

# **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

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

return b.compareTo(a);

});

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

Collections.sort(names, (String a, String b) -> b.compareTo(a));

List now has a sort method. Also the java compiler is aware of the parameter types so you can skip them as well.

names.sort((a, b) -> b.compareTo(a));

## **Functional Interfaces**

Each lambda corresponds to a given type, specified by an interface. A so called *functional interface* must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're **free to add default methods** to your functional interface. To ensure that your interface meet the requirements, you should add the **@FunctionalInterface** annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

It should have only one abstract method and it can have one default method also. @FunctionalInterface is optional.

|  |
| --- |
| 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(); |

|  |
| --- |
| Thread t1 = new Thread(new Runnable(){  @Override  public void run() {  System.out.println("Running Thread 1");  }  });  t1.start(); |
| JAVA 8:  Thread t1 = new Thread(() -> {  System.out.println("Running Thread 1");  });  t1.start(); |

|  |
| --- |
| Collections.sort(list, new Comparator<String>(){  @Override  public int compare(String str1, String str2)  {  return -str1.compareTo(str2);  }  }); |
| JAVA 8:  Collections.sort(list, (str1,str2) -> {  return -str1.compareTo(str2);  });  If list is List<String> str1 and str2 acts as a String. Or use below code.  Comparator<String> comp = (str1, str2) -> {  return -str1.compareTo(str2);  };  Collections.sort(list, comp); |

## 

## **forEach**

Iterable Interface has forEach() method, List is implemented Iterable Interface.

[**forEach**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)> action)

Consumer is functional interface. All functional interface are in java.util.function.

public interface **Consumer<T>**

It has one method[**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)

|  |
| --- |
| for(String str : list)  {  System.out.println(str);  } |
| JAVA 8:  list.forEach((str) -> {  System.out.println(str);  }); |

map.forEach((k, v) -> System.***out***.println("Key " + k + " Value " + v));

public interface **BiConsumer<T,U>**

It has one method accept(T t, U u)

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

## **Lambda Scopes**

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.

### **Accessing local 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);

### **Accessing fields and static variables**

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

class Lambda4 {

static int outerStaticNum;

int outerNum;

void testScopes() {

Converter<Integer, String> stringConverter1 = (from) -> {

outerNum = 23;

return String.valueOf(from);

};

Converter<Integer, String> stringConverter2 = (from) -> {

outerStaticNum = 72;

return String.valueOf(from);

};

}

}

### **Accessing Default Interface 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);

## **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](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Predicate<T>**

boolean test([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) t)

Predicate<Person> predicate = (p) -> {

return p.getAge() >= 18;

};

list.forEach(person -> {

if(predicate.test(person))

System.out.println(person);

});

Or

Predicate<Person> predicate = (p) -> p.getAge() >= 18;

### **Functions**

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

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.

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

### **Optional**

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

**To create an empty Optional object:**

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

empty.isPresent(); //false

The *isPresent()* API method is used to check if there is a value inside the *Optional* object. A value is present if and only if we have created *Optional* with a non-null value.

**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);

}

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.

**Conditional Return with filter():**

The *filter* API is used to run an inline test on the wrapped value. It takes a predicate as an argument and returns an *Optional* object.

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

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

String name = "muni";

if(name != null && name.equals("muni")) {

System.out.println(name);

}

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

if(optional.filter(n -> n.equals("muni")).isPresent()) {

System.out.println(optional.get());

}

Exp:

public boolean priceIsInRange2(Modem modem2) {

     return Optional.ofNullable(modem2)

       .map(Modem::getPrice)

       .filter(p -> p >= 10)

       .filter(p -> p <= 15)

       .isPresent();

 }

The *map* call is simply used to transform a value to some other value. Keep in mind that this operation does not modify the original value.

**Transforming Value with map():**

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.

String password = " password ";

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

boolean correctPassword = passOpt

      .map(String::trim)

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

      .isPresent();

**Transforming Value with flatMap():**

Just like the *map* API, we also have the *flatMap* API as an alternative for transforming values. The difference is that *map* transforms values only when they are unwrapped whereas *flatMap* takes a wrapped value and unwraps it before transforming it.

Person person = new Person("john", 26);

Optional<Person> personOptional = Optional.of(person);

Optional<Optional<String>> nameOptionalWrapper  = personOptional.map(Person::getName);

Optional<String> nameOptional  = nameOptionalWrapper.orElseThrow(IllegalArgumentException::new);

String name1 = nameOptional.orElse("");

assertEquals("john", name1);

String name = personOptional

      .flatMap(Person::getName)

      .orElse("");

assertEquals("john", name);

## **Default Method in Interfaces**

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

## **Static Method in Interface:**

public interface Interface1 {

public String m1();

default public String m2(){

return "default method";

}

static public String m3(){

return "static method";

}

}

Subclass can’t override m3() method, any class can use m3() method by Interface1.m3().

