Java 8 features
----

1)Functional Interfaces

A functional interface is an interface that has exactly one abstract method. Java 8 introduced the @FunctionalInterface annotation to define a functional interface.

```
@FunctionalInterface
interface GreetingService {
   void sayMessage(String message);
}

public class FunctionalInterfaceExample {
   public static void main(String[] args) {
      // Lambda expression implementing the sayMessage method
      GreetingService greet = message -> System.out.println("Hello, " + message);
      greet.sayMessage("Ramesh");
   }
}
```

2) Default and Static Methods in Interfaces

Java 8 introduced default and static methods in interfaces, allowing developers to add new methods to interfaces without breaking existing implementations.

```
interface Vehicle {
   // Default method
   default void print() {
      System.out.println("I am a vehicle");
   }

   // Static method
   static void blowHorn() {
      System.out.println("Blowing horn");
   }
}
```

Multiple Inheritance of Behavior

```
public interface I1 {

        public void m1(int a,int b);
        default void m2() {
                System.out.println("i1 m2");
        }

        static void m3() {

        }
}


public interface I2 {

        default void m2() {
                System.out.println("i2 m2");
        }

}


public class C1 implements I1,I2{

        @Override
        public void m2() {
                I2.super.m2();
                I1.super.m2();
        }
}
```

3) Lambda Expressions
A lambda expression is simply a function without a name. It can even be used as a parameter in a function
syntax: (parameters) -> { statements; }

Lambda expression provides an implementation of the Java 8 Functional Interface. An interface that has only one abstract method is called a functional interface.

```
public interface I1 {

        public void m1();
}
```

```
public static void main(String[] args) {
                I1 obj = ()->{
                        System.out.println("hello");
                };

                obj.m1();
        }
```

We can discuss Runnable and Comparator

```
Collections.sort(listOfPerson, (Person o1, Person o2) -> {
        return o1.getAge() - o2.getAge();
     });
```

4) Method Reference

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of the functional interface. It is a compact and easy form of a lambda expression.

Reference to a static method

Example:

ContainingClass::staticMethodName

Reference to an instance method of a particular object

Example:

containingObject::instanceMethodName

Reference to a constructor

Example:

ClassName::new

Constructor

```
public class ReferenceToConstructor {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}

interface Messageable{
    Message getMessage(String msg);
}

class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
```

5) Parallel Array Sorting

This algorithm offers O(n log(n)) performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.

```
 public static void main(String[] args) {
        int[] numbers = {5, 3, 8, 1, 9, 4, 7, 6, 2, 0};

        System.out.println("Before Sorting: " + Arrays.toString(numbers));

        // Parallel sort
        Arrays.parallelSort(numbers);

        System.out.println("After Sorting: " + Arrays.toString(numbers));
    }
```

6)Pre-defined Functional Interfaces

a)UnaryOperator   ---> Same input and output

```
UnaryOperator<Integer> u1 = (x)->x*x;
UnaryOperator<Integer> u2 = (x)->x+2;

System.out.println(u1.andThen(u2).apply(2));
```

b)BinaryOperator --> same two inputs

```
BinaryOperator<Integer> b1 = (x,y)->x+y;
System.out.println(b1.apply(100, 200));
```

c)Function
d)Bi-Function
e)Predicate --> which takes any input and returns boolean

f)Supplier --> T get()
g)Consumer<T> ----> void accept(T t)

7)Stream Api
----

methods in Stream api

a)forEach(Consumer action)

```
List<Integer> list = Arrays.asList(10,20,30,40,50);
list.stream().forEach(System.out::println);
```

note: forEach(consumer) is there in List and forEach(BiConsumer) is there in Map

b)filter(Predicate p)

```
List<Integer> list = Arrays.asList(10,20,30,40,50);

list.stream().filter(x->x>15).forEach(System.out::println);
```

c)map(Function) changes the values

```
list.stream().map(x->x*2).forEach(System.out::println);
```

d)flatMap(function) which converts List of List to list

```
List<Integer> list = Arrays.asList(10,20,30,40,50);
List<Integer> l1 = Arrays.asList(1,2,3,4,5);
List<Integer> l2 = Arrays.asList(-1,-2,-3,-4,-5);
List<List<Integer>> list1 = Arrays.asList(list,l1,l2);
list1.stream().flatMap(List::stream).collect(Collectors.toList()).forEach(System.out::println);;
```

e)Sort Elements in a Stream

The sorted() method sorts the elements of the stream.

