * 2).limit(…).collect(Collectors.toList());*

The values are not calculated until they are needed. To avoid unterminated processing, you must use a size-limiting operator like limit of findFirst

COLLECT

Using methods in the Collectors class, you can output a Stream as many types.

List: *stringStream.collect(toList());*
String: stringStream.collect(joining(delimiter)).toString();
Set: *anyStream.collect(toSet());*
Other collection: *anyStream.collect(toCollection(CollectionType::new));*
Map: *strm.collect(partioningBy(e -> e.getSalary() > 1000000));*
strm.collect(groupingBy(Employee::getDepartment));

The StringJoiner Class
Java 8 added new StingJoiner class that builds delimiter-separated Strings, with optional prefix and suffix. They also added static “join” method to the string class, it uses StringJoiner.

```
StringJoiner joiner1 = new StringJoiner(“, “);
String result1 =  joiner1.add(“Java”).add(“Lisp”).add(“Ruby”).toString();

String result2 = String.join(“, “, “Java”, “Lisp”, “Ruby”);

String lastNames = ids.Stream().map(Employees::findGoogler)
                                .filter(e ->  e != null)
                                .map(Employee::getLastNames)
                                .collect(Collectors.joining(“, “))
                                .toString();
```

NEW LIST METHODS

forEach()
someList.forEach(someConsumer);
=
someList.stream().forEach(someConsumer);

removeIf()
someList.removeIf(somePredicate);
=
someList.stream().filter(somePredicate.negate());

replaceAll()
someList.replaceAll(someUnaryOperator);
=
someList.stream().map(someFunction);

Sort()
someList.sort(someComparator);
=
someList.stream().sorted(someCompartor);

DISCLAIMER: These methode modify the original List. This can not be used if you practice fictional programming!

NEW MAP METHODS

forEach(function)
For each Map entry, passes the key and value to the function.
map.forEach((key, value) -> System.out.println(key + " " + value);

replaceAll(function)
For each Map entry, passes the key ant the value to the function, then replaces existing value with that output.
shapeAreas.replaceAll((shape, area) -> Math.abs(area));

merge(key, initialValue, function)
Lets you update existing values. If no value is found, store the initial value otherwise pass the old en initial value to the function, and overwrite current result with that updated result.
messages.merge(key, message, (old, initial) -> old+initial);
messages.merge(key, message, String::concat);

computeIfAbsent(key, function)
Lets you remember previous computations. If value is found for the key, return is otherwise pass the key to the function, store the output in Map and return it.
map.computeIfAbsent(arg, val -> codeForOriginalMethod(val));

DISCLAIMER: These methode modify the original List. This can not be used if you practice fictional programming!