JAVA

8 Features

MARCH 15, 2024

JAVA FULLSTACK GURU
Hyderabad

Java 1.8v Features

Java 1.0

Java 1.1

Java 1.2   (Collection Framework)

..

Java 1.5 (Big Release)

..

Java 1.8 (Big Release) ------- Functional Programming

..

Java 19

Java 1.8v Features

-> Java 1.8v introduced lot of new features in java

-> Java 1.8v new features changed java programming style

Main Objectivies of Java 1.8v

-> Simplify Java Programming

-> Enable Functional Programming

-> Write more readable and consice code

Java 1.8 Features

1) Interface changes

    1.1) Default Methods

1.2) Static Methods

2) Functional Interfaces (@FunctionalInterface)

2.1) Predicate & BiPredicate

2.2) Consumer & BiConsumer

2.3) Supplier

2.4) Function & BiFunction

3) Lambda Expressions

4) Method References & Constructor References

5) ****** Stream API ********

6) Optional class (to avoid null pointer exceptions)

7) Spliterator

8) StringJoiner

9) forEach ( ) method

10) Date & Time API

11) Nashron Engine

12) I/O Stream Changes (Files.lines(Path p))

13) Base64 Encoding & Decoding

Interface changes

-> Interface means collection of abstract methods

Note: The method which doesn't contain body is called as abstract method

-> A class can implement interface using "implements"

-> When a class is implementing interface its mandatory that class should implement all abstract methods of that interface othewise class can't be compile.

=> Here i am taking one interface with one abstract method. All the classes which are implementing that interface should overide interface method(s).

```
interface Vehicle {
        public abstract void startVechicle ( );
}
class Car implements Vehicle {
        public void startVehicle ( ) {
                // logic to start car
        }
}
class Bus implements Vehicle {
        public void startVehicle ( ) {
                // logic to start  bus
        }
}
class Bike implements Vehicle {
        public void startVehicle ( ) {
                // logic to start  bike
```

```
    }
}
```

=> If we add new method in interface then Car, Bike and Bus will fail at compile time.

=> To overcome above problem we will use Default & Static methods

1) Interface can have concreate methods from 1.8v

2) Interface concrete method should be default or static

3) interface default methods we can override in impl classes

4) interface static methods we can't overide in impl classes

5) We can write multiple default & static methods in interface

6) Default & Static method introduced to provide backward compatability

Ex: forEach ( ) method added in java.util.Iterable interface as default method in 1.8v

```
package in.ashokit;
interface Vehicle {
    public void start();
    public default void m1() {
    }
    public default void m2() {
    }
    public static void clean() {
        System.out.println("cleaning completed...");
        }
}
public class Car implements Vehicle {
        public void start() {
                System.out.println("car started...");
        }
        public static void main(String[] args) {
                Car c = new Car();
                Vehicle.clean();
                c.start();
        }
}
```

Lambda Expressions

-> Introdced in java 1.8v

-> Java is called as Object Oriented Programming language. Everything will be represented using Classes and Objects.

-> From 1.8v onwards Java is also called as Functional Programming Language.

-> In OOP language Classes & Objects are main entities. We need to write methods inside the class only.

-> Functional Programming means everything will be represented in the form functions. Functions can exist outside of the class. Functions can be stored into a reference variable. A function can be passed as a parameter to other methods.

-> Lambda Expressions introduced in Java to enable Functional Programming.

What is Lambda

-> Lambda is an anonymous function

- No Name
- No Modifier
- No Return Type

Ex:-1

```
public void m1 ( ) {
        s.o.p("hi");
}
 ( ) -> { s.o.p ("hi") }
```

Note: When we have single line in body then curly braces are optional

```
 ( ) -> s.o.p ("hi");
```

Ex:-2

```
public void add (int a, int b){
        s.o.p(a+b);
}
( int a, int b)  -> { s.o.p (a+b) } ;
```

(or)

```
(int a, int b) -> s.o.p (a+b);
```

(or)

Lambda Expression : (a, b) -> s.o.p(a+b);

Ex:-3

```
public int getLength (String name) {
        return name.length ( );
}
(String name) -> { return name.length ( ) };
(String name) -> return name.length ( ) ;
(name) -> return name.length ( );
```

Lambda Expression : name -> name.length ( ) ;

Ex:-4

```
public Double getEmpSalary (Employee emp) {
        return emp.getSalary ( );
}
```

Lambda Expression : emp -> emp.getSalary ( );

Functional Interfaces

-> The interface which contains only one abstract method is called as Functional Interface

-> Functional Interfaces are used to invoke Lambda expressions

-> Below are some predefined functional interfaces

```
                        Runnable ------------> run ( ) method
                        Callable ----------> call ( ) method
                        Comparable -------> compareTo ( )
```

-> To represent one interface as Functional Interface we will use @FunctionalInterface annotation.

```
@FunctionalInterface
public interface MyInterface {
        public void m1( );
}
```

Note: When we write @FunctionalInterface then our compiler will check interface contains only one abstract method or not.

