Java SE 8 for the Really Impatient

Chapter 1

* A lambda expression is a block of code with parameters.
* Use a lambda expression whenever you want a block of code executed at a later point in time.
* Lambda expressions can be converted to functional interfaces.
* Lambda expressions can access effectively final variables from the enclosing scope.
* Method and constructor references refer to methods or constructors without invoking them.
* You can now add default and static methods to interfaces that provide concrete implementations.
* You must resolve any conflicts between default methods from multiple interfaces.
  1. Why lambdas?

Examples:

* Concurrency (a block of code was passed to a thread pool)
* Comparator (a block of code was passed to a sort() method)

**Integer.compare(x,y)** returns 0 if x == y, negative number if x<y and positive number, if x>y.

* Listener interface (a block of code was passed to a button)

1.2) The syntax of lambda expressions

**(String first, String second) -> Integer.compare(first.length(), second.length());**

is equivalent to

**(String first, String second) ->{**

**if (first.length() < second.length()) return -1;**

**else if (first.length() > second.length()) return 1;**

**else return 0;**

**}**

Lambda expression with no parameters:

**() -> { for (int i = 0; i < 1000; i++) doWork(); }**

If the parameter types of a lambda expression can be inferred, you can omit them.

**Comparator<String> comp = (first, second) -> Integer.compare(first.length(), second.length());**

If a method has a single parameter with inferred type, you can even omit the parentheses:

**EventHandler<ActionEvent> listener = event -> System.out.println(“Thanks for clicking!”);**

1.3) Functional interfaces

Functional interface is an interface with a single abstract method.

**Arrays.sort(words, (first, second) -> Integer.compare(first.length(), second.length()));**

**button.setOnAction(event -> System.out.println(“Thanks for clicking!”));**

You can save our string comparison lambda in a variable of BiFunction<T, U, R> type:

**BiFunction<String, String, Integer> comp = (first, second) -> Integer.compare(first.length(), second.length());**

However, that does not help you with sorting.

You can tag any functional interface with the **@FunctionalInterface** annotation.

This has 2 advantages:

* The compiler checks that the annotated entity is an interface with a single abstract method.
* The javadoc page includes a statement that your interface is a functional interface.

Checked exceptions matter when a lambda is converted to an instance of a functional interface. If the body of a lambda expression may throw a checked exception, that exception needs to be declared in the abstract method of the target interface.

1.4) Method references

**button.setOnAction(event -> System.out.println(event));**

**button.setOnAction(System.out::println);**

Suppose you want to sort strings regardless of letter case:

**Arrays.sort(strings, String::compareToIgnoreCase);**

There are 3 principal cases:

* **object::instanceMethod**
* **Class::staticMethod**
* **Class::instanceMethod**

Just like lambda expressions, method references don’t live in isolation. They are always turned into instances of functional interfaces.

**this::equals** is the same as **x -> this.equals(x)**

The method expression **super::instanceMethod** uses **this** as the target and invokes the superclass version of the given method.

In an inner class, you can capture the **this** reference of an enclosing class as **EnclosingClass.this::method** or **EnclosingClass.super::method**

1.5) Constructor references

Constructor references are just like method references, except that the name of the method is **new**.

You can turn list of strings into an array of buttons:

**List<String> labels = …;**

**Stream<Button> stream = labels.stream().map(Button::new);**

**List<Button> buttons = stream.collect(Collectors.toList());**

You can form constructor references with array types. For example, **int[]::new** is equivalent to the lambda expression **x -> new int[x]**

Array constructor references are useful to overcome a limitation of Java. It is not possible to construct an array of a generic type T. The expression **new T[n]** is an error since it would be erased to **new Object[n]**. The stream library solves that problem with constructor references:

**Button[] buttons = stream.toArray(Button[]::new);**

1.6) Variable scope

Often, you want to be able to access variables from an enclosing method or class in a lambda expression.

**public static void repeatMessage(String *text*, int *count*)**

**{**

**Runnable r = () -> {**

**for (int i = 0; i < *count*; i++)**

**{**

**System.out.println(*text*);**

**Thread.yield();**

**}**

**};**

**new Thread(r).start();**

**}**

A lambda expression has 3 ingredients:

* a block of code
* parameters
* values for the **free** variables, that is, the variables that are not parameters and not defined inside the code

The lambda expression has two **free** variables, **text** and **count**. Values of the variables have been ***captured*** by the lambda expression.

