**java.util.stream**

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

To Process objects of java collection We use iterator, enumerators.

From Java 1.8 onwards we can use streams concept.

So from Java 1.8 onwards there are two stream packages:

1. java.io for input and output through data streams

2. java.util.stream stream of objects from the collection objects.

java.io package provides classes and interfaces for processing character and binary data with respect to file and devices.

Where java.util.stream package provides classes and interfaces to support functional style operations on streams of elements.

What is the difference between collection and stream?

ans:

If we want to represent a group of individual objects as a single object then we will use collection objects.

If we want to process a group of objects from the collection obj then we will go for stream objects.

How can we get stream object?

ans:

We can get Stream obj(java.util.stream.Stream interface impl class object) by using any one of the following methods:

1. default Stream<E> stream()

2. default Stream<E> parallelStream()

Once we got this stream by using that stream we can process objects of the collection.

We can process the objects in the following two phases:

1) configuration

2) processing

we can do configuration by filter mech or by map mech.

Filtering:

we can configure a filter to filter elements from the collection based on some boolean condition.

To do this we use filter() method of stream interface.

Stream<T> filter(Predicate<? super T> predicate)

List<Resouce> resources = .....;

Stream<Resource> rstream = resources.stream();

Stream<Resource> adminsStream = rstream.filter( (r) -> r.getRole().contains("admin") );

**Mapping**

If we want to create a separate new object for every object present in the collection depending on our requirement then we should go for map() method of stream interface.

<R> Stream<R> map(Function<? super T,? extends R> mapper)

Example:

List<Resouce> resources = .....;

Stream<Resource> rstream1 = resources.stream();

Stream<String> resourcesNames = rstream1.map(r -> r.getName());

**Processing**

Once we are done with configuration. we can process objects by using several methods as shown below:

Processing by collect() method

Processing by count() method

Processing by sorted() method

Processing by min() and max() methods

Processing by forEach() method

Processing by toArray() method

Processing by Stream.of() method

**Terminal and Non-Terminal Operations**

**Non-Terminal Operations(configuration)**

filter()

map()

flatMap()

distinct()

limit()

peek()

**Terminal Operations(processing)**

anyMatch()

allMatch()

noneMatch()

collect()

count()

findAny()

findFirst()

forEach()

min()

max()

reduce()

toArray()

**Non Terminal Operations:**

The non terminal operations are used to transform or filter the elements in the stream.

Whenever we perform a non-terminal operation to a stream we will get a new stream back.

**filter:**

is used to filter out elements from a stream.

the filter method takes predicate as parameters and uses that predicate to filter out.

**map:**

is used to convert an object in the stream a different obj.

**flatMap:**

is used to map a single element/object to multiple elements/objects

**distinct:**

is used return a new stream which will contain only the distinct elements from the original stream.

that means if any duplicates are there those will be removed.

**limit:**

is used to return a new stream which will limit the number of elements to a number given as parameter to this method.

**peak:**

this peak methods takes consumer as argument.

it will call the consumer on each element in the stream and create a new stream returns it.

examples:

**Resource.java**

**package** com.satya.java8.stream;

**import** java.util.List;

**public** **class** Resource {

**private** **int** id;

**private** String name;

**private** String role;

**private** List<String> projects;

**private** **double** salary;

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** List<String> getProjects() {

**return** projects;

}

**public** **void** setProjects(List<String> projects) {

**this**.projects = projects;

}

**public** String getRole() {

**return** role;

}

**public** **void** setRole(String role) {

**this**.role = role;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Resource() {

}

**public** Resource(**int** id, String name, String role, List<String> projects, **double** salary) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.role = role;

**this**.projects = projects;

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Resource [id=" + id + ", name=" + name + ", role=" + role + ", projects=" + projects + ", salary="

+ salary + "]";

}

}

**NonTerminalOpsTestCase1.java**

**package** com.satya.java8.stream;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** NonTerminalOpsTestCase1 {

**public** **static** **void** main(String[] args) {

List<Resource> resources = **new** ArrayList<Resource>();

resources.add(**new** Resource(1, "yogita", "tester", Arrays.*asList*("proj1", "proj2", "proj3"), 100));

resources.add(**new** Resource(2, "bunny", "techwriter", Arrays.*asList*("proj5", "proj1"), 200));

resources.add(**new** Resource(3, "sunny", "network admin", Arrays.*asList*("proj4", "proj1", "proj3"), 500));

resources.add(**new** Resource(5, "ashok", "systems admin", Arrays.*asList*("proj3", "proj1"), 500));

resources.add(**new** Resource(6, "vinod", "lead", Arrays.*asList*("proj1"), 600));

resources.add(**new** Resource(7, "bond", "architect", **new** ArrayList<String>(), 700));

resources.add(**new** Resource(9, "baburao", "architect", **new** ArrayList<String>(), 700.0));

Stream<Resource> rstream = resources.stream();

Stream<Resource> adminsStream = rstream.filter((r) -> r.getRole().contains("admin"));

// rstream.forEach((r)->System.out.println(r));

System.***out***.println("---------- filtered stream");

// adminsStream.forEach((r)->System.out.println(r));

adminsStream.forEach(System.***out***::println);

System.***out***.println("---------- map");

Stream<Resource> rstream1 = resources.stream();

Stream<String> resourcesNames = rstream1.map(r -> r.getName());

resourcesNames.forEach(System.***out***::println);

System.***out***.println("----------------- flat map");

Stream<Resource> rstream2 = resources.stream();

