Muni Swamy Palla

Contents

1. LAMBDA EXPRESSIONS

- Lambda expressions are anonymous functions. Maintainable and readable.
- It is used mainly where one time use of the code.
- Lambda expressions can only appear in places where they will be assigned to a variable whose type is a functional interface.
- A functional interface has single abstract method.
  Ex: Runnable, Callable, Comparator, TimerTask
- 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.

1.1. Why Lambdas?

Lambda expressions are added in Java 8 and provide below functionalities.

- Enable to treat **functionality as a method argument**, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and **executed on demand**.

A “lambda expression” is a block of code that you can pass around so it can be executed later, once or multiple times.

```
Runnable r1 = new Runnable() {

        @Override

        public void run() {

        System.out.println("Running Thread 1");

       }

};

Thread t1 = new Thread(r1).start();
```

```
JAVA 8:

Runnable r1 = () -> {

        System.out.println("Running Thread 1");

}// took less code

Thread t1 = new Thread(r1).start();
```

```
for(String str : list)

{

        System.out.println(str);

}
```

```
JAVA 8:

list.forEach((str) -> {

        System.out.println(str);

});
```

```
Collections.sort(names, (String a, String b) -> {

    return a.compareTo(b);

});
```

If the parameter types of a lambda expression can be inferred (end) or the java compiler is aware of the parameter types so you can skip them as well. you can omit type.

```
Comparator<String> comp = Collections.sort(names, (a, b) -> {

    return a.compareTo(b);
}); // If list is List<String> str1 and str2 acts as a String.
```

For one line method bodies you can skip both the braces {} and the return keyword. But it gets even shorter:

```
Comparator<String> comp = Collections.sort(names, (a, b) -> a.compareTo(b));
```

We can't use stream() and filter() for map.

NOTE: You can add annotations or the final modifier to lambda parameters in the same way as for method parameters:

```
(final String name) -> ...

(@NonNull String name) -> ...
```

1.2 Functional Interfaces

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a functional interface.

NOTE: You can't even assign a lambda expression to a variable of type Object, Object is not a functional interface.

The Java API defines a number of very generic functional interfaces in the java.util.function package.

NOTE: You can tag any functional interface with the @FunctionalInterface annotation. This has two advantages.

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

It is not required to use the annotation. Any interface with a single abstract method is, by definition, a functional interface. But using the @FunctionalInterface annotation is a good idea.

Finally, note that 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. For example, the following would be an error:

```
Runnable sleeper = () -> { System.out.println("Zzz"); Thread.sleep(1000); };

// Error: Thread.sleep can throw a checked InterruptedException
```

Since the `Runnable.run` cannot throw any exception, this assignment is illegal. To fix the error, you have two choices. can catch the exception in the body of the lambda expression. Or assign the lambda to an interface whose single abstract method can throw the exception. For example,

Muni
the call method of the Callable interface can throw any exception. Therefore, you can assign the lambda to a Callable<Void> (if you add a statement return null).

Predicate

Predicates are Boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true

predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;

Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isNotEmpty = isEmpty.negate();
```

Predicate is functional interface and it has `test(T t)` method.

```
@FunctionalInterface

public interface Predicate<T>

boolean test(T t)
```

Functions

The most simple and general case of a lambda is a functional interface with a method that receives one value and returns another. Default methods can be used to chain multiple functions together (compose, andThen).

```
public interface Function<T, R> { … }
```

One of the usages of the Function type in the standard library is the Map.computeIfAbsent

```
Integer value = nameMap.computeIfAbsent("John", s -> s.length());
```

we may replace the lambda with a method reference that matches passed and returned value types.

```
Integer value = nameMap.computeIfAbsent("John", String::length);
```

```
Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"
```

Suppliers

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

```
Supplier<Person> personSupplier = Person::new;

personSupplier.get();   // new Person
```

Consumers

Consumers represent operations to be performed on a single input argument.

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));
```

Comparators

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

Key Selector Variant

The Comparator.comparing static function accepts a sort key Function and returns a Comparator for the type which contains the sort key:

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(

   Function<? super T,? extends U> keyExtractor)

Arrays.sort(employees, Comparator.comparing(Employee::getName));
```

Key Selector and Comparator Variant

There is another option that facilitates overriding the natural ordering of the sort key by providing the *Comparator* that creates a custom ordering for the sort key:

```
static <T,U> Comparator<T> comparing( Function<? super T,? extends U>
keyExtractor,

    Comparator<? super U> keyComparator)

Arrays.sort(employees, Comparator.comparing(Employee::getName, (s1, s2) ->
s2.compareTo(s1)));
```

Using Comparator.reversed

```
Arrays.sort(employees, Comparator.comparing(Employee::getName,
Comparator.reverseOrder()));
```

Using Comparator.comparingInt

There is also a function Comparator.comparingInt which does the same thing as Comparator.comparing, but it takes only int selectors.

```
Arrays.sort(employees, Comparator.comparingInt(Employee::getAge));
```

Similarly Comparator.comparingLong, Comparator.comparingDouble

This section covers functions *nullsFirst* and *nullsLast*, which consider *null* values in ordering and keep the *null* values at the beginning or end of the ordering sequence.

```
Arrays.sort(employees,
Comparator.nullsFirst(Comparator.comparing(Employee::getName)));
```

```
Arrays.sort(employees,
Comparator.nullsLast(Comparator.comparing(Employee::getName)));
```

Using Comparator.thenComparing

The *thenComparing* function lets you set up lexicographical ordering of values by provisioning multiple sort keys in a particular sequence.

```
Arrays.sort(employees,
Comparator.comparing(Employee::getAge).thenComparing(Employee::getName))
;
```

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");

Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0

comparator.reversed().compare(p1, p2);  // < 0
```

1.3 Method References

You can pass this method expression:

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

String::compareToIgnoreCase is the same as (x, y) -> x.compareToIgnoreCase(y).

Similarly, Math::pow is equivalent to `(x, y) -> Math.pow(x, y)`.

