# Java 8

## Lambda expressions

* + Functional interface
    - Predicate<T> -> **boolean** *test*(T t)
    - BiPredicate<T, U> -> **boolean** *test*(T t, U u)
    - Supplier<T> -> **T** *get()*
    - Consumer<T> -> **void** *accept*(T t)
    - BiConsumer<T, U> -> **void** *accept*(T t, U u)
    - Function<T, R> -> **R** *apply*(T t)
    - BiFunction<T, U, R> -> **R** *apply*(T t, U u)
    - UnaryOperator<T> -> **T** *apply*(T t)
    - BinaryOperator<T> -> **T** *apply*(T t1, T t2)
  + Final and Effectively Final
    - Lambda takes a snapshot of local variables. Local vars that’s used in lambda MUST NOT change (effectively final), it gives compiler error, even if it’s changed before lambda definition.
  + Method References – turns into lambdas in the background
    - Bound – bounded to some instance -> don’t have to specify which instance to call it on
    - Unbound – need to specify which instance to call it on. First parameter is used for executing the instance method
    - Static (Unbound) – calling static method in lambda
    - Constructor MR – calling constructor in lambda -> in MR just Type::new

## Streams

* Source – where the stream comes from e.g. array, collection or file
* Intermediate operations – transforms the stream into another one
  + filter()
  + distinct() – returns a stream with duplicate values removed
  + limit()
  + map() – creates a one-to-one mapping between elements in the stream and elements in the next stage of the stream. It transforms the data.
  + flatMap() – takes each element in the stream e.g. Stream<List<String>> and makes single stream Stream<String> that contains all elements
  + sorted(Comparator?.comparing(…))
* Terminal operation – required to start the whole process and produces the result
  + *reduce*() – combines a stream into a single object. It processes all elements
    - T reduce(T identity, BinaryOperator<t> accumulator)
    - Optional<T> reduce(BinaryOperator<T> accumulator)
    - <U> U reduce (U identity, BiFunction accumulator, BinaryOperator combiner)
  + *collect()* – mutable reduction because we use the same mutable object while accumulating. This makes it more efficient than regular reductions
    - Collectors
      * .joining()
      * .averagingInt()
      * .toMap()
      * .groupingBy(Predicate)
      * .partitioningBy(Predicate) – special case of grouping where there are only two possible groups *true* and *false*

## Collections and Generics

## Concurrency

* new Thread().start() -> creating new thread and method is executed in that new thread
* new Thread().run() -> doesn’t create new thread, method is executed in same thread
* new Thread().join(); -> main thread must wait here until new Thread() finishes task
* ExecutorService
  + Single thread pool executor
    - Tasks are processed squentially
  + Cached thread pool executor
    - Creates new threads as needed and reuses threads that have become free
    - Care needed as the number of threads can become very large
  + Fixed thread pool executor
    - Creates a fixed number of threads which is specified at the start
  + A *Future<V>* object represents the result of an asynchronous computation.
    - Methods are provided to check if the computation is complete – *isDone()* and
    - To retrieve the result of that computation – *get().*
    - The result can only be retrieved using the method *V get()* when the computation has completed, blocking if necessary until it is ready.