-> In Java 8 several predefined Functional interfaces got introduced they are

1) Predicate & BiPredicate
2) Consumer & BiConsumer
3) Supplier
4) Function & BiFunction

-> The above interfaces are provided in java.util.function package

Predicate

-> It is predefined Functional interface

-> It is used check condition and returns true or false value

-> Predicate interface having only one abstract method that is test (T t)

```
                        interface Predicate{
                                boolean test(T t);
```

```
                    }


// Predicate Example
package in.ashokit.java8;
import java.util.function.Predicate;
public class PredicateDemo {
        public static void main(String[] args) {
                Predicate<Integer> p = i -> i > 10;
                System.out.println(p.test(5));
                System.out.println(p.test(15));
        }
}
```

Task: Declare names in an array and print names which are starting with 'A' using lambda expression.

```
String[ ]  names = {"Anushka", "Anupama", "Deepika", "Kajol",
"Sunny" };
```

```
package in.ashokit.java8;
import java.util.function.Predicate;
public class PredicateDemo2 {
public static void main(String[] args) {
String[ ] names = { "Anushka", "Anupama", "Deepika", "Kajol",
"Sunny" };
Predicate<String> p = name -> name.charAt(0) == 'A';
for (String name : names) {
        if ( p.test(name) ) {
                System.out.println(name);
            }
        }
    }
}
```

Task-2 : Take list of persons and print persons whose age is >= 18 using Lambda Expression

```
package in.ashokit.java8;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class PredicatePersonsDemo {
public static void main(String[] args) {
        Person p1 = new Person("John", 26);
        Person p2 = new Person("Smith", 16);
        Person p3 = new Person("Raja", 36);
        Person p4 = new Person("Rani", 6);
        List<Person> persons = Arrays.asList(p1, p2, p3, p4);
        Predicate<Person> predicate = p -> p.age >= 18;
        for (Person person : persons) {
            if (predicate.test(person)) {
                System.out.println(person.name);
            }
        }
    }
}
```

Predicate Joining

-> To combine multiple predicates we will use Predicate Joining

and ( ) method

or ( ) method

Task-1 : Print emp names who are working in Hyd location in DB team.

```
package in.ashokit.java8;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
class Employee {
        String name;
        String location;
        String dept;
        Employee(String name, String location, String dept) {
                this.name = name;
                this.location = location;
                this.dept = dept;
        }
}
public class PredicateJoinDemo {
public static void main(String[] args) {
Employee e1 = new Employee("Anil", "Chennai", "DevOps");
Employee e2 = new Employee("Rani", "Pune", "Networking");
Employee e3 = new Employee("Ashok", "Hyd", "DB");
Employee e4 = new Employee("Ganesh", "Hyd", "DB");
List<Employee> emps = Arrays.asList(e1, e2, e3, e4);
Predicate<Employee> p1 = (e) -> e.location.equals("Hyd");
Predicate<Employee> p2 = (e) -> e.dept.equals("DB");
Predicate<Employee> p3 = (e) -> e.name.startsWith("A");

                // Predicate Joining
Predicate<Employee> p = p1.and(p2).and(p3);
            for (Employee e : emps) {
                  if (p.test(e)) {
                        System.out.println(e.name);
                  }
            }
      }
}
```

Supplier Functional Interface

-> Supplier is a predefined functional interface introduced in java 1.8v

-> It contains only one abstract method that is get ( ) method

-> Supplier interface will not take any input, it will only returns the value.

Ex:

OTP Generation

```
package in.ashokit.java8;
import java.util.function.Supplier;
public class SupplierDemo {
public static void main(String[] args) {
Supplier<String> s = () -> {
                  String otp = "";
                  for (int i = 1; i <= 6; i++) {
                    otp = otp + (int) (Math.random() * 10);
                    }
                    return otp;
                };
                System.out.println(s.get());
                System.out.println(s.get());
                System.out.println(s.get());
                System.out.println(s.get());
                System.out.println(s.get());
                System.out.println(s.get());
        }
}
```

Consumer Functional Interface

-> Consumer is predefined functional interface

-> It contains one abstract method i.e accept (T t)

-> Consumer will accept input but it won't return anything

Note: in java 8 forEach ( ) method got introduced. forEach(Consumer consumer) method will take Consumer as parameter.

```
package in.ashokit.java8;

import java.util.Arrays;

import java.util.List;

import java.util.function.Consumer;
```

```
public class ConsumerDemo {
        public static void main(String[] args) {
Consumer<String> c = (name) -> System.out.println(name + ", Good Evening");
c.accept("Ashok");
c.accept("John");
c.accept("Rani");
List<Integer> numbers = Arrays.asList(10, 20, 30, 40);
                // for loop
                // for each loop
                // iterator
                // list iterator
                numbers.forEach(i -> System.out.println(i));
        }
}
```