The technical term for a block of code together with the values of the free variables is a ***closure***.

In Java, lambda expressions are ***closures***. In fact, inner classes have been ***closures*** all along.

In a lambda expression, you can only reference variables whose value doesn’t change.

Mutating variables in a lambda expression is not threadsafe.

From Java 8 an inner class can access any effectively final local variable – that is, any variable whose value does not change.

There is a trick that lets a lambda expression update a counter in an enclosing local scope:

**int[] counter = new int[1];**

**button.setOnAction(event -> counter[0]++);**

Code like this is not threadsafe, but for a button callback that doesn’t matter.

It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

**Path first = Paths.get(“/usr/bin”);**

**Comparator<String> comp = (first, second) -> Integer.compare(first.length(), second.length());**

//Error: variable first already defined

When you use the **this** keyword in a lambda expression, you refer to the **this** parameter of the method that creates the lambda.

**public class Application()**

**{**

**public void doWork()**

**{**

**Runnable runner = () -> {…; System.out.prinln(*this*.toString()); …};**

**}**

**}**

The expression **this.toString()** calls the **toString** method of the **Application** object, not the **Runnable** instance.

1.7) Default methods

**list.forEach(System.out::println);**

Default methods are interface methods with concrete implementations. Those methods can be safely added to existing interfaces.

What happens if the exact same method is defined as a default method in one interface and then again as a method of a superclass or another interface? The rules in Java are:

* Superclasses win.
* Interfaces clash. The Java compiler reports an error and leaves it up to the programmer to resolve the ambiguity. You can choose one of the conflicting methods, like this:

**class Student implements Person, Named**

**{**

**public String getName() { return Person.super.getName();}**

**}**

The “class wins” rule ensures compatibility with Java7. If you add default methods to an interface, it has no effect on code that worked before there were default methods (you can never make a default method that redefines one of the methods in the **Object** class).

1.8) Static methods in interfaces

Up to Java 8, it has been common to place static methods in companion classes. (interface **Collection**/class **Collections**, interface **Path**/class **Paths**)

In Java 8, one could have added methods from **Paths** to the **Path** interface:

**public interface Path**

**{**

**public static Path get(String first, String… more)**

**{**

**return FileSystems.getDefault().getPath(first, more);**

**}**

**}**

Then the Paths class is no longer necessary.

A method such as

**public static void shuffle(List<?> list)**

would work well as a default method of the List interface

**public default void shuffle()**

You could simply call **list.shuffle()** on any list.

**public static <T> List<T> nCopies(int n, T o)** could be a static method of the List interface. Then you would call **List.nCopies(10, “Fred”)** instead of **Collections.nCopies(10, “Fred”)** and it would be clear to the reader that the result is a list.

**Comparator.comparing(Person::name)**, using a “key extraction” function

**Comparator.compare(String::length)**, turns a function(the key extractor) into a more complex function (the key-based comparator).

Chapter 2

Streams are the key abstraction in Java 8 for processing collections of values and specifying what you want to have done, leaving the scheduling of operations to the implementation.

* **Iterators** imply a specific traversal strategy and prohibit efficient concurrent execution.
* You can create **streams** from **collections**, **arrays**, **generators**, or **iterators**.
* Use **filter** to select elements and **map** to transform elements.
* Other operations for transforming streams include **limit**, **distinct**, and **sorted**.
* To obtain a result from a stream, use a reduction operator such as **count**, **max**, **min**, **findFirst**, or **findAny**. Some of these methods return an **Optional** value.
* The **Optional** type is intended as a safe alternative to working with **null** values. To use it safely, take advantage of the **ifPresent** and **orElse** methods.
* You can collect stream results in **collections**, **arrays**, **strings**, or **maps**.
* The **groupingBy** and **partitioningBy** methods of the **Collectors** class allow you to split the contents of a stream into groups, and to obtain a result for each group.
* There are specialized streams for the primitive types **int**, **long**, and **double**.
* When you work with parallel streams, be sure to avoid side effects, and consider giving up ordering constraints.
* You need to be familiar with a small number of functional interfaces in order to use the stream library.