## **Method References (::)**

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

The above example code can be further simplified by utilizing static method references:

Converter<String, Integer> converter = Integer::valueOf;

Integer converted = converter.convert("123");

System.out.println(converted); // 123

# **Stream API**

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 date 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();

## **Filter**

Filter accepts a predicate to filter all elements of the stream. This operation is *intermediate* which enables us to call another stream operation (*forEach*) on the result. *forEach* accepts a consumer to be executed for each element in the filtered stream. *forEach* is a terminal operation. It's *void*, so we cannot call another stream operation.

stringCollection

.stream()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println); // prints the names starts with ‘a’.

## **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*.

stringCollection

.stream()

.sorted()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println); // "aaa1", "aaa2"

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 *stringCollection* is untouched:

System.out.println(stringCollection);

// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

## **Map**

*intermediate* operation map converts each element into another object via the given function.

stringCollection

.stream()

.map(String::toUpperCase)

.sorted((a, b) -> b.compareTo(a))

.forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

## **Match**

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are *terminal* and return a *boolean* result.

boolean anyStartsWithA = stringCollection

.stream()

.anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

boolean allStartsWithA = stringCollection

.stream()

.allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

boolean noneStartsWithZ = stringCollection

.stream()

.noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true

## **Count**

Count is a *terminal* operation returning the number of elements in the stream as a *long*.

long startsWithB = stringCollection.stream().filter((s) -> s.startsWith("b"))

.count();

System.out.println(startsWithB); // 3

## **Parallel Streams**

As mentioned above streams can be either sequential or parallel. Operations on sequential streams are performed on a single thread while operations on parallel streams are performed concurrent on multiple threads.

long count = values.stream().sorted().count();

// sequential sort took: 899 ms

long count = values.parallelStream().sorted().count();

// parallel sort took: 472 ms

# **New Date API**

Java 8 contains a brand new date and time API under the package *java.time.* The new Date API is comparable with the [Joda-Time](http://www.joda.org/joda-time/) library, however it's [not the same](http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html). The following examples cover the most important parts of this new API.

New date API is immutable and thread safe.

LocalDate localDate = LocalDate.*now*();

LocalTime localTime = LocalTime.*now*();

LocalDateTime localDateTime = LocalDateTime.*now*();

## **Clock**

Clock provides access to the current date and time. Clocks are aware of a *timezone* and may be used instead of *System.currentTimeMillis()*to retrieve the current milliseconds. Such an instantaneous point on the time-line is also represented by the class *Instant*. Instants can be used to create legacy *java.util.Date* objects.

Clock clock = Clock.systemDefaultZone();

long millis = clock.millis();

Instant instant = clock.instant();

Date legacyDate = Date.from(instant); // legacy java.util.Date

## **Timezones**

*Timezones* are represented by a *ZoneId*. They can easily be accessed via static factory methods. *Timezones* define the offsets which are important to convert between instants and local dates and times.

System.out.println(ZoneId.getAvailableZoneIds());

// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");

ZoneId zone2 = ZoneId.of("Brazil/East");

System.out.println(zone1.getRules());

System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]

// ZoneRules[currentStandardOffset=-03:00]

## **LocalTime**

*LocalTime* represents a time without a timezone, e.g. 10pm or 17:30:15. The following example creates two local times for the timezones defined above. Then we compare both times and calculate the difference in hours and minutes between both times.

LocalTime now1 = LocalTime.now(zone1);

LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);

long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3

System.out.println(minutesBetween); // -239

*LocalTime* comes with various factory method to simplify the creation of new instances, including parsing of time strings.

LocalTime late = LocalTime.of(23, 59, 59);

System.out.println(late); // 23:59:59

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedTime(FormatStyle.SHORT)

.withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);

System.out.println(leetTime); // 13:37

## **LocalDate**

*LocalDate* represents a distinct date, e.g. 2014-03-11. It's immutable and works exactly analog to *LocalTime*. The sample demonstrates how to calculate new dates by adding or substracting days, months or years. Keep in mind that each manipulation returns a new instance.