Retrieve student record based on student id and return that record

Predicate ------> takes inputs ----> returns true or false ===> test ( )

Supplier -----> will not take any input---> returns output ===> get ( )

Consumer ----> will take input ----> will not return anything ===> accept ( )

Function -----> will take input ---> will return output ===> apply ( )

Function Functional Interface

-> Function is predefined functional interface

-> Funcation interface having one abstract method i.e apply(T r)

```
        interface Function<R,T>{
                R apply (T t);
        }
```

-> It takes input and it returns output

```
package in.ashokit.java8;
import java.util.function.Function;
public class FunctionDemo {
public static void main(String[] args) {
Function<String, Integer> f = (name) -> name.length();
        System.out.println(f.apply("ashokit"));
        System.out.println(f.apply("hyd"));
        System.out.println(f.apply("sachin"));

    }
}
```

Task : Take 2 inputs and perform sum of two inputs and return ouput

```
BiFunction<Integer,Integer,Integer> bif = (a,b) -> a+b;
Integer sum = bi.apply(10,20);
```

Method References

-> Method reference means Reference to one method from another method

```
package in.ashokit.java8;

@FunctionalInterface

interface MyInterface {

        public void m1();

}

public class MethodRef {

        public static void m2() {

                System.out.println("This is m2( ) method");

        }

        public static void main(String[] args) {

                MyInterface mi = MethodRef::m2;

                mi.m1();

        }

}

package in.ashokit.java8;

public class InstanceMethodRef {

        public void m1() {

                for (int i = 1; i <= 5; i++) {

                        System.out.println(i);

                }
}
    public static void main(String[] args) {
        InstanceMethodRef im = new InstanceMethodRef();
        Runnable r = im::m1;
        Thread t = new Thread(r);
        t.start();
    }
}
public class Test {
    public static void main(String[] args) {
        // Doctor d = new Doctor();
        Supplier<Doctor> s = Doctor::new;
        Doctor doctor = s.get();
        System.out.println(doctor.hashCode());

    }

}
class Doctor {
    public Doctor() {
        System.out.println("Doctor constructor....");
    }
}
```

Task : WAJP to print numbers from 1 to 5 using Thread with the help of Runnable interface

```
//Approach-1
public class ThreadDemo1 implements Runnable {
        @Override
        public void run() {
                for (int i = 1; i <= 5; i++) {
                        System.out.println(i);
                }
        }

        public static void main(String[] args) {
                ThreadDemo1 td = new ThreadDemo1();
                Thread t = new Thread(td);
                t.start();
        }
}
package in.ashokit.java8;
// Approach-2
public class ThreadDemo2 {
        public static void main(String[] args) {
                Runnable r = new Runnable() {
                        @Override
```

```
            public void run() {
                for (int i = 1; i <= 5; i++) {
                    System.out.println(i);
                }
            }
        };
        Thread t = new Thread(r);
        t.start();
    }
}
// Approach - 3 using Lambda Expression
package in.ashokit.java8;
public class ThreadDemo3 {
    public static void main(String[] args) {
        Runnable r = () -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println(i);
            }
        };

        Thread t = new Thread(r);
        t.start();
    }
```

```
}
```

Task: WAJP to store numbers in ArrayList and sort numbers in desending order

```
// Approach-1 ( without Lambda)

package in.ashokit.java8;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;


public class NumbersSort1 {

        public static void main(String[] args) {

                ArrayList<Integer> al = new ArrayList<>();

                al.add(5);

                al.add(3);

                al.add(4);

                al.add(1);

                al.add(2);

                System.out.println("Before Sort :: " + al);

                Collections.sort(al, new NumberComparator());

                System.out.println("After Sort :: " + al);

        }

}

class NumberComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer i, Integer j) {
        if (i > j) {
            return -1;
        } else if (i < j) {
            return 1;
        }
        return 0;
    }
}
// Approach-2 ( with Lambda)
package in.ashokit.java8;
import java.util.ArrayList;
import java.util.Collections;

public class NumbersSort1 {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<>();
        al.add(5);
        al.add(3);
        al.add(4);
        al.add(1);
        al.add(2);
        System.out.println("Before Sort :: " + al);
        Collections.sort(al, (i, j) -> (i > j) ? -1 : 1);
        System.out.println("After Sort :: " + al);
    }
}
```

forEach (Consumer c) method

-> forEach (Consumer c) method introduced in java 1.8v

-> forEach ( ) method added in Iterable interface

-> forEach ( ) method is a default method (it is having body)

-> This is method is used to access each element of the collection (traverse collection from start to end)

```
package in.ashokit.java8;
import java.util.ArrayList;
public class NumbersSort1 {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<>();
        al.add(5);
        al.add(3);
        al.add(4);
        al.add(1);
        al.add(2);
        al.forEach(i -> System.out.println(i));
        }
}
```