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
numbers.stream().sorted().forEach(System.out::println);
```

f)count

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
long count = numbers.stream().filter(x->x<0).count();
System.out.println(count);
```

g)Limit the Stream Size

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
numbers.stream().limit(3).forEach(System.out::println);
```

h)Skip Elements in a Stream

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
numbers.stream().skip(3).forEach(System.out::println);
```

i)Find the First Element in a Stream (below program to get max 3rd element)

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
int num = numbers.stream().sorted().skip(2).findFirst().get();
System.out.println(num);
```

j)Check if Any Match in a Stream

```
List<Integer> numbers = Arrays.asList(10, -2, 3, 4, 5, 6);
boolean result = numbers.stream().anyMatch(x->x==5);
System.out.println(result);
```

Terminal Operators in java 8
—-----------------------------------

1)The `anyMatch()` method checks if any element in the stream matches the given predicate. If any match is found, it returns `true`; otherwise, `false`.

```
List<Integer> list = Arrays.asList(10, 20, 3, 15, 6);
boolean result = list.stream().anyMatch(x -> x % 2 == 0);
```

2) `allMatch()`

The `allMatch()` method checks if all elements in the stream match the given predicate. It returns `true` only if all elements satisfy the predicate.

```
List<Integer> list = Arrays.asList(10, 20, 30, 150, 6);
boolean result = list.stream().allMatch(x -> x % 2 == 0);
```

3. noneMatch()
The noneMatch() method checks if no elements in the stream match the given predicate. It returns true if none of the elements satisfy the condition.

4.collect()
The collect() method is used to collect the stream elements into a collection, such as a List, Set, or Map.

5.count()
The count() method returns the number of elements in the stream.

6. findFirst()
The findFirst() method retrieves the first element in the stream.

7. forEach()
The forEach() method performs an action for each element in the stream. It is typically used for printing or other side effects.

8. min()
The min() method returns the smallest element from the stream based on the specified comparator.

9. max()
The max() method returns the largest element from the stream based on the specified comparator.

10. reduce()
The reduce() method combines elements in the stream into a single value based on a binary operation.

```
List<Integer> list = Arrays.asList(10,20,30,40);
int result = list.stream().reduce((x,y)->x-y).get();
System.out.println(result);
```

8)Collectors in java8

The Collectors class in Java 8 is part of the java.util.stream package and provides various utility methods to collect the results of stream operations. It's primarily used with the Stream.collect() method to convert a stream into a different form, such as a List, Set, Map, or even a concatenated String. Collectors simplify tasks like grouping, partitioning, and reducing data from streams.

```
Collectors.toList()
Collectors.toSet()
List<Integer> list = Arrays.asList(10,20,30,40,50);
List<Integer> l1 = Arrays.asList(10,2,3,4,50);
List<Integer> l2 = Arrays.asList(-1,-2,-3,-4,-5);
List<List<Integer>> list1 = Arrays.asList(list,l1,l2);
Set<Integer> set =  list1.stream().flatMap(List::stream).collect(Collectors.toSet());
System.out.println(set);
```

```
Collectors.toMap()
List<String> list = Arrays.asList("one","two","three","four");
Map<String, Integer>map =  list.stream().collect(Collectors.toMap(x->x, x->x.length()));
```

```
System.out.println(map);
```

Joining Elements into a String

```
List<String> list = Arrays.asList("one","two","three","four");
String result = list.stream().collect(Collectors.joining());
String result2 = list.stream().collect(Collectors.joining(","));
System.out.println(result+"   "+result2);
```

Summing Elements

You can sum the numeric elements of a stream using Collectors.summingInt(), summingLong(), or summingDouble().