2.1) From iteration to stream operations

**String contents = new String(Files.readAllBytes(Paths.get(“alice.txt”)), StandardCharsets.UTF\_8);**

**List<String> words = Arrays.asList(contents.split(“[\\P{L}]+”));** //nonletters are delimiters

**int count = 0;**

**for(String w : words)**

**{**

**if (w.length() > 12) count++;**

**}**

It is hard to parallelize the such code. In Java 8, the same operation looks like this:

**long count = words.stream()**

**.filter(w -> w.length() > 12)**

**.count()** ;

* A stream doesn’t store its elements. They may be stored in an underlying collection or generated on demand.
* Stream operations don’t mutate their source. Instead, they return new streams that hold the result.
* Stream operations are lazy when possible. This means they are not executed until their result is needed.

Streams can be easily parallelized.

**long count = words.parallelStream()**

**.filter(w -> w.length() > 12)**

**.count();**

Streams follow the “what, not how” principle.

When you work with streams, you set up a pipeline of operations in three stages:

1. You create a stream.
2. You specify *intermediate operations* for transforming the initial stream into others, in one or more steps.
3. You apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

2.2) Stream creation

You can turn any collection into a stream with the **stream** method.

If you have an array, use the static **Stream.of** method instead:

**Stream<String> words = Stream.of(contents.split(“[\\P{L}]+”));**

The **of** method has a varargs parameter, so you can construct a stream from any number of arguments:

**Stream<String> song = Stream.of(“gently”, “down”, “the”, “stream”);**

Use **Arrays.stream(array, from, to)** to make stream from a part of an array.

**Stream<String> silence = Stream.empty();** // to make a stream with no elements

**Stream<String> echos = Stream.generate( () -> ”Echo”);** // get a stream of constant values

**Stream<Double> randoms = Stream.generate(Math::random);** // get a stream of random numbers

To produce infinite sequences such as 0 1 2 3…, use **iterate** method:

**Stream<BigInteger> integers = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));**

You can use the following statement to split a string into words:

**Stream<String> words = Pattern.compile(“[\\P{L}]+”).splitAsStream(contents);**

**try(Stream<String> lines = Files.lines(path))** // Files.lines returns a Stream of all lines in a file

**{**

**Do something with lines**

**}**

2.3) The **filter**, **map** and **flatMap** methods

The argument of filter is a Predicate<T> - that is, a function from T to boolean.

**List<String> wordList = ...;**

**Stream<String> words = wordList.stream();**

**Stream<String> longWords = words.filter(w -> w.length() > 12);**

To transform values in a stream:

transform all words to lowercase

**Stream<String> lowercaseWords = words.map(String::toLowerCase);**

the resulting stream contains the first character of each word

**Stream<Character> firstChars = words.map(s -> s.charAt(0));**

Function that returns a stream of values:

**public static Stream<Character> characterStream(String s) {**

**List<Character> result = new ArrayList<>();**

**for (char c : s.toCharArray()) result.add(c);**

**return result.stream();**

**}**

To get a stream of streams like: [... ['y', 'o', 'u', 'r'], ['b', 'o', 'a', 't'], ...]

**Stream<Stream<Character>> result = words.map(w -> characterStream(w));**

To flatten it out to a stream of characters: [... 'y', 'o', 'u', 'r', 'b', 'o', 'a', 't', ...]

**Stream<Character> letters = words.flatMap(w -> characterStream(w));**

2.4) Extracting substreams and combining streams

The call stream.limit(n) returns a new stream that ends after n elements (or when the original stream ends if it is shorter).

**Stream<Double> randoms = Stream.generate(Math::random).limit(100);**

The call stream.skip(n) does the exact opposite. It discards the first n elements.

**Stream<String> words = Stream.of(contents.split("[\\P{L}]+")).skip(1);** // skipping empty string at the beginning

**Stream<Character> combined = Stream.concat(characterStream(“Hello”), characterStream(“World”));**

// Yields the stream ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']

The peek method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

**Object[] powers = Stream.iterate(1.0, p -> p \* 2)**

**.peek(e -> System.out.println("Fetching " + e))**

**.limit(20).toArray();**

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by iterate is processed lazily.

2.5) Stateful transformations

**Stream<String> uniqueWords = Stream.of("merrily", "merrily", "merrily", "gently").distinct();**

// only one “merrily” is retained

**Stream<String> longestFirst = words.sorted(Comparator.comparing(String::length).reversed());**

// sort strings that the longest one comes first

The Collections.sort method sorts a collection in place, whereas Stream.sorted returns a new sorted stream.

