1. **Streams**: A stream represents a sequence of objects and operates on data source like array/ collection.
   1. A stream simply moves data possibly filtering sorting or otherwise operating on that data in the process.
   2. A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.
   3. A stream operation itself doesn’t modify the source rather creates a new stream.
   4. Stream library is defined in java.util.stream
   5. **Terminal operations**: Terminal ops on streams consume it. Once consumed a stream can’t be reused.
   6. **Intermediate operations:** create another stream. They can be used to pipeline that performs sequence of operations, Intermediate operations doesn’t execute until a terminal operation is encountered.
      1. **Stateful:** Processing of element depends on other e.g., sorted()
      2. **Stateless:** Processing of element doesn’t depends on other e.g., filter()
   7. **Short circuit operations**:
      1. **Intermediate:** limit(), skip()
      2. **Terminal:** anyMatch, allMatch, noneMatch, findFirst and findAny
2. **BaseStream**: Defines basic functionality defined in a stream.

**interface** BaseStream<T, S **extends** BaseStream<T, S>> **extends** AutoCloseable

* 1. **Methods**: close(), isParallel(), allMatch(), anyMatch(), noneMatch(), distinct(),
     1. **Creation:** of()
     2. **Terminal**: iterator(), splitarator(),
     3. **Intermediate**: unordered(), onClose(runnable), parallel(), sequential()
  2. Stream, [Double|Int|Long]Stream extends BaseStream

1. **Stream**: **interface Stream<T> extends BaseStream<T, Stream<T>>**
   1. **Methods**:
      1. **Terminal**: collect(Collector<T, A, R>), count(), forEach(Consumer), Optional max(Comparator<T>), Optional min(Comparator<T>), reduce(T identity, BinaryOperator<T>), findFirst(), findAny(), anyMatch(predicate), allMatch(predicate)
      2. **Intermediate**: filter(Predicate), map(Function<T, R>), mapTo<Double|Int|Long>, sorted(), toArray()
2. **Primitive stream**: <Double|Int|Long>Stream extends Stream to generate stream of primitives
3. **Creation**:
   1. **Collections**: streams(), parallerStream(),
   2. **Array**: stream(T[] array)
   3. IO
4. **Reduction:** Stream.reduce() take input as an accumulator function which takes two parameters: a partial result of the reduction (in this case, the sum of all processed integers so far) and the next element of the stream (in this case, an integer). It returns a new partial result. In this case, the accumulator function is a lambda expression that adds two Integer values and returns an Integer value**.**
5. **Optional<T>**: A container object which may or may not contain a non-null value.
   1. **Methods**: isPresent(), get()
      1. **Query**: isPresent([consumer]), ifPresentOrElse(consumer, runnable), equals(), get()
      2. **Factory**: empty(), of(), ofNullable()[wraps nulls returns empty optional], stream(), or(supplier), orElseGet(supplier), orElseThrow(supplier)
      3. **Operations**: map(mapper), filter(predicate)
   2. Stream terminal operations return Optional object. Some of these methods are:

Optional<T> reduce(BinaryOperator<T> accumulator)

Optional<T> min(Comparator<? super T> comparator)

Optional<T> max(Comparator<? super T> comparator)

Optional<T> findFirst()

Optional<T> findAny()

1. **Parallel streams:** Any operation applied on parallel stream must be stateless, non-interfering and associative.
   1. <U> U reduce(U identity, BiFunction<U, ? **super** T, U> accumulator, BinaryOperator<U> combiner): Here combiner combines two values produced by accumulator.
      1. An accumulator should be stateless, non-interfering and associative.
   2. Parallel streams may be efficient for unordered elements. If order doesn’t matter call unordered()
   3. forEach() in parallel stream may not preserve ordering. For ordering we need to call forEachOrdered()
2. **Mapping:** <R> Stream<R> map(Function<? **super** T, ? **extends** R> mapper)
   1. To map to primitive stream: mapTo<Int|Long|Double>(To<Int|Long|Double>Function)
   2. **Stream.flatMap**, as it can be guessed by its name, is the combination of a map and a flat operation. That means that you first apply a function to your elements, and then flatten it. Stream.map only applies a function to the stream without flattening the stream.

To understand what flattening a stream consists in, consider a structure like [ [1,2,3], [4,5,6], [7,8,9] ] which has "two levels". Flattening this means transforming it in a "one level" structure : [ 1,2,3,4,5,6,7,8,9 ].

1. **Collecting**: To obtain collection from a stream provides collect method.
   1. <R, A> R collect(Collector<? **super** T, A, R> collector)
   2. <R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner):
   3. The result of collect operation is mutable storage object.
   4. Collector class methods: toList, toSet, toMap, summarizingInt, joining, averagingInt, counting,
2. **Splitarator**:
   1. **Methods**:
      1. **TryAdvance(Consumer):** returns false when there is no item to process, thus makes looping construct very simple.
      2. **forEachRemaining(Consumer):** performs action on each element collectively than in loop.
      3. **trySplit():** split elements to be iterated in two parts and return Splitarator reference to new part where original part remain accessible through old spliterator.