```
List<Integer> list = Arrays.asList(10,20,30,40);
int result = list.stream().collect(Collectors.summingInt(x->x));
System.out.println(result);
List<Double> list = Arrays.asList(10.1,20.2,30.4,40.0);
double result = list.stream().collect(Collectors.summingDouble(x->x));
System.out.println(result);
```

Averaging Values

To calculate the average of elements, you can use Collectors.averagingInt(), averagingLong(), or averagingDouble().

```
List<Double> list = Arrays.asList(10.1,20.2,30.4,40.0);
double result = list.stream().collect(Collectors.averagingDouble(x->x));
System.out.println(result);
```

Grouping Elements by a Key

You can use Collectors.groupingBy() to group elements of a stream by a specific key.

```
List<Integer> list = Arrays.asList(10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26);
Map<Integer, List<Integer>>map = list.stream().collect(Collectors.groupingBy(x->x%10));
System.out.println(map);
```

Partitioning Elements by a Predicate

You can partition elements into two groups based on a predicate using Collectors.partitioningBy().

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
        Map<Boolean, List<Integer>> partitioned =
numbers.stream().collect(Collectors.partitioningBy(num -> num % 2 == 0));
```

```
    System.out.println(partitioned);
```

Counting Elements

To count the number of elements in a stream, use Collectors.counting().

```
List<Integer> list = Arrays.asList(10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26);
Long count = list.stream().collect(Collectors.counting());
System.out.println(count+"   "+list.size());
```

IntStream
—----------

IntStream class is a specialization of Stream interface for int primitive. It represents a stream of primitive int-valued elements supporting sequential and parallel aggregate operations.

IntStream is part of the java.util.stream package and implements AutoCloseable and BaseStream interfaces.

```
IntStream is1 = IntStream.of(10);
is1= IntStream.of(10,20,30);
```

Multiple IntStream values

```
IntStream is = IntStream.range(1, 100);
                //is.forEach(System.out::println);
                is.forEach(x->{
                        IntStream is2 = IntStream.range(1, x+1);
                        //System.out.println(x);
                        is2.forEach(y->{
                                System.out.print("*");
                        });
                        System.out.println();
                });
```

range(int st, int end)--> returns starting and ending(exclude) values

rangeClosed(int st, int end)-->returns with end values(include) also

```
IntStream is = IntStream.range(1, 5);
is.forEach(System.out::println);

IntStream is2 = IntStream.rangeClosed(1, 5);
is2.forEach(System.out::println);
```

iterator function is useful for creating infinite streams.

```
IntStream is = IntStream.iterate(1, x->x+2).limit(5);
is.forEach(System.out::println);
```

IntSummaryStatistics class gives aggregate details

```
IntStream is = IntStream.of(10,20,30,40);
IntSummaryStatistics iss = is.summaryStatistics();
System.out.println(iss);
```

Optional in java8

Java introduced a new class Optional in JDK 8. It is a public final class and is used to deal with NullPointerException in Java applications.

```
static void displayString(Optional<String> st) {
        System.out.println(st.orElse("not theree"));

        st.ifPresentOrElse(x->System.out.println(x), ()->System.out.println("not there"));

        st.orElseGet(()->"not theree");//supplier method

    }

    public static void main(String[] args) {

        Optional<String>st= Optional.ofNullable("abc");

        displayString(st);
    }
```

Example programs

1. Find Even Numbers from a List
2. Convert List of Strings to Uppercase
3. Find Duplicate Elements
4. Find First Element Greater Than 10
5. Sort List in Descending Order
6. Sum of All Numbers
7. Count Occurrences of Each Word
8. Find Maximum Number

9. Merge Two Lists and Remove Duplicates
10. Reverse Sort Strings by Length
11. Square each number in the list
12. Join all strings into one String in the list
List("one","two","three")   --> "one,two,three"

13. group by length
List("one","two","three","eight")   --> {3:["one","two"],5:["three","eight"]}

14. find min value in the list

15. find second max value from the list

16. find frequency of characters
"banana" -->{"b":1,"a":3,"n":2}

17. Palindrem check
"madam" true
"sir" false

18. Sort given numbers from the list

19. Remove duplicate numbers from list

20. Given a String, find the first repeated character in it using Stream functions?