StringJoiner

-> java.util.StringJoiner class introduced in java 1.8v

-> It is used to join more than one String with specified delimiter

-> We can concat prefix and suffix while joininging strings using StringJoiner

```
                StringJoiner sj = new StringJoiner (CharSequence delim);
                StringJoiner sj = new StringJoiner (CharSequence delim,
CharSequence prefix, CharSequence suffix);
```

```
package in.ashokit.java8;
import java.util.StringJoiner;
public class StringJoinerDemo {
        public static void main(String[] args) {
                StringJoiner sj1 = new StringJoiner("-");
                sj1.add("ashok");
                sj1.add("it");
                sj1.add("java");
                System.out.println(sj1); // ashok-it-java
                StringJoiner sj2 = new StringJoiner("-", "(", ")");
                sj2.add("ashok");
                sj2.add("it");
                sj2.add("java");
System.out.println(sj2); // (ashok-it-java)
        }
}
```

Optional Class

-> java.util.Optional class introduced in java 1.8v

-> Optional class is used to avoid NullPointerExceptions in the program

Q) What is NullPointerException (NPE) ?

Ans) When we perform some operation on null value then we will get NullPointerException

```
String s = null;
s.length ( ) ;  // NPE
```

-> To avoid NullPointerExceptions we have to implement null check before performing operation on the Object like below.

```
String s = null;
if( s! = null ) {
        System.out.println(s.length ( ));
}
```

Note: In project there is no gaurantee that every programmer will implement null checks. If any body forgot to implement null check then program will run into NullPointerException.

-> To avoid this problem we need to use Optional class like below.

```
package in.ashokit.java8;
import java.util.Optional;
public class User {
        // Without Optional object
        public String getUsernameById(Integer id) {
                if (id == 100) {
                        return "Raju";
                } else if (id == 101) {
                        return "Rani";
                } else if (id == 102) {
                        return "John";
                } else {
                        return null;
                }
        }
        // with Optional Object
        public Optional<String> getUsername(Integer id) {
                String name = null;
                if (id == 100) {
                        name = "Raju";
                } else if (id == 101) {
                        name = "Rani";
                } else if (id == 102) {
```

```
            name = "John";
        }
        return Optional.ofNullable(name);
    }
}
package in.ashokit.java8;
import java.util.Optional;
import java.util.Scanner;
public class MsgService {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter User ID");
        int userId = s.nextInt();
        User u = new User();
        /*String userName = u.getUsernameById(userId);
        String msg = userName.toUpperCase() + ", Hello";
        System.out.println(msg);*/
        Optional<String> username = u.getUsername(userId);
        if(username.isPresent()) {
            String name = username.get();
            System.out.println(name.toUpperCase()+", Hello");
        }else {
            System.out.println("No Data Found");
```

```
                }
        }
}
```

Date & Time API Changes

-> In java we have below 2 classes to represent Date

1) java.util.Date
2) java.sql.Date

Note: When we are performing database operations then we will use java.sql.Date class.

-> For normal Date related operations we will use java.util.Date class

```
Date d = new Date ( );
System.out.prinln(d);
```

Note: When we create Object for Date class, it will represent both date and time.

-> If we want to get only date or only time then we need to format it using SimpleDateFormat class.

java.text.SimpleDateFormat

-> SimpleDateFormat is a predefined class in java.text pacakage

-> This class provided methods to perform Date conversions

Date to String conversion ===> String format (Date d)

String to Date conversion ===> Date parse(String str)

```
// Date Conversions Example

package in.ashokit.java8;
```

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class DateDemo {
	public static void main(String[] args) throws Exception {
		Date date = new Date();
		System.out.println(date);
		// Converting Date to String
		SimpleDateFormat sdf1 = new SimpleDateFormat("dd/MM/yyyy");
		String format1 = sdf1.format(date);
		System.out.println(format1);
		SimpleDateFormat sdf2 = new SimpleDateFormat("MM/dd/yyyy");
		String format2 = sdf2.format(date);
		System.out.println(format2);
		// Convert String to Date
		SimpleDateFormat sdf3 = new SimpleDateFormat("yyyy-MM-dd");
		Date parsedDate = sdf3.parse("2022-12-20");
		System.out.println(parsedDate);


	}
}
```

=> To overcome the problems of java.util.Date class java 1.8 introduced Date API changes

=> In java 1.8 version, new classes got introduced to deal with Date & Time functionalities

1) java.time.LocalDate (it will deal with only date)

2) java.time.LocalTime (it will deal with only time)

3) java.time.LocalDateTime (it will deal with both date & time)

Java 1.8 Date API Example

```
package in.ashokit.java8;

import java.time.Duration;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

import java.time.Period;

public class NewDateDemo {

        public static void main(String[] args) {

                LocalDate of = LocalDate.of(2021, 1, 20);

