## **Java 8 - lambda expression**

* Lambda expression is just an anonymous function, i.e., a function with no name and without being bounded to an identifier.
* The most important features of Lambda Expressions is that they execute in the context of their appearance. So, a similar lambda expression can be executed differently in some other context
* The benefits involved in lambda expression is functional programming over object oriented programming
* Functional programming, you can define functions, give them reference variables and pass them as method arguments and much more. JavaScript is a good example of this where you can pass callback methods

1.1 **Lambda Expression Syntax**

**(x, y) -> x + y //This function takes two parameters**

**//and return their sum.**

* Now based on type of x and y, method may be used in multiple places. Parameters can match to int, or Integer or simply String also.
* Based on context, it will either add two integers or concat two strings.

**1.2 Other Possible Syntax**

either

(parameters) -> expression //1

or

(parameters) -> { statements; } //2

or

() -> expression //3

**Examples:**

(int a, int b) -> a \* b // takes two integers and returns their multiplication

(a, b) -> a - b // takes two numbers and returns their difference

() -> 99 // takes no values and returns 99

(String a) -> System.out.println(a) // takes a string, prints its value to the console, and returns nothing

a -> 2 \* a // takes a number and returns the result of doubling it

c -> { //some complex statements } // takes a collection and do some processing

**1.3 lambda expression examples**

1) Iterating over a List and perform some operations

List<String> pointList = new ArrayList();

pointList.add("1");

pointList.add("2");

pointList.forEach(p -> {

System.out.println(p);

//Do more work

}

);

2) Create a new runnable and pass it to thread

new Thread(

() -> {System.out.println("My Runnable");}

).start();

## **2. Java 8 - Functional Interfaces**

## **What is functional interface?**

* Functional interfaces which permit exactly one abstract method inside them. These interfaces are also called Single Abstract Method interfaces (SAM Interfaces).
* In Java 8, functional interfaces can be represented using lambda expressions, method reference and constructor references as well.
* Java 8 introduces an annotation i.e. @FunctionalInterface too, which can be used for compiler level errors when the interface you have annotated violates the contracts of exactly one abstract method.

@FunctionalInterface

public interface MyFirstFunctionalInterface

{

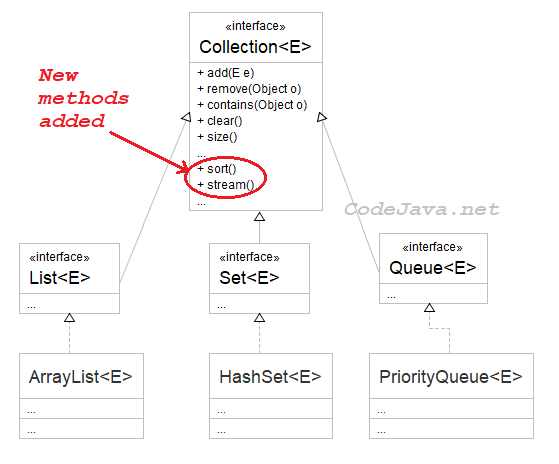
public void firstWork();

}

* A functional interface is valid even if the @FunctionalInterface annotation would be omitted. It is only for informing the compiler to enforce single [abstract method](https://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface.
* Conceptually, a functional interface has exactly one abstract method. Since [default methods](https://howtodoinjava.com/java8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re free to add default methods to your functional interface as many as you like.
* If an interface declares an *abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count*

## **3. Java 8 - Default Methods**

* Java 8 comes with some new features like Streams and Lambda expressions which require adding of some new methods in the core interfaces of the [Java Collections framework](https://www.codejava.net/java-core/collections/what-is-java-collections-framework). For example, the Collection interface needs to add two methods sort() and stream()