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

- *Class::instanceMethod*

1.4 Constructor References

Constructor references are just like method references, except that the name of the method is new. For example, Button::new is a reference to a Button constructor. Suppose you have a list of strings. Then you can turn it into an array of buttons, by calling the constructor on each of the strings.

```
List<String> labels = ...;

Stream<Button> stream =
labels.stream().map(Button::new).collect(Collectors.toList());
```

You can form constructor references with array types. For example, int[]::new is a constructor reference with one parameter: the length of the array. It 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]. That is a problem for library authors. For example, suppose we want to have an array of buttons. The Stream interface has a toArray method that returns an Object array:

```
Object[] buttons = stream.toArray();
```

But that is unsatisfactory. The user wants an array of buttons, not objects. The stream library solves that problem with constructor references. Pass Button[]::new to the toArray method:

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

The toArray method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

1.5 Variable Scope

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.

We can read final local variables from the outer scope of lambda expressions:

```
int num = 1;

Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);
```

```
public static void repeatMessage(String text, int count) {
  Runnable r = () -> {
      while (count > 0) {
      count--; // Error: Can’t mutate captured variable
      }
  };
}
```

There is a reason for this restriction. Mutating variables in a lambda expression is not thread safe.

In contrast to local variables, we have both read and write access to instance fields and static variables from within lambda expressions. This behaviour is well known from anonymous objects.

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;
    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
             outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

NOTE: Inner classes can also capture values from an enclosing scope. Before Java 8, inner classes were only allowed to access final local variables. This rule has now been relaxed to match that for lambda expressions. An inner class can access any effectively final local variable that is, any variable whose value does not change.

```
List<Path> matches = new ArrayList<>();
for (Path p : files)
new Thread(() -> { if (p has some property) matches.add(p); }).start();
// Legal to mutate matches, but unsafe
```

matches always refers to the same ArrayList object. However, the object is mutated, and that is not thread safe. If multiple threads call add, the result is unpredictable.

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
```

1.6 Default Methods

Java 8 enables us to add non-abstract method implementation to interfaces by utilizing the default keyword.

```
interface Formula {

    double calculate(int a);

    default double sqrt(int a) {

        return Math.sqrt(a);

    }

}
```

Concrete classes have to implement the only abstract method calculate(). The default method sqrt() can be used out of the box.

```
Formula formula = (a) -> {

        return a + 2;

};

formula.calculate(2);     // 4.0

formula.sqrt(16);         // 4.0
```

If the Collection interface gets new methods, such as forEach, then every program that defines its own class implementing Collection will break until it, too, implements that method. That is simply unacceptable in Java. The Java designers decided to solve this problem once and for all by allowing interface methods with concrete implementations (called default methods). NOTE: In Java 8, the forEach method has been added to the Iterable interface, a super interface of Collection.

Interfaces clash. If a super interface provides a default method, and another interface supplies a method with the same name and parameter types (default or not), then you must resolve the conflict by overriding that method.

```
interface Named {
    default String getName() { return getClass().getName() + "_" + hashCode(); }
}
interface Person {
    default String getName() { return getClass().getName() + "_" + hashCode(); }
}

class Student implements Person, Named {
    @Override
    public String getName() {
        return Person.super.getName();
    }
}
```

The class inherits two getName methods provided by the Person and Named interfaces. Rather than choosing one over the other, the Java compiler reports an error and leaves it up to the programmer to resolve the ambiguity. Simply provide a getName method in the Student class.

Now assume that the Named interface does not provide a default implementation for getName:

```
interface Named {
    String getName();
}
```

If at least one interface provides an implementation, the compiler reports an error, and the programmer must resolve the ambiguity.

i.e. in Student class we need to override getName() method to resolve compilation issue.

We just discussed name clashes between two interfaces. Now consider a class that extends a superclass and implements an interface, inheriting the same method from both. For example, suppose that Person is a class and Student is defined as

```
interface Named {
    default String getName() { return getClass().getName() + "_" + hashCode(); }
}
class Person {
    public String getName() { return getClass().getName() + "_" + hashCode(); }
}
class Student extends Person implements Named { // no compilation error Super class
is preferred.
}
```

In that case, only the superclass method matters, and any default method from the interface is simply ignored.

The “class wins” rule ensures compatibility with Java 7. If you add default methods to an interface, it has no effect on code that worked before there were default methods.

Remember the formula example from the first section? Interface Formula defines a default method sqrt() which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods **cannot** be accessed from within lambda expressions. The following code does not compile:

```
Formula formula = (a) -> sqrt(a * 100);
```

Note: Adding too many default methods to the interface is not a very good architectural decision.

1.7 Static Methods in Interfaces

As of Java 8, you are allowed to add static methods to interfaces. Have a look at the Paths class. It only has a couple of factory methods. Subclass can’t override static method, any class can use static method by InterfaceName.staticMethodName. You can construct a path from a sequence of strings, such as Paths.get("jdk1.8.0", "jre","bin"). In Java 8, one could have added this method to the Path interface:

```
public interface Path {

      public static Path get(String first, String... more) {

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

      }

}
```

In Java 8, static methods have been added to quite a few interfaces. For example, the Comparator interface has a very useful static comparing method that accepts a “key extraction” function and yields a comparator that compares the extracted keys. To compare Person objects by name, use Comparator.comparing(Person::name)

https://www.baeldung.com/java-8-comparator-comparing

2. STREAM API

- Iterators simply 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.
- A stream does not 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. For example, if you only ask for the first five long words instead of counting them all, then the filter method will stop filtering after the fifth match.

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.

NOTE: Stream operations are not executed on the elements in the order in which they are invoked on the streams. In our example, nothing happens until count is called. When the count method asks for the first element, then the filter method starts requesting elements, until it finds one that has length > 12.

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.

A java.util.Stream represents a sequence of elements on which one or more operations can be performed. Stream operations are either intermediate or terminal. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a `java.util.Collection` like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

```
list.stream().filter(p -> p > 4).forEach(p -> System.out.print(p));

list.stream().filter(p -> p > 3).forEach(System.out::println);

Stream<String> stream = Stream.of(new String[]{"muni","swamy","palla"});

stream.forEach(str -> System.out.println(str));

Stream<String> stream = Arrays.stream(new String[]{"muni","swamy","palla"});

stream.forEach(str -> System.out.println(str));

long size = list.stream().count();

long sum = list.stream().mapToInt(p -> p).sum();
```

Parallel stream useful when you are doing aggregate operations. Parallel will works divide concur algorithm it will split the data into different units and finally collects the data shows the result.