                System.out.println(of);

                LocalDate date = LocalDate.now();

                System.out.println(date);

                date = date.plusDays(3);
                System.out.println(date);
                date = date.plusMonths(1);
                System.out.println(date);
                date = date.plusYears(2);
                System.out.println(date);
        boolean leapYear = LocalDate.parse("2020-12-22").isLeapYear();
        System.out.println("Leap Year :: " + leapYear);
boolean before = LocalDate.parse("2021-12-22").isBefore(LocalDate.parse("2022-12-22"));
System.out.println("Before Date : " + before);
                LocalTime time = LocalTime.now();
                System.out.println(time);
                time = time.plusHours(2);
                System.out.println(time);
                LocalDateTime datetime = LocalDateTime.now();
                System.out.println(datetime);
                Period period = Period.between(LocalDate.parse("1991-05-20"), LocalDate.now());
System.out.println(period);
Duration duration = Duration.between(LocalTime.parse("18:00"), LocalTime.now());
System.out.println(duration);
        }
}
```

1) What are new changes in java 8 version

2) Interface Changes

    2.1) Default Methods

    2.2) Static Methods

3) Why Default & Static method introduced in java 8

4) Lambda Expressions Introduction

5) How to write Lambda Expression

6) How to invoke lambda expression

7) Functional Interfaces

    7.1) Predicate & BiPredicate

    7.2) Supplier

    7.3) Consumer & BiConsumer

    7.4) Function & BiFunction

8) Collections Sorting using Lambda

9) Thread Creation Using Lambda

10) Method References & Constructor References

11) java.util.StringJoiner class

12) java.utils.Optional class

13) forEach (Consumer c) method

14) Date & Time API Changes

    14.1) LocalDate

    14.2) LocalTime

    14.3) LocalDateTime

14.4) Period

14.5) Duration

Stream API

-> Stream API introduced in java 1.8v

-> Stream API is used to process the data

Note: Collections are used to store the data

-> Stream API is one of the major features added in java 1.8v

-> Stream in java can be defined as sequence of elements that comes from a source.

-> Source of data for the Stream can be array or collection

Few Important Points About Streams

1) Stream is not a data structure. Stream means bunch of operations applied on source data. Source can be collection or array.

2) Stream will not change original data structure of the source (It will just process the data given by the source.)

Stream Creation

-> In Java we can create Stream in 2 ways

1) Stream.of (e1, e2, e3, e4.....)

2) stream ( ) method

Java Program to Create Stream

```
package in.ashokit.streams;

import java.util.ArrayList;
import java.util.stream.Stream;
public class FirstDemo {
public static void main(String[] args) {
                // Approach-1
                Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5);
                ArrayList<String> names = new ArrayList<>();
                names.add("John");
                names.add("Robert");
                names.add("Orlen");
                // Approach-2
                Stream<String> stream2 = names.stream();
        }
}
```

Stream Operations

-> Stream API provided several methods to perform Operations on the data

-> We can divide Stream api methods into 2 types

1) Intermediate Operational Methods
2) Terminal Operational Methods

-> Intermediate Operational methods will perform operations on the stream and returns a new Stream

Ex: filter ( ) , map ( ) etc....

-> Terminal Operational methods will take input and will provide result as output.

Ex: count ( )

Filtering with Streams

-> Filtering means getting required data from original data

Ex: get only even numbers from given numbers

Ex: get emps whose salary is >= 1,00,000

Ex: Get Mobiles whose price is <= 15,000

-> To apply filter on the data, Stream api provided filter ( ) method

Ex : Stream filter (Predicate p)

Example - 1 : Filter

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
public class FirstDemo {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(66, 32, 45, 12, 20);
        /*for (Integer i : list) {
            if (i > 20) {
                System.out.println(i);
            }
        }*/
        /*Stream<Integer> stream = list.stream();
        Stream<Integer> filteredStrem = stream.filter(i -> i > 20);
        filteredStrem.forEach(i -> System.out.println(i));*/
        list.stream().filter(i -> i > 20).forEach(i -> System.out.println(i));
        }
}
```

Example - 2 : Filter

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
public class FirstDemo {
        public static void main(String[] args) {
List<String> names = Arrays.asList("John", "Anushka", "Anupama", "Smith", "Ashok");
names.stream().filter(i -> i.startsWith("A")).forEach(i -> System.out.println(i));
        }
}
```

==================

Example - 3 : Filter

==================