2.6) Simple reductions

These methods reduce the stream to a value that can be used in your program. Reductions are *terminal operations*. After a terminal operation has been applied, the stream ceases to be usable.

**count()**

**min()**

**max()**

**Optional<T>** either wraps the answer or indicates that there is none.

**Optional<String> largest = words.max(String::compareToIgnoreCase);**

**if (largest.isPresent()) System.out.println("largest: " + largest.get());**

**findFirst()**

**Optional<String> startsWithQ = words.filter(s -> s.startsWith("Q")).findFirst();**

**findAny()**

**Optional<String> startsWithQ = words.parallel().filter(s -> s.startsWith("Q")).findAny();**

**anyMatch()**

**boolean aWordStartsWithQ = words.parallel().anyMatch(s -> s.startsWith("Q"));**

**allMatch()**

**noneMatch()**

2.7) The Optinal type

An **Optional<T>** object is either a wrapper for an object of type T or for no object. It is intended as a safer alternative than a reference of type T that refers to an object or null. But it is only safer if you use it right.

2.7.1) Working with Optional values

The key to using **Optional** effetively is to use a method that either *consumes the correct* value or *produces an alternative*.

There is a second form of the **ifPresent** method that accepts a function. If the **Optional**

value exists, it is passed to that function. Otherwise, nothing happens.

**optionalValue.ifPresent(v -> Process v);**

**optionalValue.ifPresent(v -> results.add(v));**

or

**optionalValue.ifPresent(results::add);**

If you want to process the result, use **map** instead:

**Optional<Boolean> added = optionalValue.map(results::add);** // now added has one of three values: true, false or empty optional.

The other strategy for working with optional is to produce an alternative if no value is present.

**String result = optionalString.orElse(“”);**

**String result = optionalString.orElseGet( () -> System.getProperty(“user.dir”) );** // the function is only called when needed

**String result = optionalString.orElseThrow(NoSuchElementException::new);**

2.7.2) Creating Optional values

There are several static methods to create Optional object.

**public static Optional<Double> inverse(Double x)**

**{**

**return x == 0 ? Optional.empty() : Optional.of(1/x);**

**}**

**Optional.ofNullable(obj)** returns **Optional.of(obj)** if obj is not null, and **Optional.empty()** otherwise.

2.7.3) Composing Optinal value functions with flatMap

Suppose you have a method **f** yielding an **Optional<T>**, and the target type T has a method **g** yielding an **Optional<U>**. If they were normal methods, you could compose them by calling

**s.f().g()**;

But that composition doesn’t work here, since s.f() has type **Optional<T>**, not T. Instead, call

**Optional<U> = s.f().flatMap(T::g);**

**public static Optional<Double> squareRoot(Double x)**

**{**

**return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));**

**}**

Then you can compute the square root of the inverse as

**Double result = inverse(x).flatMap(MyMath::squareRoot);**

or, if you prefer,

**Double result = Optional.of(-4.0).flatMap(Test::inverse).flatMap(Test::squareRoot);**

If either the inverse method or the squareRoot returns Optional.empty(), the result is empty.

2.8) Reduction operations

If you want to compute a sum, or combine the elements of a stream to a result in another way, you can use one of the *reduce* methods.

**Stream<Integer> values = …;**

**Optional<Integer> sum = values.reduce((x,y) -> x+y);**

or

**Optional<Integer> sum = values.reduce(Integer::sum);**

The operation should be *associative*: it should not matter in which order you combine the elements.

There are many associative operations that might be useful in practice, such as sum and product, string concatenation, maximum and minimum, set union and intersection.

**Stream<Integer> values = …;**

**Integer sum = values.reduce(0, (x,y) -> x+y);** //where 0 is *identity* for addition, 0 + x = x

The identity value is returned if the stream is empty, and you no longer need to deal with the **Optional** value.

Now suppose you have a stream of objects and want to form the sum of some

property, such as all lengths in a stream of strings.

**int result = words.reduce(0,**

**(total, word) -> total + word.length(),** // accumulator function

**(total1, total2) -> total1 + total2);** // adding totals for parallel computations

or

**words.mapToInt(String::length).sum();** //simpler and more efficient

2.9) Collecting results

When you are done with a stream, you often just want to look at the results instead of reducing them to a value. You can call the **iterator** method, which yields an old-fashioned **iterator** that you can use to visit the elements. Or you can call **toArray** and get an array of the stream elements.