```
long size = list.stream().parallel().count();

long sum = list.stream().parallel().mapToInt(p -> p.getAge()).sum();
```

2.1 Stream Creation

Java 8 added a new stream() method to the Collection interface. If you have an array, use the static `Stream.of` method instead.

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

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

The static Files.lines method returns a Stream of all lines in a file. The Stream interface has AutoCloseable as a super interface. When the close method is called on the stream, the underlying file is also closed. To make sure that this happens, it is best to use the Java 7 try-with-resources statement:

```
try (Stream<String> lines = Files.lines(path)) {

Do something with lines

}
```

The stream, and the underlying file with it, will be closed when the try block exits normally or through an exception.

https://howtodoinjava.com/java8/read-file-line-by-line-in-java-8-streams-of-lines-example/

https://www.mkyong.com/java8/java-8-stream-read-a-file-line-by-line/

2.2 The filter, map, flatMap, collect, toArray, peek Methods

forEach(`Consumer<? super T> action`**)**

is a terminal operation, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used.

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

It takes a predicate as an argument and returns an Optional object.

```
public boolean priceIsInRange2(Modem modem2) {

     return Optional.ofNullable(modem2)

       .map(Modem::getPrice)

       .filter(p -> p >= 10 && p <= 15)

       .isPresent();

 }
```

map(`Function<? super T, ? extends R> mapper`**)**

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type. The map() is used to transform one object into other by applying a function.

The `map(Function mapper)` method takes a Function, technically speaking an object of `java.util.function.Function` interface. This function is then applied to each element of Stream to convert into a type you want.

We can use the **map()** method to collect the stream of even numbers and then convert each number to it's square before collecting it to a new list.

```
list.stream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * n)
            .collect(Collectors.toList());
```

The following example converts the stream of Integers into the stream of Employees:

```
Integer[] empIds = { 1, 2, 3 };
List<Employee> employees = Stream.of(empIds)
      .map(employeeRepository::findById)
      .collect(Collectors.toList());
```

Here, we obtain an Integer stream of employee ids from an array. Each Integer is passed to the function `employeeRepository::findById()` – which returns the corresponding Employee object; this effectively forms an Employee stream.

The map API returns the result of the computation wrapped inside Optional. We then have to call an appropriate API on the returned Optional to retrieve its value.

```
String name = "baeldung";
Optional<String> nameOptional = Optional.of(name);
int len = nameOptional
             .map(String::length())
             .orElse(0);
```

We can chain map and filter together to do something more powerful.

Muni

```
String password = " password ";

Optional<String> passOpt = Optional.of(password);

boolean correctPassword = passOpt.map(String::trim)

                        .filter(pass -> pass.equals("password"))

                        .isPresent();
```

Stream API also provides methods like `mapToDouble()`, `mapToInt()`, and `mapToLong()` which returns `DoubleStream`, `IntStream` and `LongStream`, which are specialized stream for double, int and long data types.

Collect(`Collector<? super T, A, R> collector`**):**

It's one of the common ways to get stuff out of the stream once we are done with all the processing.

```
list.stream().map(Employee::getName).collect(Collectors.toList());

list.stream().map(Employee::getName).collect(Collectors.toList());
```

Collectors:

```
givenList.stream().collect(toList());

givenList.stream().collect(toSet());

givenList.stream().collect(toCollection(LinkedList::new));

givenList.stream().collect(toMap(Function.identity(), String::length));

givenList.stream().collect(collectingAndThen(toList(), ImmutableList::copyOf))

givenList.stream().collect(joining());

givenList.stream().collect(counting());

givenList.stream().collect(maxBy(Comparator.naturalOrder()));

givenList.stream().collect(groupingBy(String::length, toSet()));
```

If you want to write your Collector implementation, you need to implement Collector interface and specify its three generic parameters:

```
public interface Collector<T, A, R> {...}
```

https://www.baeldung.com/java-8-collectors

toArray()

We saw how we used `collect()` to get data out of the stream. If we need to get an array out of the stream, we can simply use `toArray()`.

```
empList.stream().toArray(Employee[]::new)
```

The syntax Employee[]::new creates an empty array of Employee which is then filled with elements from the stream.

flatMap`(Function<? super T, ? extends Stream<? extends R>> mapper)`**:**

A stream can hold complex data structures like Stream<List<String>>. In cases like this, `flatMap()` helps us to flatten the data structure to simplify further operations.

we need `flatMap()` to do the following conversion :

```
Stream<String[]>       -> flatMap ->  Stream<String>

Stream<Set<String>>    -> flatMap ->  Stream<String>

Stream<List<String>>   -> flatMap ->  Stream<String>

Stream<List<Object>>   -> flatMap ->  Stream<Object>
```

How `flatMap()` works :

```
{ {1,2}, {3,4}, {5,6} } -> flatMap -> {1,2,3,4,5,6}


{ {'a','b'}, {'c','d'}, {'e','f'} } -> flatMap -> {'a','b','c','d','e','f'}
```

The call `stream.limit(n)` returns a new stream that ends after n elements.

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

peek()

sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.

peek() can be useful in situations like this. Simply put, it performs the specified operation on each element of the stream and returns a new stream which can be used further. peek() is an intermediate operation.

```
empList.stream()

       .peek(System.out::println)

       .collect(Collectors.toList());
```

2.3 Stateful Transformations

The stream transformations of the preceding sections were *stateless*. When an element is retrieved from a filtered or mapped stream, the answer does not depend on the previous elements. There are also a few stateful transformations. For example, the `distinct` method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The stream must obviously remember the elements that it has already seen.

Sorted

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

```
nameList.stream().sorted().filter((s) -> s.startsWith("a")).forEach(System.out::println);
```

Keep in mind that sorted does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of nameList is untouched:

Here, we sort strings so that the longest ones come first:

`sorted()`: It sorts the elements of stream using natural ordering. The element class must implement Comparable interface.

`sorted(Comparator<? super T> comparator)`: Here we create an instance of `Comparator` using lambda expression. We can sort the stream elements in ascending and descending order.

The following line of code will sort the list in natural ordering.

```
list.stream().sorted()
```

To reverse the natural ordering `Comparator` provides `reverseOrder()` method. We use it as follows.

```
list.stream().sorted(Comparator.reverseOrder())
```

The following line of code is using `Comparator` to sort the list.

