**STREAM**

* Gives access to Stream API functions.
* Get an instance through:
  + A Collection:

**List<Integer> list = new ArrayList<>();**

**Stream<Integer> stream = list.stream();**

* + An Array:

**String[] array = new String[10];**

**Stream<String> stream = Arrays.stream(array);**

* + A Hash Map Collection:

**HashMap<String, String> map = new HashMap<>();**

**Stream<Map.Entry<String, String>> entries = -> whole map  
 map.entrySet().stream();**

**Stream<String> keys = map.keySet().stream(); -> keys**

**Stream<String> keys = map.values().stream(); -> values**

* Each function call creates a new Stream<T> instance
  + This allows method chaining

**List<String> strings = new ArrayList<>();**

**Stream<String> stringStream = strings.stream();**

**Stream<Integer> intStream =   
 stringStream.map(s -> s.length());**

* **So what is a Stream ?**
  + Stream is not a collection and don`t store any data
  + Stream iterates over a collection
  + Does not modify data it processes
* Generic Streams
  + Can be of any type except primitives

**List<String> strings = new ArrayList<>();**

**Stream<String> strStream = strings.stream();**

**List<Integer> ints = new ArrayList<>();**

**Stream<Integer> intStream = ints.stream();**

**List<Object> objects = new ArrayList<>();**

**Stream<Object> objStream = objects.stream();**

* Optional<T>
  + Some functions can return Optional<T>

**Optional<String> first = elements.stream()**

**.sorted()**

**.findFirst();**

**if (first.isPresent()) {**

**System.out.println(first.get());**

**else**

**System.out.println("No matches.");**

* Primitive Streams
  + Work efficiently with primitive types
  + Give access to additional functions

**int[] ints = { 1, 2, 3, 4 };**

**IntStream intStream = IntStream.of(ints);**

**List<Integer> list = new ArrayList<>();**

**IntStream mappedIntStream = list.stream()**

**.mapToInt(n -> Integer.valueOf(n));**

* Intermediate Operations
  + Does not terminate the STREAM

**List<String> elements = new ArrayList<>();**

**Collections.addAll(elements, "one", "two");**

**Stream<String> stream = elements.stream()**

**.distinct()**

**.sorted()**

**.filter(s -> s.length() < 5)**

**.skip(1)**

**.limit(1);**

* Terminal Operations
  + Terminates the stream

**List<String> elements = new ArrayList<>();**

**Collections.addAll(elements, "one", "two");**

**elements.stream()**

**.distinct()**

**.forEach(s -> System.out.println(s))**

* + Useful terminal operations:
    - Reduce
    - Collect
    - forEach
* Map Operations
  + Transform the objects in the stream

**Stream<String> strStream =**

**Stream.of("1", "2", "3");**

**Stream<Integer> numStream =   
 strStream.map(Integer::valueOf);**

* Filter Operations
  + Filters objects by a given spaces

**Stream<String> strStream =   
Stream.of("one", "two", "three")**

**.filter(s -> s.length() > 3);**

* Reduce Operations
  + Check for a given condition
    - Any element matches

**boolean any = stream1.anyMatch(x -> x % 2 == 0);**

* Find Reductions
  + Find an element:
    - Gets the first element of the stream

**Optional<Integer> first = list.stream()  
 .findFirst();**

* + - Gets any element of the stream:

**Optional<Integer> first = list.stream()  
 .findAny();**

* General Reduction
  + Applies a given lambda:

**Optional<Integer> first = list.stream()  
 .reduce((x, y) -> x + y);**

* Sorting
  + Sort by passing a comparator as lambda:

**List<Integer> numbers = new ArrayList<>();**

**Collections.addAll(numbers, 7, 6, 3, 4, 5);**

**numbers.stream()**

**.sorted((x1, x2) -> Integer.compare(x1, x2))   
 .forEach(System.out::println);**

* Use any dimension of the **Hash Map** :
  + Stream over the Entry set:

**Stream<Map.Entry<String, String>> entries =  
 map.entrySet().stream();**

* + Stream over the Key set:

**Stream<String> keys = map.keySet().stream();**

* + Stream over the Value set:

**Stream<String> keys = map.values().stream();**

* Collectors
  + Collecting a Stream into a list:

**String[] strings = { "22", "11", "13" };**

**List<Integer> numbers = Arrays.stream(strings)**

**.map(Integer::valueOf)**

**.collect(Collectors.toList());**

* + You can collect streams into different collections:
    - Arrays, Sets, HashMaps, etc.

**Summary**

* **Stream API is used to traverse and query collections**
  + **Streams have "lazy" execution**
* **Streams can be Generic or Primitive**
* **Types of Operations**
  + **Intermediate, Terminal**
  + **Mapping, Filtering and Reducing**
  + **Sorting**
* **Streams can be collected into a collection**