* However, there are many sub interfaces and implementation classes in the Collection inheritance tree – so adding new methods to the Collection interface will break these existing classes – require them to implement the new methods. And worse, there are also countless of classes written by programmers that depend on the Collection inheritance tree – causing their applications to fail if the new methods added to Collection interface works the old way: concrete classes must implement methods declared in its super interfaces.
* Therefore, the designers of Java language decided to introduce ***default methods*** for interface. A default method is declared with the [default](https://www.codejava.net/java-core/the-java-language/default-keyword-in-java) modifier before the return type, and it contains method body (implementation). For example, the new methods added to the Collection interface are declared as follows:

public interface Collection<E> {

public default void sort() {

// implementation goes here

}

public default Stream<E> stream() {

// implementation goes here

}

// other abstract methods

}

* So the purpose of default methods is to make changes to existing interfaces without breaking its existing subclasses – for backward compatibility. That means you should not use default methods for completely new code. Use default methods to make changes to old code - existing interfaces – to avoid changes to existing implementation classes.

## **Default method and multiple inheritance in Java:**

There can be a case in which a class implements two different interfaces having the same default method, for example:

public interface X {

void foo();

default void bar() {

// code

}

}

public interface Y {

void doo();

default void bar() {

// code

}

}

public class XYImpl implements X, Y {

}

In this case, to avoid ambiguity, the sub class must override the common default method, for example:

public class XYImpl implements X, Y {

public void foo() {

// implement from X

}

public void doo() {

// implement from Y

}

public void bar() {

// override from X, Y

}

}

## **Default method and static method in interface in Java:**

Also since Java 8, we can write static methods (with code body) in an interface.

public interface ABC {

void foo(); // abstract method

static void code1() {

// code snippet #1

}

default void bar() {

code1();

// other bar's code

}

default void doo() {

code1();

// other doo's code

}

}

## **4. Java 8 - Streaming**

Java Stream API brings to us totally new ways for working with collections. Once you are familiar with it, you will love using it, as it makes you write code more natural, more succinct, more readable and most importantly, your productivity will go up incredibly.

Let’s start by looking at some code examples to understand how the Stream API radically changes the way we work with collections

public class Student implements Comparable<Student> {

private String name;

private int score;

public Student(String name, int score) {

this.name = name;

this.score = score;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public void setScore(int score) {

this.score = score;

}

public int getScore() {

return this.score;

}

public String toString() {

return this.name + " - " + this.score;

}

public int compareTo(Student another) {

return another.getScore() - this.score;

}

}

List<Student> listStudents = new ArrayList<>();

listStudents.add(new Student("Alice", 82));

listStudents.add(new Student("Bob", 90));

listStudents.add(new Student("Carol", 67));

listStudents.add(new Student("David", 80));

listStudents.add(new Student("Eric", 55));

listStudents.add(new Student("Frank", 49));

listStudents.add(new Student("Gary", 88));

listStudents.add(new Student("Henry", 98));

listStudents.add(new Student("Ivan", 66));

listStudents.add(new Student("John", 52));

We are required to do some calculations on this list.

First, find the students whose scores are greater than or equal to 70.

A non-stream solution would look like this:

// find students whose score >= 70

List<Student> listGoodStudents = new ArrayList<>();

for (Student student : listStudents) {

if (student.getScore() >= 70) {

listBadStudents.add(student);

}

}

for (Student student : listGoodStudents) {

System.out.println(student);

}

With the Stream API, we can replace the above code with the following:

// find students whose score >= 70

List<Student> listGoodStudents = listStudents.stream()

.filter(s -> s.getScore() >= 70)

.collect(Collectors.toList());

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

Second, calculate average score of all students. A trivial solution would look like this:

// calculate average score of all students

double sum = 0.0;

for (Student student : listStudents) {

sum += student.getScore();

}

double average = sum / listStudents.size();

System.out.println("Average score: " + average);

And here’s a stream-based version:

// calculate average score of all students

double average = listStudents.stream()

.mapToInt(s -> s.getScore())

.average().getAsDouble();

System.out.println("Average score: " + average);

## **1. What is a Stream?**

A stream represents a sequence of elements supporting sequential and parallel [aggregate operations](https://www.codejava.net/java-core/collections/java-stream-aggregate-functions-examples-intermediate-operations). Since Java 8, we can generate a stream from a collection, an array or an I/O channel.