```
list.stream().sorted(Comparator.comparing(Student::getAge))
```

To reverse the ordering, `Comparator` provides `reversed()` method. We use this method as follows.

```
list.stream().sorted(Comparator.comparing(Student::getAge).reversed())
```

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

NOTE: The `Collections.sort()` method sorts a collection in place, whereas `Stream.sorted()` returns new sorted stream.

To collect a stream into a list or set, you can simply call

```
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, use the following call instead:

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

Suppose you want to collect all strings in a stream by concatenating them. You can call.

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, average, maximum, or minimum, then use one of the methods summarizing(Int|Long|Double). These methods take a function that maps the stream objects to a number and yield a result of type (Int|Long|Double)SummaryStatistics, with methods for obtaining the sum, average, maximum, and minimum.

```
IntSummaryStatistics summary = words.collect(
Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

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.

https://www.concretepage.com/java/jdk-8/java-8-stream-sorted-example

Comparators:

The Comparator interface has a number of useful new methods, taking advantage of the fact that interfaces can now have concrete methods.

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

You can chain comparators with the thenComparing method for breaking ties. For example,

```
Arrays.sort(people,
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirst
Name));
```

If two people have the same last name, then the second comparator is used.

here we sort people by the length of their names

```
Arrays.sort(people, Comparator.comparing(Person::getName, (s, t) ->
Integer.compare(s.length(), t.length())));
```

Moreover, both the comparing and thenComparing methods have variants that avoid boxing of int, long, or double values. An easier way of producing the preceding operation would be

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

Method:

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)
```

This method returns a null-friendly comparator that considers null to be less than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order.

```
String[] strings = {"banana", "orange", null, "apple", null};

Arrays.sort(strings, Comparator.nullsFirst(Comparator.naturalOrder()));
```

suppose getMiddleName returns a null when a person has no middle name. Then you can use

```
Arrays.sort(people, Comparator.comparing(Person::getMiddleName,
Comparator.nullsFirst(naturalOrder())));
```

https://www.logicbig.com/tutorials/core-java-tutorial.html

Infinite Streams

There are two ways to generate infinite streams:

generate()

```
@Test

public void whenGenerateStream_thenGetInfiniteStream() {

    Stream.generate(Math::random)

      .limit(5)

      .forEach(System.out::println);

}
```

Here, we pass Math::random() as a Supplier, which returns the next random number.

With infinite streams, we need to provide a condition to eventually terminate the processing. One common way of doing this is using limit(). Please note that the Supplier passed to generate() could be stateful and such stream may not produce the same result when used in parallel.

iterate()

iterate() takes two parameters: an initial value, called seed element and a function which generates next element using the previous value. iterate(), by design, is stateful and hence may not be useful in parallel streams:

```
@Test

public void whenIterateStream_thenGetInfiniteStream() {

    Stream<Integer> evenNumStream = Stream.iterate(2, i -> i * 2);

    List<Integer> collect = evenNumStream

      .limit(5)

      .collect(Collectors.toList());

    assertEquals(collect, Arrays.asList(2, 4, 8, 16, 32));

}
```

Here, we pass 2 as the seed value, which becomes the first element of our stream. This value is passed as input to the lambda, which returns 4. This value, in turn, is passed as input in the next iteration.

This continues until we generate the number of elements specified by limit()which acts as the terminating condition.

2.4 Simple Reductions max, min, findFirst, findAny, anyMatch, allMatch, noneMatch

Reductions are terminal operations. `count, max, min` these methods return an Optional<T> value that either wraps the answer or indicates that there is none.

min() and max():

As the name suggests, `min()` and `max()` return the minimum and maximum element in the stream respectively, based on a comparator. They return an Optional since a result may or may not exist.

```
numbers.stream()
        .mapToInt(v -> v)
        .max().orElseThrow(NoSuchElementException::new);
```

```
empList.stream()
        .min(Comparator.comparing(Employee::getSalary))
        .orElse(null);
```

findFirst():

The `findFirst()` returns the first value in a nonempty collection. It is often useful when combined with filter.

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

findAny():

If you are okay with any match, not just the first one, then use the `findAny()` method. This is effective when you parallelize the stream since the first match in any of the examined segments will complete the computation.

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

anyMatch():

If you just want to know there is a match, use `anyMatch()`. That method takes a predicate argument, so you won't need to use filter.

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

allMatch() and noneMatch():

There are also methods `allMatch()` and `noneMatch()` that return true if all or no elements match a predicate. These methods always examine the entire stream, but they still benefit from being run in parallel.

distinct()

distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not:

```
list.stream().distinct().collect(Collectors.toList());
```

2.5 The Optional Type

An Optional<T> object is either a wrapper for an object of type T or for no object. The get method gets the wrapped element if it exists, or throws a NoSuchElementException if it doesn't. Therefore,

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

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

If you write a method that creates an Optional object, there are several static methods for that purpose. Either create an Optional.of(result) or Optional.empty(). For example,

```
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.

Optional are not functional interfaces, but nifty utilities to prevent NullPointerException. Instead of returning null you return an Optional in Java 8.

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true

optional.get();                 // "bam"

optional.orElse("fallback");    // "bam"

optional.orElseThrow(NoSuchElementException::new);

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
```

To create an empty Optional object:

```
Optional<String> empty = Optional.empty();

empty.isPresent(); //false
```

We can also create an Optional object with the static of() method:

```
String name = "baeldung";

Optional<String> opt = Optional.of(name);

if(opt.isPresent()) {

        System.out.println(opt.get()); // baeldung

}
```

The argument passed to the of() method cannot be null, otherwise, we will get a NullPointerException:

```
String name = null;

Optional<String> opt = Optional.of(name); // NullPointerException
```

But, in case we expect some null values for the passed in argument, we can use the ofNullable API:

```
String name = "baeldung";

Optional<String> opt = Optional.ofNullable(name);

if(opt.isPresent()) {

        System.out.println(opt.get()); // baeldung

}
```

In this way, if we pass in a null reference, it does not throw an exception but rather returns an empty Optional object as though we created it with the Optional.empty API:

```
String name = null;

Optional<String> opt = Optional.ofNullable(name);

opt.isPresent(); // false
```

Conditional Action With ifPresent():

Before Optional, we would do something like this:

```
if(name != null){

    System.out.println(name.length);
```

