Java8 Cheat Sheet

Java 8 Interfaces

- default methods can be implemented in the interface and can be overriden by implementations
- static methods can be called directly from the interface and can't be overriden

Functional Interfaces

Interfaces with a single unimplemented method must add @FunctionalInterface decorator, they are instantiated through anonymous classes, or lambda expressions

Interface

```
@FunctionalInterface
public interface Log {
    void log(String text);
}
```

Implementation

```
//Anonymous Class
Log logger = new Log() {

    @Override
    public void log(String text) {
        System.out.println(text);
    }
};

logger.log("Hello World!!");
```

Lambda Expressions

This Anonymous classes that have one single method can be converted to Lambda expressions:

Interface

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

Implementation

```
MathOperation addition = (int a, int b) -> a + b;
int result = addition.operation(1, 2);
```

There are some existing interfaces that can be used this way for example Comparator

```
List<Person> personList = new ArrayList<>();
personList.sort((p1,  p2) -> p2.getName().compareTo(p1.Name()));
```

Java8 Existing functional interfaces

For each functional interface it can be Bi/Tri and each parameter can have different Type:

- Supplier<R> → T Receives no parameters and returns an object of type T *get*
- Consumer<T> T → Receives an object of type T returns void *accept*
- Predicate<T> T → boolean Receives an object of type T returns boolean *test*
- Function<T, R> T → R Receives an object of type T returns an object of type R *apply*

A closure in Java can be defined to be a lambda expression, together with the values of the free variables that are captured by the lambda expression.

Method Reference

Instead of creating a lambda expression with the -> syntax you can use :: to create one that matches a Class or Object method signature

Lambda Way

```
listOfElements.forEach(element -> System.out.println(element));
```

forEach receives a consumer that means receive a type and returns void;

Method Reference Way

```
listOfElements.forEach(System.out::println);
```

we can do this because System.out.println is a method that returns void and receive an object (or a string).

Stream API

Stream is a pipeline of operations, you need a Collection to create it or use Stream interface static methods

Create a stream from collection

```
Stream<T> myStream = listOfElements.stream();
```

Use Stream Interface to create streams

```
Stream<String> myStream = Stream.generate(() -> "Hello");
myStream.forEach(System.out::println);
```

Streams always have a type <T> in order to make operations:

Stream Operations

Terminations

Collect

Always must coll a Stream in order to transform it into a data structure, 95% of the time the Collector implementation already exists in the Collectors class

```
List<T> listOfElements = myStream.collect(Collectors.toList());
```

- Collectors.toList() or .toSet(): returns all the elements in form of a list/Set
- Collectors.toCollection(ArrayList:new): returns a concrete collection of the type instantiated
- Collectors.toMap(<K> key, <V> value):

```
articles.collect(Collectors.toMap(Article::getTitle, Function.identity()));
```

Function.identity returns the object in the stream

- Collectors.collectingAndThen(transformation, listOperation) do a transformation after collecting
- Collectors.joining(separator) convert it into a single String separated by input
- Collectors.counting() counts the elements on the stream
- Collectors.maxBy()/minBy()/ returns the biggest/smallest element in the stream
- Collectors.groupingBy() group elements by some property and store them in a map

```
Map<Integer, Set<String>> result = givenList.stream()
    .collect(groupingBy(String::length, toSet()));

assertThat(result)
  .containsEntry(1, newHashSet("a"))
  .containsEntry(2, newHashSet("bb", "dd"))
  .containsEntry(3, newHashSet("ccc"));
```

- Collectors.partitioningBy() receives a predicate and group elements by thise who pass or failed the test

```
Map<Boolean, List<String>> result = givenList.stream()
  .collect(partitioningBy(s -> s.length() > 2))
// {false=["a", "bb", "dd"], true=["ccc"]}
```

Reduce

Accumulate elements on the response type, a seed Object and a BiFunction<T, T, V>

```
List<String> listOfNames = personsStream.reduce(new ArrayList<String>(), (accumulator, person) ->
```

```
accumulator.add(person.getName()));
```

Count

Returns a long with the number of elements in the stream

```
long count = elements.stream().count();
```

Operations

Map

- Transform one type into anoter type receives a Function<T,R>

Filter

- Get a subset of items that pass a test, receives a Predicate<T>

Distinct

- Get a subset of items that is not repeated in the stream, receives no functional interface

Sorted

- Returns the elements sorted as specified, receives a Comparator<T, T, int> if not set uses natural ordering

```
list.stream().sorted();
list.stream().sorted(Comparator.reverseOrder());
list.stream().sorted(Comparator.comparing(Student::getAge))
list.stream().sorted(Comparator.comparing(Student::getAge).reversed())
```

Peek

- Does not modify the stream but can place a Consumer<T> in the middle to operate on it

```
stringList.stream().peek(System.out::println);
```

Other useful methods

- skip(int n): skips the first n elements and returns the rest
- limit(int n): limits the stream to the first n elements
- flatMap(Function): convert a stream of streams<T> into a stream of elements of the type <T>