Every collection class now has the **stream()** method that returns a stream of elements in the collections:

Stream<Student> stream = listStudents.stream();

Obtaining a stream from an array:

int[] arrayIntegers = {1, 8, 2, 3, 98, 11, 35, 91};

IntStream streamIntegers = Arrays.stream(arrayIntegers);

Obtaining a stream from a file:

BufferedReader bufferReader = new BufferedReader(new FileReader("students.txt"));

Stream<String> streamLines = bufferReader.lines();

Operations can be performed on a stream are categorized into [intermediate operations](https://www.codejava.net/java-core/collections/java-stream-aggregate-functions-examples-intermediate-operations) and [terminal operations](https://www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples). We’ll see details of these operations shortly. Consider the following code:

List<Student> top3Students = listStudents.stream()

.filter(s -> s.getScore() >= 70)

.sorted()

.limit(3)

.collect(Collectors.toList());

System.out.println("Top 3 Students by Score:");

top3Students.forEach(s -> System.out.println(s));

This code can be read as: select top 3 students whose scores >= 70, and sort them by score in descending order (the natural ordering of the Student class). Here we can see the following intermediate operations: filter, sorted and limit; and the terminal operation is collect.

As you can see, the operations on a stream can be chained together (intermediate operations) and end with a terminal operation. Such a chain of stream operations is called stream pipeline.

## 

## **Intermediate Operations**

An [intermediate operation](https://www.codejava.net/java-core/collections/java-stream-aggregate-functions-examples-intermediate-operations) processes over a stream and return a new stream as a response. Then we can execute another intermediate operation on the new stream, and so on, and finally execute the terminal operation.

One interesting point about intermediate operations is that they are lazily executed. That means they are not run until a terminal operation is executed.

The Stream API provides the following common intermediate operations:

* **map()**
* **filter()**
* **sorted()**
* **limit()**
* **distinct()**

For a full list of intermediate operations, consult the [Stream Javadoc](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html).

## **Terminal Operations**

A stream pipeline always ends with a terminal operation, which returns a concrete type or produces a side effect. For instances, the collect operation produces a collection; the forEach operation does not return a concrete type, but allows us to add side effect such as print out each element.

Unlike lazily-executed terminate operations, a terminal operation is always eagerly executed. The [common terminal operations](https://www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples) provided by the Stream API include:

* **collect()**
* **reduce()**
* **forEach()**

See the [Stream Javadoc](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) for a complete list of terminal operations supported.

## **Parallel Streams**

The powerful feature of streams is that stream pipelines may execute either sequentially or in parallel. All collections support the **parallelStream()** method that returns a possibly parallel stream:

Stream<Student> parallelStream = listStudents.parallelStream();

When a stream executes in parallel, the Java runtime divides the stream into multiple sub streams. The aggregate operations iterate over and process these sub streams in parallel and then combine the results.

The advantage of parallel streams is performance increase on large amount of input elements, as the operations are executed currently by multiple threads on a multi-core CPU.

For example, the following code may execute stream operations in parallel:

listStudents.parallelStream()

.filter(s -> s.getScore() >= 70)

.sorted()

.limit(3)

.forEach(System.out::println);

The Collection’s stream() method returns a sequential stream. We can convert a sequential stream to a parallel stream by calling the **parallel()** method on the current stream. The following example shows a stream executes the sorted operation sequentially, and then execute the filter operation in parallel:

listStudents.stream()

.sorted()

.parallel()

.filter(s -> s.getScore() >= 70)

.forEach(System.out::println);

## 

## **Streams and Lambda Expressions**

As you can see in the above examples, Lambda expressions can be used as arguments in aggregate functions. This allows us to write code more flexibility and more compact. Remember that the parameter in the Lambda expression is implicitly the object being processed in the stream.