Muni

```
}
```

One could easily forget to perform the null checks in some part of the code. This can result in a NullPointerException at runtime.

```
Optional<String> opt = Optional.of("baeldung");

opt.ifPresent(name -> System.out.println(name.length()));
```

Default Value With orElse:

It takes one parameter which acts as a default value.

```
String nullName = null;

String name = Optional.ofNullable(nullName).orElse("john");
```

Default Value With orElseGet:

The orElseGet API is similar to orElse. However, instead of taking a value to return if the Optional value is not present, it takes a supplier functional interface which is invoked and returns the value of the invocation:

```
String nullName = null;

String name = Optional.ofNullable(nullName).orElseGet(() -> "john");
```

Difference Between orElse and orElseGet:

```
public String getMyDefault() {

    System.out.println("Getting Default Value");

    return "Default Value";

}

String text;

System.out.println("Using orElseGet:");

String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Default Value", defaultText); // true

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Default Value", defaultText); // true
```

When the wrapped value is not present, then both orElse and orElseGet APIs work exactly the same way.

```
String text = "Text present";

System.out.println("Using orElseGet:");

String defaultText = Optional.ofNullable(text).orElseGet(this::getMyDefault);

assertEquals("Text present", defaultText);//true getMyDefault() will not call

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Text present", defaultText); // ture getMyDefault will call
```

Notice that when using orElseGet to retrieve the wrapped value, the getMyDefault API is not even invoked since the contained value is present.

However, when using orElse, whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.

Exceptions with orElseThrow:

Instead of returning a default value when the wrapped value is not present, it throws an exception:

```
String nullName = null;

String name =
Optional.ofNullable(nullName).orElseThrow(IllegalArgumentException::new);
```

Method references in Java 8 come in handy here, to pass in the exception constructor.

Returning Value with get():

```
Optional<String> opt = Optional.of("baeldung");

String name = opt.get(); // baeldung
```

The get() API method can only return a value if the wrapped object is not null, otherwise, it throws a no such element exception.

```
String unit = person.getAddress().getCity().getStreet().getUnit();
```

This is very unsafe because any of object in the chain can be null and if you check null for every object then the code will get cluttered and you will lose the readability. Thankfully, you can use the flatMap() method of the Optional class to make it safe and still keep it readable as shown in the following example:

```
String unit= person.flatMap(Person::getAddress)
                    .flatMap(Address::getCity)
                    .flatmap(City::getStreet)
                    .map(Street::getUnit)
                    .orElse("UNKNOWN");
```

The first flatMap ensures that an `Optional<Address>` is returned instead of an `Optional<Optional<Address>>`, and the second flatMap does same to return an `Optional<City>` and so on.

2.6 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. The Collectors.toMap method has two function arguments that produce the map keys and values. For example,

```
Map<Integer, String> idToName = people.collect(Collectors.toMap(Person::getId,
Person::getName));
```

In the common case that the values should be the actual elements, use Function. identity() for the second function.

```
Map<Integer, Person> idToPerson =
people.collect(Collectors.toMap(Person::getId, Function.identity()));
```

If there is more than one element with the same key, the collector will throw an `IllegalStateException.` You can override that behaviour by supplying a third function argument that determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

```
Map<Integer, Person> idToPerson =
people.collect(Collectors.toMap(Person::getId, Function.identity(), (oldValue,
newValue) -> oldValue));
```

Both:

```
Map<Integer, Person> idToPerson =
people.collect(Collectors.toMap(Person::getId, Function.identity(), (oldValue,
newValue) -> {
     List< Person > r = new ArrayList<>(oldValue);

     r.addAll(newValue);

     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(), (oldValue,
newValue) -> { throw new IllegalStateException(); }, TreeMap::new));
```

NOTE: For each of the `toMap` methods, there is an equivalent `toConcurrentMap` method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps, but of course, you give up ordering.

2.7 Grouping and Partitioning

Java 8 now directly allows you to do GROUP BY in Java by using `Collectors.groupingBy()` method. GROUP BY is a very useful aggregate operation from SQL. It allows you to group records on certain criteria. How do you group by in Java? For example, suppose you have a list of Persons, How do you group persons by their city e.g. London, Paris or Tokyo?

```
Map<BlogPostType, List<BlogPost>> postsPerType =
posts.stream().collect(Collectors.groupingBy(BlogPost::getType));
```

Concurrent Grouping By Collector:

Similar to the groupingBy, there is the groupingByConcurrent collector, which leverages multi-core architectures. This collector has three overloaded methods that take exactly the same arguments as the respective overloaded methods of the groupingBy collector. The return type of the groupingByConcurrent collector, however, must be an instance of the ConcurrentHashMap class or a subclass of it.

To do a grouping operation concurrently, the stream needs to be parallel:

```
ConcurrentMap<BlogPostType, List<BlogPost>> postsPerType = posts.parallelStream()
  .collect(groupingByConcurrent(BlogPost::getType));
```

https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-8

mapping()

Sometimes we might need to group data into a type other than the element type. we can use mapping().

```
items.stream().collect(Collectors.
        groupingBy(Item::getPrice, Collectors.mapping(Item::getPrice,
Collectors.toSet())));
```

Removing entry in Map:

you can simply call `removeIf()` method which takes a Predicate. Just pass the condition you want to check for removing values from the map.

```
map.entrySet().removeIf(e -> "muni".equals(e.getKey()));
```

Primitive Type Streams

So far, we have collected integers in a Stream<Integer>, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types `double, float, long, short, char, byte, and boolean.` The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short, char, byte, and boolean`, use an `IntStream`, and for float, use a `DoubleStream.` The library designers didn't think it was worth adding another five stream types. To create an `IntStream`, you can call the `IntStream.of` and `Arrays.stream` methods:

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

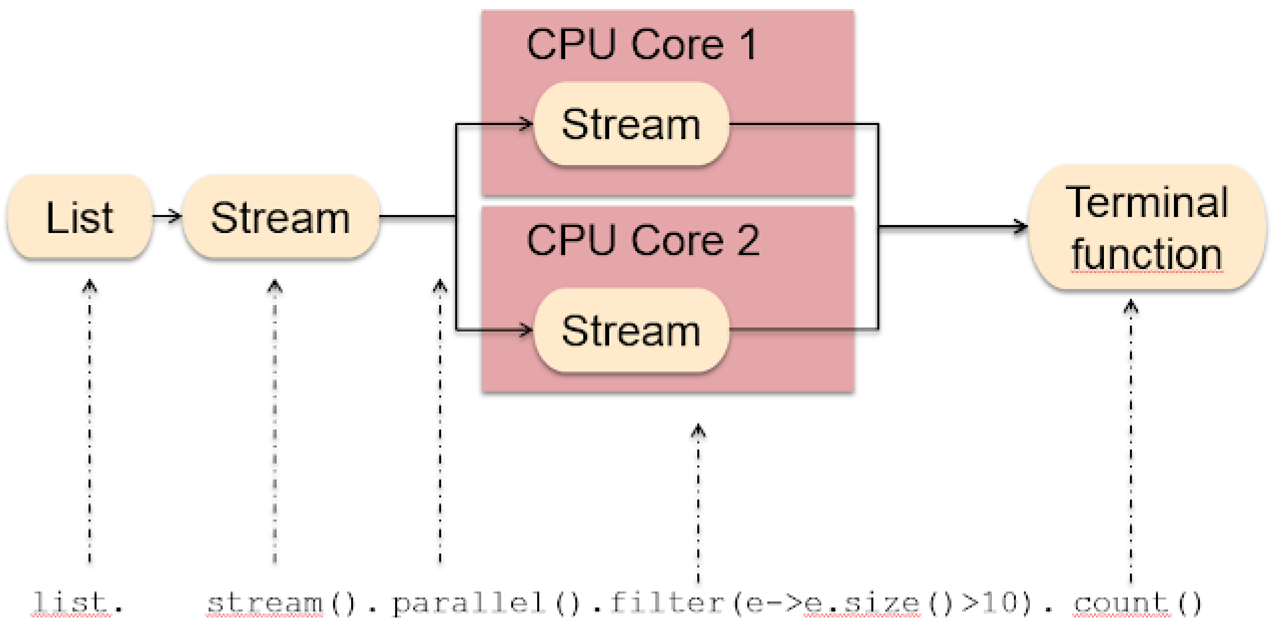
When you have a stream of objects, you can transform it to a primitive type stream with the `mapToInt, mapToLong, or mapToDouble` methods.

2.8 Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. As long as the stream is in parallel mode when the terminal method executes, all lazy intermediate stream operations will be parallelized.

we need to be aware of few things while using parallel streams:

1. We need to ensure that the code is thread-safe. Special care needs to be taken if the operations performed in parallel modifies shared data.
2. We should not use parallel streams if the order in which operations are performed or the order returned in the output stream matters. For example operations like findFirst() may generate the different result in case of parallel streams.
3. Also, we should ensure that it is worth making the code execute in parallel. Understanding the performance characteristics of the operation in particular, but also of the system as a whole – is naturally very important here.



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

words.parallel().forEach(

s -> { if (s.length() < 12) shortWords[s.length()]++; });

// Error—race condition!

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

This is very, very bad code. The function passed to forEach runs concurrently in multiple threads, updating a shared array. That's a classic race condition. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run, each of them wrong.

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

Ordering does not preclude parallelization.

CAUTION: 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 thread safe). 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 requirement 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. For example, the following is correct:

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

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

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

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

But this code is not:

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

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

// Error—interference
```

2.9 Functional Interfaces

Functional Interface	Parameter Types	Return Type	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

3. LAMBDAS

- The main reason for using a lambda expression is to defer the execution of the code until an appropriate time.
- Think what should happen when you work with a lambda expression that throws an exception.
- When working with generic functional interfaces, use ? super wildcards for argument types, ? extends wildcards for return types.

There are many reasons for executing code later, such as

- Running the code in a separate thread.
- Running the code multiple times.
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary.

```
logger.info("x: " + x + ", y: " + y);
```

What happens if the log level is set to suppress INFO messages? The message string is computed and passed to the info method, which then decides to throw it away. Wouldn't it be nicer if the string concatenation only happened when necessary? Running code only when necessary is a use case for lambdas. The standard idiom is to wrap the code in a no-arg lambda:

```
() -> "x: " + x + ", y: " + y
```

Now we need to write a method that

1. Accepts the lambda
2. Checks whether it should be called
3. Calls it when necessary

```
public static void info(Logger logger, Supplier<String> message) {
```

```
if (logger.isLoggable(Level.INFO))

logger.info(message.get());

}
```

now have variants that accept a Supplier<String>.You can directly call

```
logger.info(() -> "x: " + x + ", y:" + y)
```

When an exception is thrown in a lambda expression, it is propagated to the caller.

One of the unhappy consequences of type erasure is that you cannot construct a generic array at runtime. For example, the toArray() method of Collection<T> and Stream<T> cannot call T[] result = new T[n]. Therefore, these methods return Object[] arrays. In the past, the solution was to provide a second method that accepts an array.

```
String[] result = words.toArray(String[]::new);
```

The general rule is that you use super for argument types, extends for return types.

4. THE NEW DATE AND TIME API

- All java.time objects are immutable.
- An Instant is a point on the time line (similar to a Date).
- In Java time, each day has exactly 86,400 seconds (i.e., no leap seconds).
- A Duration is the difference between two instants.
- LocalDateTime has no time zone information.
- ZonedDateTime is a point in time in a given time zone (similar to GregorianCalendar).
- Use a Period, not a Duration, when advancing zoned time, in order to account for daylight savings time changes.
- Use DateTimeFormatter to format and parse dates and times.

The static method call Instant.now() gives the current instant. To find out the difference between two instants, use the static method Duration.between.

```
Instant start = Instant.now();

Instant end = Instant.now();
```

```
Duration timeElapsed = Duration.between(start, end);

long millis = timeElapsed.toMillis();
```

You can get the length of a Duration in conventional units by calling toNanos, toMillis, toSeconds, toMinutes, toHours, or toDays.

NOTE: The Instant and Duration classes are immutable, and all methods, such as multipliedBy or minus, return a new instance.

