Java 8 features cheat sheet

Lambdas

Based on *@FunctionalInterface*. General syntax:

```
(parameters) -> expression
(x, y) -> {statements; return r;}
```

Interface	Input -> Return
Predicate<T>	T -> Boolean
Consumer<T>	T -> void
Supplier<T>	() -> T
Function<T,R>	T -> R
UnaryOperator<T>	T -> T
BinaryOperator<T>	(T,T) -> T
BiPredicate<L,R>	(L,R) -> Boolean
BiConsumer<L,R>	(L,R) -> void
BiFunction<L,R,U>	(L,R) -> U

Many variations exist for primitive types Integer, Double, Long (e.g. LongToDoubleFunction)

Method reference

Method references can be used where they match a required functional interface.

1. Static method `Integer::parseInt`
2. Instance method `String::length`
3. Existing object `car::getEngine`

Examples

```
list.sort(String::compareToIgnoreCase)
Supplier<Car> c1 = Car::new;
Car c = c1.get()
Function<Int..,Car> c2=Car::new;
Car c = c2.apply(10)
```

Creating Streams

Streams can be created in several ways – finite or infinite. Most common is Stream.of().

Stream creation method	Comment
Stream.of(T t)	From array
Stream.of(T... values)	Varargs version
Arrays.stream(array)	Is different from Stream.of() with primitives!
Stream.empty()	To prevent null values
Stream.generate(Supplier<T> s)	Generate infinite stream using supplier function
Stream.iterate(T seed, UnaryOperator<T> f)	Generates infinite series based on seed value
list.stream()	Elements of a list
map.entrySet().stream()	Entries of a map
String.chars()	Stream of Integers!

Stream methods

Streams are Terminal when they do not return a Stream<T>.

Here follow the key methods (the top section list the ***intermediate*** operations).

Method name(s)	Return	Argument
filter	Stream<T>	Predicate<T>
distinct	Stream<T>	
map	Stream<R>	Function<T,R>
flatMap	Stream<R>	Function<T, Stream<R>>
sorted	Stream<T>	Comparator<T>
peek	Stream<T>	Consumer<T>
limit	Stream<T>	long maxSize
skip	Stream<T>	long n
anyMatch, noneMatch, allMatch,	Boolean	Predicate<T>
findAny, findFirst	Optional<T>	
reduce	Optional<T>	BinaryOperator<T>
forEach	void	Consumer<T>
collect	R	Collector<T,A,R>
count	long	
max	T	Comparator<T>
min	T	Comparator<T>
toArray	Object[]	

Note that toArray can also be passed a constructor reference if you want an array of a specific type.

The map() function has many variants, such as mapToInt, mapToDouble, mapToObject. Choose the appropriate one!

Collecting results using collect()

The `Stream.collect()` operation accumulates elements from a stream into a container such as a collection. It comes in two forms.

The first form takes three arguments (1) a supplier of that which is to be created (e.g. HashSet::new), (2) an accumulator – that adds elements to the target (e.g. HashSet::add) and (3) a combiner that merges two objects into one (e.g. HashSet::addAll).

```
<R> R collect(Supplier<R> supplier,
BiConsumer<R,? super T> accumulator,
BiConsumer<R,R> combiner)
```

The second form however, is most used, since it takes as argument a Collector:

`<R,A> R collect(Collector<? super T,A,R> collector)`

The `Collectors` class provides many convenience implementations of standard reduction operations such as Collectors.toList(), Collectiors.joining. The table below lists the most common ones.

Collectors.<function>	Argument
toList	
toSet	
toMap	Function keyMapper, Function valueMapper
toCollection	Supplier<Collection>
counting	
summing(Int|Long|Double)	ToIntFunction<? super T> mapper
joining	String delimiter
groupingBy	Function<T,K>
groupingBy	Function<T,K>, Collector<? super T,A,D>
partitioningBy	Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
summarizing(Int|Long|Double)	ToIntFunction<? super T> mapper

Here are some examples:

Create a list:

```
List<String> result =
stream.collect(Collectors.toList())
```

Create a TreeSet:

```
Set<String> result =
stream.collect(Collectors.toCollection(Tree
Set::new))
```

Create a Map from a stream of Person objects:

```
Map<Integer, String> results =
stream.collect(Collectors.toMap(Person::get
Id, Person.getName))

Map<Integer, Person> results =
stream.collect(Collectors.toMap(Person::get
Id, Function.identity())
```

Note that `Function.identity()` returns the object itself.

Collectors.groupingBy and partitioningBy

When you want to group your results on some property, use Collectors.groupingBy(). The groupingBy() function has two forms. The first one takes only a "classifier function" which will default to Lists of grouped values as value in the returned Map:

```
Map<String, List<Person>>
personsByProfession =
stream.collect(Collectors.groupingBy(Person
::getProfession))
```

The second form takes a second argument, besides the classifier: a "downstream collector" that determines the type of grouping value in the map.

```
Map<String, Set<Person>>
personsByProfession =
stream.collect(Collectors.groupingBy(Person
::getProfession, Collectors.toSet()))
```

Note that you can make these statements a bit more readable by using a static import of Collectors:

```
import static
java.util.streams.Collectors.*;

Map<String, Set<Person>>
personsByProfession =
stream.collect(groupingBy(Person::getProfes
sion, toSet()))
```

Partitioning is similar:

```
Map<Boolean, List<Person>> managerOrNot =
stream.collect(Collectors.partitioningBy(p
-> p.getProfession().equals("manager"))
```

Optional

The key to using Optional effectively is to *use a method that either consumes the correct value or produces an alternative*.

Key methods:

Optional	Return
Optional.of(T)	Optional<T>
get()	T
isPresent()	boolean
ifPresent(Consumer<T>)	void
orElse(T)	T
orElseGet(Supplier<T>)	T
orElseThrow(Supplier<X>)	X

For example,

```
String result = optionalString.orElse("")
```

will assign an empty string to result if the optional contains a null value.