Consider the following example:

listStudents.stream()

.sorted()

.filter(s -> s.getScore() >= 70)

.forEach(System.out::println);

Here, we use a Lambda expression in the filter operation and a static method reference in the forEach operation. The s parameter is of type Student because the stream is a sequence of Student objects.

Some operations can transform a stream of type A to a stream of type B, such as the map operation in the following example:

listStudents.stream()

.filter(s -> s.getScore() >= 70)

.map(s -> s.getName())

.sorted()

.forEach(name -> System.out.println(name));

In Lambda expressions used with the filter and map operations, the s parameter is of type Student. However the map operation produces a stream of Strings, so the name parameter in the Lambda expression in the forEach operation is of type String. So pay attention to this kind of transformation when using Lambda expressions.

## **Streams vs. Collections**

A [collection](https://www.codejava.net/java-core/collections) is a data structure that holds elements. Each element is computed before it actually becomes a part of that collection.

On the other hand, a stream is not a data structure. A stream is a pipeline of operations that compute the elements on-demand. Though we can create a stream from a collection and apply a number of operations, the original collection doesn’t change. Hence streams cannot mutate data

And a key characteristic of streams is that they can transform data, as operations on a stream can produce another data structure, like the map and collect operation as shown in the above examples.

## **5. Java 8 - Date and Time**

Java 8 introduced new APIs for *Date* and *Time* to address the shortcomings of the older *java.util.Date* and *java.util.Calendar*.

## **Issues with the Existing *Date*/*Time* APIs**

* **Thread Safety** – The *Date* and *Calendar* classes are not thread safe, leaving developers to deal with the headache of hard to debug concurrency issues and to write additional code to handle thread safety. On the contrary the new *Date* and *Time* APIs introduced in Java 8 are immutable and thread safe, thus taking that concurrency headache away from developers.
* **APIs Design and Ease of Understanding** – The *Date* and *Calendar* APIs are poorly designed with inadequate methods to perform day-to-day operations. The new *Date/Time* APIs is ISO centric and follows consistent domain models for date, time, duration and periods. There are a wide variety of utility methods that support the commonest operations.
* ***ZonedDate* and *Time*** – Developers had to write additional logic to handle timezone logic with the old APIs, whereas with the new APIs, handling of timezone can be done with *Local* and *ZonedDate*/*Time* APIs.

## **Using *LocalDate*, *LocalTime* and *LocalDateTime***

The most commonly used classes are *LocalDate*, *LocalTime* and *LocalDateTime*. As their names indicate, they represent the local Date/Time from the context of the observer.

These classes are mainly used when timezone are not required to be explicitly specified in the context.

### **Working With *LocalDate***

The *LocalDate* represents **a date in ISO format (yyyy-MM-dd) without time**.

It can be used to store dates like birthdays and paydays.

An instance of current date can be created from the system clock as below:

|  |  |
| --- | --- |
| 1 | LocalDate localDate = LocalDate.now(); |

The *LocalDate* representing a specific day, month and year can be obtained using the “*of*” method or by using the “*parse*” method. For example the below code snippets represents the *LocalDate* for 20 February 2015:

|  |  |
| --- | --- |
| 1  2  3 | LocalDate.of(2015, 02, 20);  LocalDate.parse("2015-02-20"); |

The *LocalDate* provides various utility methods to obtain a variety of information. Let's have a quick peek at some of these APIs methods.