```
package in.ashokit.streams;
import java.util.stream.Stream;
public class FirstDemo {
        public static void main(String[] args) {
                User u1 = new User("Anushka", 25);
                User u2 = new User("Smith", 30);
                User u3 = new User("Raju", 15);
                User u4 = new User("Rani", 10);
                User u5 = new User("Charles", 35);
                User u6 = new User("Ashok", 30);
                Stream<User> stream = Stream.of(u1, u2, u3, u4, u5, u6);
// stream.filter(u -> u.age >= 18).forEach(u -> System.out.println(u));
/*stream.filter(u -> u.age >= 18 && u.name.startsWith("A"))
                                    .forEach(u -> System.out.println(u));*/
                stream.filter(u -> u.age >= 18)
                        .filter(u -> u.name.startsWith("A"))
                        .forEach(u -> System.out.println(u));
        }
}
class User {
        String name;
        int age;
        User(String name, int age) {
                this.name = name;
                this.age = age;
        }
public String toString() {
```

```
                return "User [name=" + name + ", age=" + age + "]";
        }
}
```

====================

Mapping Operations

====================

-> Mapping operations are belongs to intermediate operations in the Stream api

-> Mapping operations are used to transform the stream elements and return transformed elements as new Stream

Ex : Stream map (Function function) ;

========================

Example-1 : map ( ) method

========================

```
public class FirstDemo {
public static void main(String[] args) {
        List<String> names = Arrays.asList("india","usa","uk", "japan");
                /*for(String name : names) {
                        System.out.println(name.toUpperCase());
                }*/


names.stream().map(name -> name.toUpperCase()).forEach(n ->
System.out.println(n));
names.stream().mapToInt(name -> name.length()).forEach(i ->
System.out.println(i));


    }

}
```

==========================

Example-2 : map ( ) method

=========================

```
public class FirstDemo {

public static void main(String[] args) {

List<String> names = Arrays.asList("Ashok", "Anil", "Raju", "Rani",
"John", "Akash", "Charles");

// print name with its length which are starting with 'A' using Stream API

                        //Ashok - 5

                        //Anil  - 4

                        //Akash - 5

        names.stream()

            .filter(name -> name.startsWith("A"))

            .map(name -> name + "-" +name.length())

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

    }

}
```

=======================

Example-3 : map ( ) method

========================

```
class Employee ( ) {
    String name;
    int age;
    double salary;
}
```

Task : Print Emp Name with Emp age whose salary is >= 50,000 using Stream API.

```
public class FirstDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee("John", 35, 55000.00);
        Employee e2 = new Employee("David", 25, 45000.00);
        Employee e3 = new Employee("Buttler", 35, 35000.00);
        Employee e4 = new Employee("Steve", 45, 65000.00);
        Stream<Employee> stream = Stream.of(e1, e2, e3, e4);
        /*stream.filter(e -> e.salary >= 50000.00)
             .map(e -> e.name+" - " +e.age)
             .forEach(e -> System.out.println(e));*/
        stream.filter(e -> e.salary >= 50000.00)
          .forEach(e -> System.out.println(e.name + "-" + e.age));
        }
}
class Employee {
        String name;
        int age;
        double salary;
        public Employee(String name, int age, double salary) {
                this.name = name;
                this.age = age;
                this.salary = salary;
        }
}
```

==

Q) What is flatMap(Function f) method ?

==

-> It is used to flaten list of streams into single stream

```
public class FirstDemo {
        public static void main(String[] args) {
List<String> javacourses = Arrays.asList("core java", "adv java",
"springboot");
List<String> uicourses = Arrays.asList("html", "css", "bs", "js");
List<List<String>> courses = Arrays.asList(javacourses, uicourses);
//courses.stream().forEach(c -> System.out.println(c));
Stream<String> fms = courses.stream().flatMap(s -> s.stream());
fms.forEach(c -> System.out.println(c));
        }
}
```

==========================

Slicing Operations with Stream

==========================

1) distinct ( ) => To get unique elements from the Stream

2) limit (long maxSize) => Get elements from the stream based on given size

3) skip (long n) => It is used to skip given number of elements from starting position of the stream

Note: All the above 3 methods are comes under Intermediate Operational Methods. They will perform operation and returns new Stream.

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
public class FirstDemo {
        public static void main(String[] args) {
List<String> javacourses = Arrays.asList("corejava", "advjava",
"springboot", "restapi", "microservices");
javacourses.stream().limit(3).forEach(c -> System.out.println(c));
javacourses.stream().skip(3).forEach(c -> System.out.println(c));
List<String> names = Arrays.asList("raja", "rani", "raja", "rani",
"guru");
names.stream().distinct().forEach(name -> System.out.println(name));
        }
}
```

==============================

Matching Operations with Stream

==============================

1) boolean anyMatch (Predicate p)
2) boolean allMatch (Predicate p)
3) boolean noneMatch (Predicate p)

Note: The above 3 methods are belongs to Terminal Operations because they will do operation and they will return result directley (they won't return stream)

-> The above methods are used to check the given condition and returns true or false value based on condition.

