Lambda/Streams Programming Laboratory
Java 8/9 Quick Reference

Lambda Syntax

```
() -> 42
a -> a + 1                  // can omit parentheses for single untyped parameter
(a,b) -> a + b              // but required for multiple parameters
(int a, int b) -> a + b     // type either all parameters, or none
() -> { println(42); return 42; }     // lambda body can be a statement
```

Method References

static	`RefType::staticMethod`	`(args) -> RefType.staticMethod(args)`
bound	`expr::instMethod`	`(args) -> expr.instMethod(args)`
unbound	`RefType::instMethod`	`(arg0,rest) -> arg0.instMethod(rest)`
ctor	`Clsname::new`	`(args) -> new ClsName(args)`

Iterable<T>

`forEach(Consumer<T> c)`
supplies each element in turn to c

`removeIf(Predicate<T> p)`
removes elements satisfying p

List<E>

`replaceAll(UnaryOperator<E> op)`
replaces each element with the result of applying op *to it*

Map<K,V>

`compute(K key, BiFunction<K,V,V> remappingFunction)`
computes a new mapping from key *and its current mapped value (or null)*

`computeIfAbsent(K key, Function<K,V> mfn)`
if there is no current mapping for key, *creates one using* mfn *and enters it*

`computeIfPresent(K key, BiFunction<K,V,V> remappingFunction)`
creates a mapping from key *(if present and non-null) and its current mapped value*

`forEach(BiConsumer<K,V> c)`
executes c *for each key-value pair in this* Map

`merge(K key, V v, BiFunction<V,V,V> remappingFunction)`
if key *present, calculates new value from old value and* v, *o.w. associates key with* v

`putIfAbsent(K key, V value)`
associates key *with* value *if* key *is not present or is null*

`remove(Object k, Object val)`
removes mapping for k *from this map if it is present and has the value* val

`replace(K key, V oldValue, V newValue)`
replaces the entry for key *only if currently mapped to* oldvalue

`replaceAll(BiFunction<K,V,V> fn)`
replaces each entry's value with the result of applying fn *to the entry's key and value*

Comparator<T>

`comparing(Function<T,U> keyExtractor)`
returns a Comparator *that compares on the field extracted by* keyExtractor

`nullsLast(Comparator<T> comp)`
returns a Comparator *that accepts* null, *treating it as greater than non-null*

`reversed()`
returns a Comparator *that imposes the reverse ordering of this* Comparator

`thenComparing(Comparator<T> comp)`
returns a Comparator *that orders elements considered equal by this* Comparator

Stream<T> intermediate operations

`distinct()`
removes duplicates

`dropWhile(Predicate<? super T> predicate)` – Java 9+
for an ordered stream, drops the longest prefix of elements satisfying predicate

`filter(Predicate p)`
removes elements not satisfying p

`flatMap(Function<T, Stream<R>>)`
transforms each element to a stream of R, *then flatten these into a single stream*

`skip(int n)/limit(int n)`
*discards(*skip*) or preserves(*limit*) the first* n *elements*

`map(Function<T,R> mapper)/mapToInt/mapToObj`
transforms each element using the function mapper

`sorted()/sorted(Comparator<T> comp)`
sorts elements according to natural order, or the supplied Comparator

`takeWhile(Predicate<T> predicate)` – Java 9+
for an ordered stream, returns the longest prefix of elements matching predicate

Stream<T> terminal operations

`collect(Collector<T,A,R> collector)`
uses collector *to perform a mutable reduction operation on the stream elements*

`count()`
returns the count of stream elements

`max(Comparator<T> comp)`
returns the maximum element using comp *(*comp *not required for* IntStream*)*

`reduce(BinaryOperator<T> acc) / reduce(T id, BinaryOperator<T> acc)`
reduces stream elements (only use for immutable and primitive values)

`toArray()`
returns an array containing the stream elements

Collectors

`counting()`
returns a Collector *that counts the number of input elements.*

`filtering(Predicate<T> p, Collector<T,A,R>> c)` – Java 9+
returns a Collector that filters elements over c *before collecting*

`flatMapping(Function<T,Stream<U>> m, Collector<U,A,R> c)` – Java 9+
returns a Collector that applies the function m *to each elements before collecting*

CONTINUED OVERLEAF

Lambda/Streams Programming Laboratory
Java 8/9 Quick Reference

Collectors (continued)

`groupingBy(Function<T,K> classifier)`
returns a Collector *that classifies elements into a* Map<K, List<T>>

`joining() / joining(CharSequence delimiter)`
returns a Collector *that concatenates* String *elements (with the given delimiter)*

`partitioningBy(Predicate p)`
returns a Collector *that classifies elements into a* Map<Boolean, List<T>>

`summingInt(ToIntFunction<T> mapper)`
returns a Collector *that sums the result of mapping each element with* mapper

`toList()`
returns a Collector *that accumulates elements into a new* List

`toSet()`
returns a Collector *that accumulates elements into a new* Set

Pattern

`splitAsStream(CharSequence input)`
returns a Stream<String> *by splitting* input *around matches of the pattern*

Matcher

`results()` – Java 9+
returns a stream of matches for the input pattern

`replaceAll(Function<MatchResult,String> fn)` – Java 9+
replaces every match of the input pattern with the result of applying fn

Scanner

`findAll(Pattern p)` – Java 9+
returns a stream of matches for the input pattern

`tokens()` – Java 9+
returns a stream of delimiter-separated tokens