The following code snippet gets the current local date and adds one day:

|  |  |
| --- | --- |
| 1 | LocalDate tomorrow = LocalDate.now().plusDays(1) |

This example obtains the current date and subtracts one month. Note how it accepts an *enum* as the time unit:

|  |  |  |  |
| --- | --- | --- | --- |
|  | LocalDate previousMonthSameDay = LocalDate.now().minus(1, ChronoUnit.MONTHS);  In the following two code examples we parse the date “2016-06-12” and get the day of the week and the day of the month respectively. Note the return values, the first is an object representing the *DayOfWeek* while the second in an *int* representing the ordinal value of the month:   |  |  | | --- | --- | |  | DayOfWeek sunday = LocalDate.parse("2016-06-12").getDayOfWeek();    int twelve = LocalDate.parse("2016-06-12").getDayOfMonth(); | |

### **Working With *LocalTime***

The *LocalTime* represents **time without a date**.

Similar to *LocalDate* an instance of *LocalTime* can be created from system clock or by using “parse” and “of” method. Quick look at some of the commonly used APIs below.

An instance of current *LocalTime* can be created from the system clock as below:

|  |  |
| --- | --- |
| 1 | LocalTime now = LocalTime.now(); |

In the below code sample*,* we create a *LocalTime* representing 06:30 AM by parsing a string representation:

|  |  |
| --- | --- |
| 1 | LocalTime sixThirty = LocalTime.parse("06:30"); |

The Factory method “of” can be used to create a *LocalTime*. For example the below code creates *LocalTime* representing 06:30 AM using the factory method:

|  |  |
| --- | --- |
| 1 | LocalTime sixThirty = LocalTime.of(6, 30); |

The below example creates a *LocalTime* by parsing a string and adds an hour to it by using the “plus” API. The result would be *LocalTime* representing 07:30 AM:

|  |  |
| --- | --- |
| 1 | LocalTime sevenThirty = LocalTime.parse("06:30").plus(1, ChronoUnit.HOURS); |

Various getter methods are available which can be used to get specific units of time like hour, min and secs like below:

|  |  |
| --- | --- |
| 1 | int six = LocalTime.parse("06:30").getHour(); |

### **Working With *LocalDateTime***

The *LocalDateTime* is used to represent **a combination of date and time**.

This is the most commonly used class when we need a combination of date and time. The class offers a variety of APIs and we will look at some of the most commonly used ones.

An instance of *LocalDateTime* can be obtained from the system clock similar to *LocalDate* and *LocalTime:*

|  |  |
| --- | --- |
| 1 | LocalDateTime.now(); |

The below code samples explain how to create an instance using the factory “of” and “parse” methods. The result would be a *LocalDateTime* instance representing 20 February 2015, 06:30 AM:

|  |  |
| --- | --- |
| 1 | LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30); |

|  |  |
| --- | --- |
| 1 | LocalDateTime.parse("2015-02-20T06:30:00"); |

There are utility APIs to support addition and subtraction of specific units of time like days, months, year and minutes are available. The below code samples demonstrates the usage of “plus” and “minus” methods. These APIs behave exactly like their counterparts in *LocalDate* and *LocalTime:*

|  |  |
| --- | --- |
| 1 | localDateTime.plusDays(1); |

|  |  |
| --- | --- |
| 1 | localDateTime.minusHours(2); |

Getter methods are available to extract specific units similar to the date and time classes. Given the above instance of *LocalDateTime*, the below code sample will return the month February:

|  |  |
| --- | --- |
| 1 | localDateTime.getMonth(); |

## 

## **Using *ZonedDateTime* API**

Java 8 provides *ZonedDateTime* when we need to deal with time zone specific date and time. The *ZoneId* is an identifier used to represent different zones. There are about 40 different time zones and the *ZoneId* are used to represent them as follows.

In this code snippet we create a *Zone* for Paris:

|  |  |
| --- | --- |
| 1 | ZoneId zoneId = ZoneId.of("Europe/Paris"); |

A set of all zone ids can be obtained as below:

|  |  |
| --- | --- |
| 1 | Set<String> allZoneIds = ZoneId.getAvailableZoneIds(); |

The *LocalDateTime* can be converted to a specific zone:

|  |  |
| --- | --- |
| 1 | ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId); |