```
LocalDate today = LocalDate.now(); // Today's date

LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);

alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);

// Uses the Month enumeration
```

A LocalTime represents a time of day, such as 15:30:00. You can create an instance with the now or of methods:

```
LocalTime rightNow = LocalTime.now();

LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)
```

There is a LocalDateTime class, representing a date and time. That class is suitable for storing points in time in a fixed time zone. However, if you need to make calculations that span the daylight savings time, or if you need to deal with users in different time zones, you should use the ZonedDateTime.

```
ZonedDateTime skipped = ZonedDateTime.of(

LocalDate.of(2013, 3, 31),

LocalTime.of(2, 30),

ZoneId.of("Europe/Berlin"));
```

date.toInstant() and cal.toZonedDateTime() to covert old java.util.Date, java.util. GregorianCalendar to new.

5. CONCURRENCY ENHANCEMENTS

- Updating atomic variables has become simpler with the updateAndGet/accumulateAndGet methods.
- LongAccumulator/DoubleAccumulator are more efficient than AtomicLong/AtomicDouble under high contention.

- Updating entries in a ConcurrentHashMap has become simpler with the compute and merge methods.
- The Arrays class has methods for parallel sorting, filling, and prefix operations.

3.1 Atomic Values

java.util.concurrent.atomic package provided classes for lock-free mutation of variables. For example, you can safely generate a sequence of numbers like this:

```
public static AtomicLong nextNumber = new AtomicLong();

// In some thread . . .

long id = nextNumber.incrementAndGet();
```

The incrementAndGet method atomically increments the AtomicLong and returns the post-increment value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

The LongAccumulator generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation, as well as its neutral element. To incorporate new values, call accumulate. Call get to obtain the current value. The following has the same effect as a LongAdder:

```
LongAccumulator adder = new LongAccumulator(Long::sum, 0);

// In some thread . . .

adder.accumulate(value);
```

If you anticipate high contention, you should simply use a LongAdder instead of an AtomicLong. The method names are slightly different. Call increment to increment a counter or add to add a quantity, and sum to retrieve the total.

```
LongAdder adder = new LongAdder();

adder.increment();
```

There are also DoubleAdder and DoubleAccumulator that work in the same way, except with double values.

ConcurrentHashMap

The method putIfAbsent() puts a new value into the map only if no value exists for the given key. At least for the ConcurrentHashMap implementation of this method is thread-safe just like put()so you don't have to synchronize when accessing the map concurrently from different threads:

```
map.putIfAbsent("c3", "p1");
```

The method getOrDefault() returns the value for the given key. In case no entry exists for this key the passed default value is returned:

```
String value = map.getOrDefault("hi", "there");
```

6. MISCELLANEOUS GOODIES

- Joining strings with a delimiter is finally easy: String.join(", ", a, b, c) instead of a + ", " + b + ", " + c.
- Integer types now support unsigned arithmetic.
- There are new mutators in Collection (removeIf) and List (replaceAll, sort).
- Files.lines lazily reads a stream of lines.
- Files.list lazily lists the entries of a directory, and Files.walk traverses them recursively.
- There is finally official support for Base64 encoding.
- Annotations can now be repeated and applied to type uses.

```
String joined = String.join("/", "usr", "local", "bin"); // "usr/local/bin"
```

Integer types now support unsigned arithmetic. For example, instead of having a Byte represent the range from –128 to 127, you can call the static method Byte.toUnsignedInt(b) and get a value between 0 and 255. In general, with unsigned numbers, you lose the negative values and get twice the range of positive values. The Byte and Short classes have methods toUnsignedInt, and Byte, Short, and Integer have methods toUnsignedLong.

Methods Added to Collection Classes and Interface in Java 8

Class/Interface	New Methods
Iterable	forEach
Collection	removeIf
List	replaceAll, sort
Map	forEach, replace, replaceAll, remove(key, value) (removes only if key mapped to value), putIfAbsent, compute, computeIf(Absent | Present), merge
Iterator	forEachRemaining
BitSet	stream

Working with Files

Read File:

To read the lines of a file lazily, use the Files.lines method. It yields a stream of strings, one per line of input:

```
Stream<String> lines = Files.lines(Paths.get(fileName));

Optional<String> passwordEntry = lines.filter(s ->
s.contains("password")).findFirst();
```

As soon as the first line containing password is found, no further lines are read from the underlying file. The Files.lines method defaults to UTF-8. You can specify other encodings by supplying a Charset argument.

The easiest way to make sure the file is indeed closed is to use a Java 7 try-with-resources block:

```
try (Stream<String> lines = Files.lines(path)) {

Optional<String> passwordEntry = lines.filter(s -> s.contains("password")).findFirst();

}
```

When stream is closed, underlying file also be closed. Because Stream interface extends AutoCloseable.

NOTE: If you want to be notified when the stream is closed, you can attach an onClose handler. Here is how you can verify that closing filteredLines actually closes the underlying stream:

```
try (Stream<String> filteredLines = Files.lines(path).onClose(() ->
System.out.println("Closing")).filter(s -> s.contains("password"))) { ... }
```

If an IOException occurs as the stream fetches the lines, that exception is wrapped into an UncheckedIOException which is thrown out of the stream operation. (This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.)

The static Files.list method returns a Stream<Path> that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries. Since reading a directory involves a system resource that needs to be closed, you should use a try block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
...
}
```

The list method does not enter subdirectories. To process all descendants of a directory, use the Files.walk method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
// Contains all descendants, visited in depth-first order
}
```

You can limit the depth of the tree that you want to visit by calling Files.walk(pathToRoot, depth).

File Write

```
String[] words = {"hello", "refer", "world", "level"};
try (PrintWriter pw = new PrintWriter(Files.newBufferedWriter(Paths.get(fileName)))) {
        Stream.of(words).forEach(pw::println);
}
```

Here we use forEach() to write each element of the stream into the file by calling PrintWriter.println()

Base64 Encoding:

The Base64 encoding encodes a sequence of bytes into a (longer) sequence of printable ASCII characters.

```
Base64.Encoder encoder = Base64.getEncoder();
String password = "password";
String encodedPassword =
encoder.encodeToString(password.getBytes(StandardCharsets.UTF_8));
```

7. JAVA 8 INTERVIEW QUESTIONS

Q. Difference between Abstract class and Interface in Java 8

1. The main difference between an abstract class and interface in Java 8 is the fact that an abstract class is a class and an interface is an interface. A class can have a state which can be modified by non-abstract methods but an interface cannot have the state because they can't have instance variables.
2. The second difference is that an interface cannot have a constructor even in Java 8 but you may remember that abstract class can also have a constructor in Java. All methods of an interface were abstract but since Java 8 you can define non-abstract methods in the form of default and static methods inside interface in Java.

Q. Converting steam to Array.