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
public class FirstDemo {
        public static void main(String[] args) {
                Person p1 = new Person("John", "USA");
                Person p2 = new Person("Steve", "JAPAN");
                Person p3 = new Person("Ashok", "INDIA");
                Person p4 = new Person("Ching", "CHINA");
                List<Person> persons = Arrays.asList(p1, p2, p3, p4);
boolean status1 = persons.stream().anyMatch(p ->
p.country.equals("INDIA"));
System.out.println("Any Indian Available ? :: " + status1);
boolean status2 = persons.stream().anyMatch(p ->
p.country.equals("CANADA"));
System.out.println("Any Canadian Available ? :: " + status2);
boolean status3 = persons.stream().allMatch(p ->
p.country.equals("INDIA"));
System.out.println("All Persons from India ? :: " + status3);
boolean status4 = persons.stream().noneMatch(p ->
p.country.equals("MEXICO"));
System.out.println("No Persons from Mexico ? :: " + status4);
        }
}
class Person {
        String name;
        String country;
        public Person(String name, String country) {
                this.name = name;
                this.country = country;
        }  }
```

====================

Collectors with Stream

===================

-> Collectors are used to collect data from Stream

====================

Example-1 : Collectors

====================

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class FirstDemo {
        public static void main(String[] args) {
                Person p1 = new Person("John", "USA");
                Person p2 = new Person("Steve", "JAPAN");
                Person p3 = new Person("Ashok", "INDIA");
                Person p4 = new Person("Ching", "CHINA");
                Person p5 = new Person("Kumar", "INDIA");
                List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5);
                List<Person> indians = persons.stream().filter(p ->
p.country.equals("INDIA")).collect(Collectors.toList());
indians.forEach(i -> System.out.println(i)); } }
```

```
class Person {
	String name;
	String country;
	public Person(String name, String country) {
		this.name = name;
		this.country = country;
	}
	@Override
	public String toString() {
		return "Person [name=" + name + ", country=" + country + "]";
	}
}
```

===================

Example-2: Collectors

===================

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class FirstDemo {
	public static void main(String[] args) {
        Person p1 = new Person("John", "USA");
        Person p2 = new Person("Steve", "JAPAN");
        Person p3 = new Person("Ashok", "INDIA");
        Person p4 = new Person("Ching", "CHINA");
        Person p5 = new Person("Kumar", "INDIA");
        List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5);
        // collect names of persons who are belongs to india and store
into names collection
        List<String> names = persons.stream()
        .filter(p -> p.country.equals("INDIA")).map(p -> p.name)
        .collect(Collectors.toList());
        System.out.println(names);
    }
}
class Person {
    String name;
    String country;
    public Person(String name, String country) {
        this.name = name;
        this.country = country;
    }
    @Override
    public String toString() {
```

```
return "Person [name=" + name + ", country=" + country + "]";
        }
}
```

===

Set - 1 : Intermediate Operations   (will return Stream)

===

Filters ----> filter ( )

Mappings ----> map ( ) & flatMap ( )

Slicing   ----> distinct ( ) & limit () & skip ( )

===

Set - 2 : Terminal Operations (will return result)

===

Finding   ---> findFirst ( ) & findAny ( )

Matching ---> anyMatch ( ) & allMatch ( ) & noneMatch ( )

Collecting ---> collect ( )

============

Requirement

===========

=> Write a java program to get MAX, MIN and AVG salary from given employees data using Stream API.

```
package in.ashokit.streams;

import java.util.Arrays;

import java.util.Comparator;
import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

public class FirstDemo {

        public static void main(String[] args) {

                Employee e1 = new Employee(1, "Robert", 26500.00);

                Employee e2 = new Employee(2, "Abraham", 46500.00);

                Employee e3 = new Employee(3, "Ching", 36500.00);

                Employee e4 = new Employee(4, "David", 16500.00);

                Employee e5 = new Employee(5, "Cathy", 25500.00);

                List<Employee> list = Arrays.asList(e1, e2, e3, e4, e5);

Optional<Employee> max =
list.stream().collect(Collectors.maxBy(Comparator.comparing(e ->
e.salary)));

System.out.println("Max Salary :: " + max.get().salary);

Optional<Employee> min =
list.stream().collect(Collectors.minBy(Comparator.comparing(e ->
e.salary)));

System.out.println("Min Salary :: " + min.get().salary);

                Double avgSalary =
list.stream().collect(Collectors.averagingDouble(e -> e.salary));

                System.out.println(avgSalary);

