# Java 8 (Streams, lambda expressions, method references, Optional class features)

- streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.
- A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.
- A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation (like collect, count etc).
- All intermediate operations are lazy, so they're not executed until a result of a processing is actually needed.

**Data source:** collections and arrays

Package "java.util.stream"

## What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

- **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes element on demand. It never stores the elements.
- **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement.

SELECT max(salary), employee_id, employee_name FROM Employee

## Generating Streams

With Java 8, Collection interface has two methods to generate a Stream.

- **stream()** − Returns a sequential stream considering collection as its source.
- **parallelStream()** − Returns a parallel Stream considering collection as its source.

**Que: Find occurrence of given char in the string.**

```
        String s = "what a good day";
        long count = findOccurrance( s, 'a' );
    public static long findOccurrance2( String string, char c ) {
        String temp = string.replace( " ", "" );
         long occur = temp.chars().mapToObj( i->(char)i ).filter( x -> x.equals( c ) ).count();
       //or  long occur = temp.codePoints().mapToObj( i -> (char)i ).filter( x -> x.equals( c ) ).count();
        return occur;
    }
```

chars() => returns IntStream

mapToObj( i->(char)i ) => returns Stream<Character>

**Que: Find occurrence of each character in string.**

```
public static HashMap<Character, Integer> findOccurrance( String string ) {
        String temp = string.replace( " ", "" );
        HashMap<Character, Integer> hm = new HashMap<Character, Integer>();
        temp.chars().mapToObj( i -> (char)i ).forEach( x -> {
            if ( hm.containsKey( x ) )
                hm.put( x, hm.get( x ) + 1 );
            else
                hm.put( x, 1 );

        } );
        hm.keySet().stream().forEach( k -> {
            System.out.println( k + ": " + hm.get( k ) );
        } );

        return hm;
    }
```

**Stream Creations:**

ways to get stream

1. *Arrays.stream(array): //from an existing array(primitive or obj array both)*
2. *Stream.of(arrayOfEmps); //from an existing array((primitive or obj array both)*
3. *empList.stream(); //from an existing list or set or collection*
4. `StreamSupport.stream(al.spliterator(), parallelBoolean);//false will give sequential stream //al is a collection`
5. using Stream.builder():

```
//          Stream.Builder<Employee> empStreamBuilder = Stream.builder();
//          empStreamBuilder.accept(arrayOfEmps[0]);
//          empStreamBuilder.accept(arrayOfEmps[1]);
//          empStreamBuilder.accept(arrayOfEmps[2]);
//          Stream<Employee> empStream = empStreamBuilder.build();
```

```
public static <T> Stream<T> stream(Spliterator<T> spliterator,
                                   boolean parallel)
```

```
        Set<String> set = new TreeSet<>(Arrays.asList("three", "one",
"two"));
        Stream<String> stream = StreamSupport.stream(set.spliterator(),
false);
        stream.forEach(System.out::println);
```

## Java – How to convert Array to Stream

In Java 8, you can either use `Arrays.stream` or `Stream.of` to convert an Array into a Stream.

## 1. Object Arrays

For object arrays, both `Arrays.stream` and `Stream.of` returns the same output.

Note-For object arrays, the `Stream.of` method calls the `Arrays.stream` internally. So better to use Arrays.stream.

```
String[] array = {"a", "b", "c", "d", "e"};

        //Arrays.stream
        Stream<String> stream1 = Arrays.stream(array);
        stream1.forEach(x -> System.out.println(x));

        //Stream.of
        Stream<String> stream2 = Stream.of(array);
        stream2.forEach(x -> System.out.println(x));
```

## 2. Primitive Arrays

For primitive array, the `Arrays.stream` and `Stream.of` will return different output.

```
        // 1. Arrays.stream -> IntStream
        IntStream intStream1 = Arrays.stream(intArray);
        intStream1.forEach(x -> System.out.println(x));

        // 2. Stream.of -> Stream<int[]>
        Stream<int[]> temp = Stream.of(intArray);

        // Cant print Stream<int[]> directly, convert / flat it to IntStream
        IntStream intStream2 = temp.flatMapToInt(x -> Arrays.stream(x));
        intStream2.forEach(x -> System.out.println(x));
```

**Stream operations/methods:**

**Intermediate operations:**

- forEach
- map
- filter
- limit
- skip
- sorted
- findFirst
- distinct

Terminal operations:

- collect : return type list/set/collection/charSequence etc
- count : return type long

- collect(Collectors.*toList*() );
- collect(Collectors.*toSet*() );
- collect(Collectors.*toCollection*(Vector::new) );
- collect(Collectors.*joining*(", ") ).toString();