**String[] result = words.toArray(String[]::new);** // pass the correct type in the array constructor, words.toArray() has type Object[]

Now suppose you want to collect the results in a **HashSet**. Use **collect**, it takes three arguments:

1. A **supplier**  to make new instances of the target object.
2. An **accumulator** that adds an element to the target. (**add**)
3. A **combiner** that merges two objects into one. (**addAll**)

The target object need not be a collection. It could be a **StringBuilder** or an object that tracks a count and a sum.

**HashSet<String>** **result = stream.collect(HashSet::new, HashSet::add, HashSet::addAll);**

In practice, you don’t have to do that because there is a convenient **Collector** interface for these three functions, and **Collectors** class with factory methods for common collectors.

**List<String> result = stream.collect(Collectors.toList());**

or

**Set<String> result = stream.collect(Collectors.toSet());**

If you want to control which kind of set you get:

**TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));**

To collect all strings in a stream by concatenating them:

**String result = stream.collect(Collectors.joining());**

With delimiter:

**String result = stream.collect(Collectors.joining(“, ”));**

If your stream contains objects other that strings, you need to first convert them to strings:

**String result = stream.map(Object::toString).collect(Collectors.joining(“, ”));**

If you want to reduce the stream to a sum, average, maximum, or minimum, then use **summarizing**. This method take a function that maps the stream objects to a number and yield a result of type **SummaryStatistics**.

**IntSummaryStatistics summary = words.collect(Collectors.summarizingInt(String::length));**

**double avarageWordLength = summary.getAverage();**

**double maxWordLength = summary.getMax();**

To print elements:

**stream.forEach(System.out::println);**

On a parallel stream, the elements can be traversed in arbitrary order. If you want to execute them in stream order, call **forEachOrdered** instead. Of course, you might give up most or all of the benefits of parallelism.

The **forEach** and **forEachOrdered** methods are terminal operations. You cannot use the stream again after calling them. If you want to continue using the stream, use **peek** instead.

2.10) Collecting into maps

Suppose you have a **Stream<Person>** and want to collect the elements into a map so that you can later look up people by their id.

**Map<Integer, String> idToName = people.collect(**

**Collectors.toMap(Person::getId,**

**Person::getName));**

or

**Map<Integer, Person> idToPerson = people.collect(**

**Collectors.toMap(Person::getId,**

**Function.identity()));**

If there is more that one element with the same key, the collector will throw an **IllegalStateException**. You can override that behavior by supplying a third function argument that determines the value for the key, given the existing and the new value.

We construct a map that contains, for each language in the available locales, as key its name in your default locale (such as “German”), and as value its localized name (such as “Deutch”).

**Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());**

**Map<String, String> languageNames = locales.collect(**

**Collectors.toMap( l -> l.getDisplayLanguage(),**

**l -> l.getDisplayLanguage(l),**

**(existingValue, newValue) -> existingValue));**

Suppose, we want to know all languages in a given country. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

**Map<String, Set<String>> countryLanguageSets = locales.collect(**

**Collectors.toMap(l -> l.getDisplayCountry(),**

**l -> Collections.singleton(l.getDisplayLanguage()),**

**(a,b) -> {**

**Set<String> r = new HashSet<>(a);**

**r.addAll(b),**

**return r;}));**

If you want a **TreeMap**, then you supply the constructor as the fourth argument.

**Map<Integer, Person> idToPerson = people.collect(**

**Collectors.toMap(Person::getId,**

**Function.identity(),**

**(existingValue, newValue) -> {throw new IllegalStateException();},**

**TreeMap::new));**

For each of the **toMap** methods, there is an equivalent **toConcurrentMap** method that yields a concurrent map.

2.11) Grouping and partitioning

Forming groups of values with the same characteristic is very common, and the **groupingBy** method supports it directly.

Group locales by country:

**Map<String, List<Locale>> countryToLocales = locales.collect(**

**Collectors.groupingBy(Locale::getCountry));**

The function Locale::getCountry is the *classifier function* of the grouping. You can now look up all locales for a given country code:

**List<Locale> swissLocales = countryToLocales.get(“CH”);** //yields [it\_CH, de\_CH, fr\_CH]

When the classifier function is a predicate function (returning a boolean value), the stream elements are partitioned into two lists: those where the function returns true and the complement. In this case, it is more efficient to use **partitioningBy** instead of **groupingBy**.

**Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(**

**Collectors.partitioningBy(l -> l.getLanguage().equals(“en”)));**

**List<Locale> englishLocales = englishAndOtherLocales.get(true);**

The **groupingBy** method yields a map whose values are lists. If you want to process those lists in some way, you supply a *“downstream collector”*.

If you want sets instead of lists:

**Map<String, Set<Locale>> countryToLocaleSet = locales.collect(**

**groupingBy(Locale::getCountry),**

**toSet());**

Several other collectors are provided for *downstream processing* of grouped elements:

***counting*** produces a count of the collected elements:

**Map<String, Long> countryToLocaleCounts = locales.collect(**

**groupingBy(Locale::getCountry,**

**counting());**

counts how many locales there are for each country.

***summing*** takes a function argument, applies the function to the downstream elements, and produces their sum:

**Map<String, Integer> stateToCityPopulation = cities.collect(**

**groupingBy(City::getState,**

**summingInt(City::getPopulation)));**

computes the sum of populations per state in a stream of cities.

***maxBy*** and ***minBy*** take a comparator and produce maximum and minimum of the downstream elements:

**Map<String, City> stateToLargestCity = cities.collect(**

**groupingBy(City::getState,**

**maxBy(Comparator.comparing(City::getPopulation))));**

produces the largest city per state.

***mapping*** applies a function to downstream results, and it requires yet another collector for processing its results:

**Map<String, Optional<String>> stateToLongestCityName = cities.collect(**

**groupingBy(City::getState,**

**mapping(City::getName,**

**maxBy(Comparator.comparing(String::length)))));**

Here, we group cities by state. Within each state, we produce the names of the cities and reduce by maximum length.

To gather a set of languages in a country:

**Map<String, Set<String>> countryToLanguages = locales.collect(**

**groupingBy(l -> l.getDisplayCountry(),**

**mapping(l -> l.getDisplayLanguage(),**

**toSet())));**

In this form, you don’t need to worry about combining the individual sets.

If the grouping or mapping function has return type **int**, **long** or **double**, you can collect elements into a *summary statistics object*:

**Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(**

**groupingBy(City::getState,**

**summarizingInt(City::getPopulation)));**

***reducing*** methods apply a general reduction to downstream elements.

**reducing(binaryOperator)** // identity is null

**reducing(identity, binaryOperator)**

**reducing(identity, mapper, binaryOperator)**

Get a comma-separated string of all city names in each state. We map each city to its name and then concatenate them.

**Map<String, String> stateToCityNames = cities.collect(**

**groupingBy(City::getState,**

**reducing(“”,**

**City::getName,**

**(s, t) -> s.length() == 0 ? t : s + “, ” + t)));**

As with Stream.reduce, Collectors.reducing is rarely necessary. In this case, you can achieve the same result more naturally as:

**Map<String, String> stateToCityNames = cities.collect(**

**groupingBy(City::getState),**

**mapping(City::getName,**

**joining(“, ”)));**

The downstream collectors can yield very convoluted expressions. You should only use them in connection with **groupingBy** or **partitioningBy** to process the downstream map values. Otherwise, simply apply methods such as **map**, **reduce**, **count**, **max** or **min** directly on streams.

2.12) Primitive type streams

The stream library has specialized types:

**IntStream** (use for **byte**, **short**, **char**, **boolean**)

**LongStream**

**DoubleStream** (use for **float**)

that store primitive values directly, without using wrappers.

**IntStream stream = IntStream.of(1,1,2,3,5);**

or

**stream = Arrays.stream(values, from, to);** // values is an int[] array

As with object streams, you can also use the static **generate** and **iterate** methods.