        }

}

class Employee {
	int id;
	String name;
	double salary;
	public Employee(int id, String name, double salary) {
		this.id = id;
		this.name = name;
		this.salary = salary;
	}
}
```

Group By using Stream

-> Group By is used categorize the data / Grouping the data

-> When we use groupingBy ( ) function with stream they it will group the data as Key-Value(s) pair and it will return Map object

-> In below example employees will be grouped based on Country name.

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
public class FirstDemo {
public static void main(String[] args) {
```

```
Employee e1 = new Employee(1, "Robert", 26500.00, "USA");
Employee e2 = new Employee(2, "Abraham", 46500.00, "INDIA");
Employee e3 = new Employee(3, "Ching", 36500.00, "CHINA");
Employee e4 = new Employee(4, "David", 16500.00, "INDIA");
Employee e5 = new Employee(5, "Cathy", 25500.00, "USA") ;

List<Employee> list = Arrays.asList(e1, e2, e3, e4, e5);
Map<String, List<Employee>> data = list.stream()
.collect(Collectors.groupingBy(e -> e.country));
System.out.println(data);
    }
}
class Employee {
    int id;
    String name;
    double salary;
    String country;
public Employee(int id, String name, double salary, String country) {
        this.id = id;
        this.name = name;
        this.salary = salary;
        this.country = country;
    }
```

```
}
```

Parallel Streams

-> Generally Streams will execute in sequence order

-> To improve execution process of the stream we can use parallel streams

-> Paralell Streams introduced to improve performance of the program.

```
package in.ashokit.streams;
import java.util.stream.Stream;
public class ParallelDemo {
public static void main(String[] args) {
System.out.println("====== Serial Steam ========");
Stream<Integer> ss = Stream.of(1, 2, 3, 4);
ss.forEach(n -> System.out.println(n + " :: " + Thread.currentThread()));
System.out.println("====== Parallel Strem =======");
Stream<Integer> ps = Stream.of(1, 2, 3, 4);
ps.parallel().forEach(n -> System.out.println(n + " :: " +
Thread.currentThread()));
        }
}
```

Java Spliterator

-> Like Iterator and ListIterator, Spliterator is one of the Java Iterator

-> Spliterator introduced in java 1.8v

-> Spliterator is an interface in collections api

-> Spliterator supports both serial & paralell programming

-> Spliterator we can use to traverse both Collections & Streams

-> Spliterator can't be used with Map implementation classes

```
package in.ashokit.streams;
import java.util.Arrays;
import java.util.List;
import java.util.Spliterator;
public class ParallelDemo {
        public static void main(String[] args) {
        List<String> names = Arrays.asList("sachin", "sehwag", "dhoni");
        Spliterator<String> spliterator = names.stream().spliterator();
        spliterator.forEachRemaining(n -> System.out.println(n));
        }
}
```

Stream Reduce

```
package demo;
import java.util.Arrays;
public class Sum {
        public static void main(String[] args) {
                int[] nums = { 1, 2, 3, 4, 5 };
                /*int sum = 0;
                for(int i : nums) {
```

```
sum = sum + i;
                }
                System.out.println(sum);*/
                int reduce = Arrays.stream(nums).reduce(0, (a,b) -> a+b);
                System.out.println(reduce);
        }
}
```

Nashorn Engine in Java 1.8

-> Nashorn is a Java Script Engine which is used to execute Java Script code using JVM

-> Create a javascript file like below (filename : one.js)

```
-------------------- one.js -------------------------
var hello = function(){
        print("Welcome to JavaScript");
}
hello();
```

-> Open command prompt and execute below command

```
        syntax :  jjs one.js
```

-> We can execute above Java Script file using Java program like below

```
import java.io.*;
import javax.script.*;
```

```
public class Demo {
        public static void main(String... args) throws Exception {
        ScriptEngine se = new 
ScriptEngineManager().getEngineByName("Nashorn");
                se.eval(new FileReader("one.js"));
        }
}
```

==========================

I/O Streams Changes in Java 8

==========================

Task : Write a java program to read a file data and print it on the console

-> To read file data we can use FileReader & BufferedReader classesFileReader ----> It will read the data character by character (slow performance)

BufferedReader ---> It will read the data line by line

Files.lines(Path path) ---> It will read all lines at a time and returns as a Stream

```
package demo;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;
public class ReadFileData {
        public static void main(String[] args) throws Exception {
```

```
		/*FileReader fr = new FileReader(new File("info.txt"));
		BufferedReader br = new BufferedReader(fr);
		String line = br.readLine();
		while (line != null) {
			System.out.println(line);
			line = br.readLine();
		}
		br.close();*/
		String filename = "info.txt";
		try (Stream<String> stream = Files.lines(Paths.get(filename))){
	stream.forEach(line -> System.out.println(line));
		}catch(Exception e) {
		e.printStackTrace();
		}
	}
}
```

=======================

Java 8 Base64 Changes

=======================

-> Base64 is a predefined class available in java.util package

-> Base64 class providing methods to perform encoding and decoding

```
	Encoder encoder = Base64.getEncoder();
```

```
// converting String to byte[] and passing as input for encode( ) method
            byte[] encode = encoder.encode(pwd.getBytes());
            // Converting byte[] to String
            String encodedPwd = new String(encode);
            System.out.println(encodedPwd);
            Decoder decoder = Base64.getDecoder();
            byte[ ] decode = decoder.decode(encodedPwd);
            String decodedPwd = new String(decode);
            System.out.println(decodedPwd);
```