LocalDate today = LocalDate.now();

LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);

LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);

DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek); // FRIDAY

Parsing a *LocalDate* from a string is just as simple as parsing a *LocalTime*:

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedDate(FormatStyle.MEDIUM)

.withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);

System.out.println(xmas); // 2014-12-24

## **LocalDateTime**

*LocalDateTime* represents a date-time. It combines date and time as seen in the above sections into one instance. *LocalDateTime* is immutable and works similar to *LocalTime* and *LocalDate*. We can utilize methods for retrieving certain fields from a date-time:

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();

System.out.println(dayOfWeek); // WEDNESDAY

Month month = sylvester.getMonth();

System.out.println(month); // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);

System.out.println(minuteOfDay); // 1439

With the additional information of a *timezone* it can be converted to an instant. Instants can easily be converted to legacy dates of type *java.util.Date*.

Instant instant = sylvester

.atZone(ZoneId.systemDefault())

.toInstant();

Date legacyDate = Date.from(instant);

System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014

Formatting date-times works just like formatting dates or times. Instead of using pre-defined formats we can create formatters from custom patterns.

DateTimeFormatter formatter =

DateTimeFormatter

.ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);

String string = formatter.format(parsed);

System.out.println(string); // Nov 03, 2014 - 07:13

Unlike *java.text.NumberFormat* the new *DateTimeFormatter* is immutable and **thread-safe**.

# **Miscellaneous**

## **String Joining**

String string = String.*join*(",", "muni","swamy","palla");//muni,swamy,palla

String string = String.*join*(" and ", "muni","swamy","palla");// u can use String[] also. Or you can use StringJoiner class.

StringJoiner joiner = **new** StringJoiner(",");

joiner.add("muni").add("swamy").add("palla");// muni,swamy,palla

# Interview Questions

### **Q1. 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.

## **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.

**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;

### **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

### 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.

### **What is a Lambda Expression and what is it used for?**

In very simple terms, a lambda expression is a function.

### **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()*.

### **What is stream pipelining in Java 8?**

Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories: intermediate operations and terminal operations.

**Difference between map() and flatmap()?**

The key difference between map() and flatmap() function is that when you use [map()](http://java67.blogspot.com/2015/01/java-8-map-function-examples.html), 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()](http://java67.blogspot.com/2016/03/how-to-use-flatmap-in-java-8-stream.html)is used to flat that into a Stream of values.

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

#### **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.

#### **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

# **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.

List<Employee1> empList = **new** ArrayList<>();

empList.add(**new** Employee1("Nataraja G", "Accounts", 8000));

empList.add(**new** Employee1("Nagesh Y", "Admin", 15000));

empList.add(**new** Employee1("Vasu V", "Security", 2500));

empList.add(**new** Employee1("Amar", "Admin", 12500));

empList.stream().filter(emp -> emp.getAccount().matches("Admin")).findFirst().ifPresent(System.***out***::println);

empList.stream().filter(emp -> emp.getAccount().matches("Admin")).findAny().ifPresent(System.***out***::println);

empList.parallelStream().filter(emp -> emp.getAccount().matches("Admin")).findAny()

.ifPresent(System.***out***::println);

Distinct:

Arrays.*stream*(**new**Integer[]{1,2,3,4,5,2,3}).distinct().collect(Collectors.*toList*());

String output = list.stream().distinct().collect(Collectors.joining(","));

long l = list.stream().distinct().count();

List<Book> list = **new** ArrayList<>();

list.add(**new** Book("Core Java", 200));

list.add(**new** Book("Core Java", 200));

list.add(**new** Book("Learning Freemarker", 150));

list.add(**new** Book("Spring MVC", 300));

list.add(**new** Book("Spring MVC", 300));

**long** l = list.stream().distinct().count();

System.***out***.println("No. of distinct books:" + l);

list.stream().distinct().forEach(b -> System.***out***.println(b.getName() + "," + b.getPrice()));

// class should override hashCode() and equals() in order to get distinct elements.

**Output:**

No. of distinct books:3

Core Java,200

Learning Freemarker,150

Spring MVC,300

One of the possible solutions would be to implement a stateful *Predicate:*

list.stream().filter(*distinctByKey*(b -> b.getName()))

.forEach(b -> System.***out***.println(b.getName()+ "," + b.getPrice()));

**private** **static** <T> Predicate<T> distinctByKey(Function<? **super** T, ?> keyExtractor) {

Map<Object,Boolean> seen = **new** ConcurrentHashMap<>();

**return** t -> seen.putIfAbsent(keyExtractor.apply(t), Boolean.***TRUE***) == **null**;

}

Output:

Core Java,200

Learning Freemarker,150

Spring MVC,300