```
T[] arrayOfT = stream.toArray(T[]::new)
```

```
String[] arrayOfCities = stream.toArray(String[]::new);
```

Q. Remove all empty Strings from List

```
List<String> filtered = strList.stream().filter(x -> x.startsWith("a")).count();
```

https://www.java67.com/2014/04/java-8-stream-examples-and-tutorial.html

Q. What new features were added in Java 8?

Java 8 ships with several new features but the most significant are the following:

- **Lambda Expressions** – a new language feature allowing treating actions as objects
- **Method References** – enable defining Lambda Expressions by referring to methods directly using their names
- **Optional** – special wrapper class used for expressing optionality
- **Functional Interface** – an interface with maximum one abstract method, implementation can be provided using a Lambda Expression
- **Default methods** – give us the ability to add full implementations in interfaces besides abstract methods
- **Nashorn, JavaScript Engine** – Java-based engine for executing and evaluating JavaScript code
- **Stream API** – a special iterator class that allows processing collections of objects in a functional manner

- **Date API** – an improved, immutable JodaTime-inspired Date API

Along with these new features, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

Q. Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

Q. Static method reference:

A method reference can be identified by a double colon separating a class or object name and the name of the method. It has different variations such as constructor reference:

```
String::new;
```

Static method reference:

```
String::valueOf;
```

Bound instance method reference:

```
str::toString;
```

Unbound instance method reference:

```
String::toString;
```

Q. Describe some of the functional interfaces in the standard library.

There are a lot of functional interfaces in the java.util.function package, the more common ones include but not limited to:

- Function – it takes one argument and returns a result
- Consumer – it takes one argument and returns no result (represents a side effect)
- Supplier – it takes not argument and returns a result
- Predicate – it takes one argument and returns a boolean
- BiFunction – it takes two arguments and returns a result
- BinaryOperator – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types

- UnaryOperator – it is similar to a Function, taking a single argument and returning a result of the same type

Q. What is a default method and when do we use it?

Collection interface does not have *forEach* method declaration. Thus, adding such method would simply break the whole collections API.

Java 8 introduces default method so that *Collection* interface can have a default implementation of *forEach* method without requiring the classes implementing this interface to implement the same.

Q. What is a Lambda Expression and what is it used for?

In very simple terms, a lambda expression is a function that can be referenced and passed around as an object.

Q. What is the difference between intermediate and terminal operations?

Stream operations are combined into pipelines to process streams. All operations are either intermediate or terminal.

Intermediate operations are those operations that return *Stream* itself allowing for further operations on a stream.

Some of the intermediate operations are *filter*, *map*, *flatMap*, distinct(), sorted() etc.

Terminal operations include *forEach*, *reduce, Collect*, *sum() count(), min() and max()*.

Q. Difference between map() and flatmap()?

The key difference between map() and flatmap() function is that when you use map(), it applies a function on each element of stream and stores the value returned by the function into a new Stream. This way one stream is transformed into another e.g. a Stream of String is transformed into a Stream of Integer where each element is length of corresponding Stream. Key thing to remember is that the function used for transformation in map() returns a single value. If map() uses a function, which, instead of returning a single value returns a Stream of values than you have a Stream of Stream of values, and flatmap() is used to flat that into a Stream of values.

```
List listOfIntegers = Stream.of("1", "2", "3", "4") .map(Integer::valueOf)
.collect(Collectors.toList());
```

Q. Pipeline Of Operations

A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. In the below example, int array is the source, *filter()* and *distinct()* are intermediate operations and *forEach()* is a terminal operation.

```
IntStream.of(new int[] {4, 7, 1, 8, 3, 9, 7}).filter((int i) -> i >
5).distinct().forEach(System.out::println);
```

Q. Streams are lazily populated

All elements of a stream are not populated at a time. They are lazily populated as per demand because intermediate operations are not evaluated until terminal operation is invoked.

Q. Streams are traversable only once

You can’t traverse the streams more than once just like iterators. If you traverse the stream first time, it is said to be consumed.

```
List<String> nameList = Arrays.asList("Dinesh", "Ross", "Kagiso", "Steyn");

Stream<String> stream = nameList.stream();

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

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

//Error: stream has already been operated upon or closed
```

Q. findFirst(), findAny() defference

Stream.findFirst() returns the first element of this stream, or no element if the stream is empty.

Stream.findAny() returns anyone element of the stream, or an no element if the stream is empty. In non-parallel streams, findAny() will return the first element in most of the cases but this behaviour is not guaranteed. Stream.findAny() method has been introduced for performance gain in case of parallel streams, only.

Q. Why do we need to use Java 8 Stream API in our projects?

- When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
- When want to Perform operations Lazily.
- When we want to write Functional Style programming.
- When we want to perform Parallel Operations.
- When want to use Internal Iteration
- When we want to perform Pipelining operations.
- When we want to achieve better performance.

Q. What is Spliterator

- Spliterator supports both Sequential and Parallel processing of data.
- We can use it for both Collection API and Stream API classes.
- We can NOT use this Iterator for Map implemented classes.
- It uses tryAdvance() method to iterate elements individually in multiple Threads to support Parallel Processing.
- It uses forEachRemaining() method to iterate elements sequentially in a single Thread.
- It uses trySplit() method to divide itself into Sub-Spliterators to support Parallel Processing.

```
Spliterator<String> namesSpliterator = nameList.spliterator();

namesSpliterator.forEachRemaining(System.out::println);
```

Q. Differences between Collection API and Stream API

1. Collection objects are created eagerly while Stream API objects are created lazily.

2. Iterate and Consume elements at any number of times while iterate and Stream consumes only once.

3. any change made on Stream doesn't reflect on original collection

What is the difference between PermGenSpace and MetaSpace?

In jdk 8 onwards PermGenSpace is removed. Earlier PermGenSpace is used for storing the metadata. Metadata means storing the information about classes like bytecodes, names and JIT information. Java classes metadata now stored in native heap and this space is called MetaSpace. Metaspace grows automatically by default and will be garbage collected.