You can also generate integer ranges:

**IntStream zeroTo99 = IntStream.range(0, 100);** // Upper bound is excluded

**IntStream zeroTo100 = IntStream.rangeClosed(0, 100);** // Upper bound is included

When you have a stream of objects, you can transform it to a primitive type stream with the **mapToInt, mapToLong or mapToDouble** methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an **IntStream:**

**Stream<String> words = …;**

**IntStream lengths = words.mapToInt(String::length);**

To convert a primitive type stream to an object stream, use the **boxed** method:

**Stream<Integer> integers = Integer.range(0, 100).boxed();**

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

* The **toArray** methods return primitive type arrays.
* Methods that yield an optional result return an **OptionalInt, OptionalLong or OptionalDouble.** These classes are analogous to the **Optional** class, but they have methods **getAsInt, getAsLong** and **getAsDouble** instead of the **get** method.
* There are methods **sum, average, max** and **min** that return the sum, average, maximum and minimum. These methods are not defined for object streams.
* The **summaryStatistics** method yields an object of type **IntSummaryStatistics, LongSummaryStatistics or DoubleSummaryStatistics** that can simultaneously report the sum, average, maximum and minimum of the stream.

The **Random** class has methods **ints, longs** and **doubles** that return primitive type streams of random numbers.

2.13) Parallel streams

By default, stream operations create sequential streams, except for **Collection.parallelStream().** The **parallel** method converts any sequential stream into a parallel one.

**Stream<String> parallelWords = Stream.of(wordArray).parallel();**

As long as the stream is in parallel mode when the terminal method executes, all lazy intermediate stream operations will be parallelized. When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are *stateless* and can be executed in an arbitrary order.

Here is an example of something you cannot do:

**int[] shortWords = new int[12];**

**words.parallel().forEach(**

**s -> { if(s.length() < 12) shortWords[s.length()]++;** // Race condition

**}**

**);**

**System.out.println(Arrays.toString(shortWords));**

The function passed to **forEach** runs concurrently in multiple threads, updating a shared array.

It is your responsibility to ensure that any functions that you pass to parallel stream operations are threadsafe. In our example, you could use an array of **AtomicInteger** objects for the counters.

By calling the **Stream.unordered** method, you indicate that you are not interested in ordering. One operation that can benefit from this is **Stream.distinct**

You can also speed up the **limit** method by dropping ordering. If you just want any n elements from a stream and you don’t care which ones you get, call:

**Stream<T> sample = stream.parallel().unordered().limit(n);**

Merging maps is expensive. For that reason, the **Collectors.groupingByConcurrent** method uses a shared concurrent map.

**Map<String, List<String>> result = cities.parallel().collect(**

**Collectors.groupingByConcurrent(City::getState));**

It is very important that you don’t modify the collection that is backing a stream while carrying out a stream operation (even if the modification is threadsafe). Remember that streams don’t collect their own data—the data is always in a separate collection. If you were to modify that collection, the outcome of the stream operations would be undefined. The JDK documentation refers to this equirement as *noninterference*. It applies both to sequential and parallel streams.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point when the terminal operation executes.

**List<String> wordList = ...;**

**Stream<String> words = wordList.stream();**

**wordList.add("END");** // Ok

**long n = words.distinct().count();**

**Stream<String> words = wordList.stream();**

**words.forEach(s -> if (s.length() < 12) wordList.remove(s));** // interference

2.14) Functional interfaces

You have seen many operations whose argument is a function:

**Stream<String> longWords = words.filter(s -> s.length() >= 12);**

in Javadoc:

**Stream<T> filter(Predicate<? super T> predicate)**

**Predicate** is an interface with one nondefault method returning a **boolean** value.

Due to wildcards you can filter the stream (where T is **Employee**) with a Predicate<**Employee**>, a Predicate<**Person**> or a Predicate<**Object**>. This flexibility is particularly important for supplying method references. For example, you may want to use **Person::isAlive** to filter a **Stream<Employee>**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Functional interface** | **Parameter types** | **Return types** | **Description** |
| Supplier<T> | none | T | Supplies a value of type T |
| Consumer<T> | T | void | Consumes a value of type T |
| BiConsumer<T,U> | T,U | void | Consumes values of types T and U |
| Predicate<T> | T | boolean | A Boolean-valued function |
| ToIntFunction<T>  ToLongFunction<T>  ToDoubleFunction<T> | T | int  long  double | An int-, long- or double-valued function |
| IntFunction<R>  LongFunction<R>  DoubleFunction<R> | int  long  double | R | A function with argument of type int, long or double |
| Function<T, R> | T | R | A function with argument of type T |
| BiFunction<T,U,R> | T,U | R | A function with arguments of types T and U |
| UnaryOperator<T> | T | T | A unary operator on the type T |
| BinaryOperator<T> | T,T | T | A binary operator on the type T |