**stream to array direct:** `empList.stream().toArray(Employee[]::new);`

**other functions on stream:**

allMatch, anyMatch, and noneMatch: return type boolean

```
int a[]={3, 2, 2, 3, 7, 3, 5};

System.out.println( "boolean is: " +Arrays.stream( a ).anyMatch( i->i==7 ) ); //true
System.out.println("boolean is: " +Arrays.stream( a ).allMatch( i->i==7 ) ); //false
System.out.println( "boolean is: "+ Arrays.stream( a ).noneMatch( i->i==7 )); //false
```

System.out.println( "list with unique ele: :" +Arrays.stream( a ).distinct().mapToObj( x->(int)x ).collect( Collectors.toList() ) ); //[3, 2, 7, 5]

//mapToObj: to convert an IntStream into Stream<Integer> //required in case of stream created from primitive array

## Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

The strategy for this operation is provided via the Collector interface implementation.

## Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
        //List
      ArrayList<String> el = new ArrayList<String>();
        el.add( "hi1" );
        el.add( "" );
        el.add( "hi2" );
        el.add( "h3" );
        List<String> filtered1 = el.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
        System.out.println( "dkv: el: "+filtered1 );

        List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
        List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
        System.out.println( "dkv: "+filtered );



        //Array
         String strings1[] = {"abc", "", "bc", "efg", "abcd","", "jkl"};


         List<String> array1 = Stream.of( strings1 ).filter(string -> !string.isEmpty()).collect(Collectors.toList());
         System.out.println( "dkv:array1:  "+array1 );




        List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
       //get list of unique squares
       List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
       System.out.println( "dkv: "+squaresList );



    //get count of empty string in list/array. count return type is long.
    long count = strings.stream().filter(string -> string.isEmpty()).count();
    System.out.println( "dkv: "+count );


    //parallel stream
    long count1 = strings.parallelStream().filter(string -> string.isEmpty()).count();
    System.out.println( "dkv1: "+count1 );
    List<Integer> numbers1 = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

    IntSummaryStatistics stats = numbers1.stream().mapToInt((x) -> x).summaryStatistics();

    System.out.println("Highest number in List : " + stats.getMax());
    System.out.println("Lowest number in List : " + stats.getMin());
    System.out.println("Sum of all numbers : " + stats.getSum());
    System.out.println("Average of all numbers : " + stats.getAverage());
    System.out.println("count of all numbers : " + stats.getCount());



    Random random = new Random();
    random.ints().limit(10).forEach(System.out::println);
    System.out.println( "dkv: sorted:" );
    random.ints().limit(10).sorted().forEach(System.out::println);
    }
```

## *Stream Creation*

obtain a stream from an existing array:

```
private static Employee[] arrayOfEmps = {
   new Employee(1, "Jeff Bezos", 100000.0),
    new Employee(2, "Bill Gates", 200000.0),

    new Employee(3, "Mark Zuckerberg", 300000.0)

};

Stream.of(arrayOfEmps);

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();

empList.stream().forEach(e -> e.salaryIncrement(10.0));
```

Filter:

```
    List<Employee> employees = Stream.of(empIds)

      .map(employeeRepository::findById)

      .filter(e -> e != null)

      .filter(e -> e.getSalary() > 200000)

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

findFirst

findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty:

```
  Integer[] empIds = { 1, 2, 3, 4 };



    Employee employee = Stream.of(empIds)

      .map(employeeRepository::findById)

      .filter(e -> e != null)

      .filter(e -> e.getSalary() > 100000)

      .findFirst()

      .orElse(null);
```

Here, the first employee with the salary greater than 100000 is returned. If no such employee exists, then null is returned.

## 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()*:

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

Some operations are deemed **short-circuiting operations(**like skip,limit**)**. Short-circuiting operations allow computations on infinite streams to complete in finite time:

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

//2,4,8,16,32,64,128,256

    List<Integer> collect = infiniteStream

      .skip(3)

      .limit(5)

      .collect(Collectors.toList());



    assertEquals(collect, Arrays.asList(16, 32, 64, 128, 256));
```

## allMatch, anyMatch, *and* noneMatch

These operations all take a predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined:

```
  List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);



    boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);

    boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);

    boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);
    assertEquals(allEven, false);

    assertEquals(oneEven, true);

    assertEquals(noneMultipleOfThree, false);
```

*allMatch()* checks if the predicate is true for all the elements in the stream. Here, it returns *false* as soon as it encounters 5, which is not divisible by 2.

## Advanced *collect*

joining

```
    String empNames = empList.stream()

      .map(Employee::getName)

      .collect(Collectors.joining(", "))

      .toString();



    assertEquals(empNames, "Jeff Bezos, Bill Gates, Mark Zuckerberg");
```

### toSet

We can also use *toSet()* to get a set out of stream elements:

```
    Set<String> empNames = empList.stream()

            .map(Employee::getName)

            .collect(Collectors.toSet());



    assertEquals(empNames.size(), 3);
```

## toCollection

We can use *Collectors.toCollection()* to extract the elements into any other collection by passing in a *Supplier<Collection>*. We can also use a constructor reference for the *Supplier*:

```
Vector<String> empNames = empList.stream()

            .map(Employee::getName)

            .collect(Collectors.toCollection(Vector::new));
```

Here, an empty collection is created internally, and its *add()* method is called on each element of the stream.

Parallel stream:

```
empList.stream().parallel().forEach(e -> e.salaryIncrement(10.0));
```

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

- 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.
- 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.
- 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.

## Infinite Streams

Sometimes, we might want to perform operations while the elements are still getting generated. We might not know beforehand how many elements we'll need. Unlike using list or map, where all the elements are already populated, we can use infinite streams, also called as unbounded streams.

There are two ways to generate infinite streams:

```
Stream.generate(Math::random)

      .limit(5)
      .forEach(System.out::println);

    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.

## Lambda expressions

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming and simplifies the development a lot.

**Syntax:**

```
parameter -> expression body
```

Following are the important characteristics of a lambda expression.

- **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.

- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

```
//with type declaration
 MathOperation addition = (int a, int b) -> a + b;

 //with out type declaration
 MathOperation subtraction = (a, b) -> a - b;

 //with return statement along with curly braces
 MathOperation multiplication = (int a, int b) -> { return a * b; };

 //without return statement and without curly braces
 MathOperation division = (int a, int b) -> a / b;
```

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

**Examples:**

**StreamSupport.stream(orgList.spliterator(), false).findAny().get().getContact().getName();**

**StreamSupport.stream( userList.spliterator(), false ).allMatch( userStatus->userStatus.getUserStatus().equals(status) );**

**StreamSupport.stream(siteList.spliterator(), false).anyMatch( siteUuid -> siteUuid.getUuid().equals( siteId ));**

**codes.stream().map( code -> code.getCode().getValue()).collect( Collectors.toList() ).containsAll( expectedRoleList );**

```
case "organization":

            flag=StreamSupport.stream(entityRoles.spliterator(), false).filter( uer ->
uer.getEntityType().equals( "ORG" ) && uer.getRoleName().equals( role ) )

            .map( ent -> ent.getEntityId() ).collect( Collectors.toList() ).contains( entityId ) ;

            break;

     int siteResultSize= StreamSupport.stream(siteList.spliterator(), false).filter( siteResult->{

        boolean siteName = false,orgId = false;

        if(parameters.containsKey( "name" )){

           siteName=siteResult.getName().contains( parameters.get( "name" ) );

        }

        else

           siteName=true;

        if(parameters.containsKey( "orgid" )){

           orgId=siteResult.getOrganizationId().equals( parameters.get( "orgid" ) );

        }

        else

           orgId=true;

        return siteName && orgId;

     }).collect( toList() ).size();
```

```
        public boolean verifyV2RolesInResponse( String response,
com.ge.hc.pfh.shared.automation.entity.request.V2Application applicationRequest ) throws
UomAutomationException {

application = responseParser.jsonStringToObject( response, V2Application.class );

List<V2AppRoles> applicationRoles = application.getApplicationRoles();

List< V2AppRoles> rolesInRequest = applicationRequest.getRoles();

return StreamSupport.stream( applicationRoles.spliterator(), false ).map( role -> role.getCode() ).collect(
Collectors.toList() ).containsAll( StreamSupport.stream( rolesInRequest.spliterator(), false ).map( roles -
> roles.getCode() ).collect( Collectors.toList() ) );


        }
```

**Method References:**

Method references help to point to methods by their names. A method reference is described using "::" symbol. A method reference can be used to point the following types of methods –

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

```
        List names = new ArrayList();

        names.add("Mahesh");

        names.add("Suresh");

        names.forEach(System.out::println); //static method
```

```
Vector<String> empNames = empList.stream()

            .map(Employee::getName)    //instance method

            .collect(Collectors.toCollection(Vector::new)); // Constructors
```

Here we have passed System.out::println method as a static method reference.

**Optional class:**

One of the most interesting features that Java 8 introduces to the language is the new Optional class. The main issue this class is intended to tackle is the infamous NullPointerException that every Java programmer knows only too well.

Essentially, this is a wrapper class that contains an optional value, meaning it can either contain an object or it can simply be empty.

```
Optional<User> opt = Optional.ofNullable(user);

   String name = "John";
```

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



    assertEquals("John", opt.get());
```