Stream<String> rs = rstream2.flatMap(r -> r.getProjects().stream());

rs.forEach(System.***out***::println);

System.***out***.println("-------------- distinct");

Stream<Resource> rstream3 = resources.stream();

Stream<String> rs1 = rstream3.flatMap(r -> r.getProjects().stream());

Stream<String> distinct = rs1.distinct();

distinct.forEach(System.***out***::println);

System.***out***.println("-------------- limit");

Stream<Resource> rstream4 = resources.stream();

Stream<String> rs2 = rstream4.flatMap(r -> r.getProjects().stream());

Stream<String> limit = rs2.limit(3);

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

System.***out***.println("-------------- peek");

Stream<Resource> rstream5 = resources.stream();

Stream<Resource> peek = rstream5.peek(r -> r.setRole(r.getRole()+", emp of infosys"));

peek.forEach(System.***out***::println);

}

}

**Terminal Operations(processing)**

anyMatch()

allMatch()

noneMatch()

collect()

count()

findAny()

findFirst()

forEach()

min()

max()

reduce()

toArray()

**TerminalOpsTestCase.java**

**package** com.satya.java8.stream;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.Optional;

**import** java.util.stream.Collectors;

**import** java.util.stream.Stream;

**public** **class** TerminalOpsTestCase {

**public** **static** **void** main(String[] args) {

List<Resource> resources = **new** ArrayList<Resource>();

resources.add(**new** Resource(1, "yogita", "tester", Arrays.*asList*("proj1", "proj2", "proj3"), 100));

resources.add(**new** Resource(2, "bunny", "techwriter", Arrays.*asList*("proj5", "proj1"), 200));

resources.add(**new** Resource(3, "sunny", "network admin", Arrays.*asList*("proj4", "proj1", "proj3"), 500));

resources.add(**new** Resource(5, "ashok", "systems admin", Arrays.*asList*("proj3", "proj1"), 500));

resources.add(**new** Resource(6, "vinod", "lead", Arrays.*asList*("proj1"), 600));

resources.add(**new** Resource(7, "bond", "architect", **new** ArrayList<String>(), 700));

resources.add(**new** Resource(9, "baburao", "architect", **new** ArrayList<String>(), 700.0));

// anymatch

Stream<Resource> stream = resources.stream();

**if** (stream.anyMatch(r -> r.getName().contains("ashok"))) {

System.***out***.println("yes we have ashok here");

} **else** {

System.***out***.println("we do not have ahsok here");

}

// all match

Stream<Resource> stream2 = resources.stream();

**if** (stream2.allMatch(r -> (r.getProjects() != **null** && r.getProjects().size() != 0))) {

System.***out***.println("all resources have projects");

} **else** {

System.***out***.println("few resources are not having projects");

}

// non match

Stream<Resource> stream3 = resources.stream();

**if** (stream3.noneMatch(r -> (r.getProjects() == **null** && r.getProjects().size() == 0))) {

System.***out***.println("few resources are not having projects");

} **else** {

System.***out***.println("all resources have projects");

}

//collect

Stream<Resource> stream4 = resources.stream();

List<Resource> collectedList = stream4.collect(Collectors.*toList*());

System.***out***.println(collectedList);

Stream<Resource> stream5 = resources.stream();

// count

System.***out***.println(stream5.count());

//findany

Stream<Resource> stream6 = resources.parallelStream();

Stream<Resource> stream7 = stream6.filter(r -> r.getRole().equals("architect"));

// The Java Stream findAny() method can find a single element from the Stream.

Optional<Resource> optional1 = stream7.findAny();

**if** (optional1.isPresent()) {

Resource r = optional1.get();

System.***out***.println(r);

} **else** {

System.***out***.println("no such architect");

}

// find first

Stream<Resource> stream8 = resources.parallelStream();

Stream<Resource> stream9 = stream8.filter(r -> r.getRole().equals("architect"));

Optional<Resource> optional3 = stream9.findFirst();

**if** (optional3.isPresent()) {

System.***out***.println(optional3.get());

}

// for each

Stream<Resource> stream10 = resources.stream();

stream10.forEach(r -> r.setId(r.getId() + 100));

System.***out***.println(resources);

// min

Stream<Resource> stream11 = resources.stream();

Optional<Resource> optional = stream11.min((r1, r2) -> (**int**) (r1.getSalary() - r2.getSalary()));

**if** (optional.isPresent()) {

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

}

// max

Stream<Resource> stream12 = resources.stream();

Optional<Resource> optional4 = stream12.max((r1, r2) -> (**int**) (r1.getSalary() - r2.getSalary()));

**if** (optional4.isPresent()) {

System.***out***.println(optional4.get());

}

// reduce

Stream<Resource> stream13 = resources.stream();

List<String> rolesList = **new** ArrayList<String>();

stream13.forEach(r -> rolesList.add(r.getRole()));

Stream<String> rolesStream = rolesList.stream();

Optional<String> reduced = rolesStream.reduce((r1, r2) -> r1.concat("," + r2));

**if** (reduced.isPresent()) {

System.***out***.println("reduced to single string:");

System.***out***.println(reduced.get());

}

// toArray

Stream<Resource> stream14 = resources.stream();

Object a[] = stream14.toArray();

System.***out***.println("toArray-------------");

**for** (Object e : a)

System.***out***.println(e);

Stream<String> stream15 = Stream.*of*("satya", "raju", "bunny");

stream15.forEach(System.***out***::println);

Stream<String> stream16 = Stream.*of*("ravi");

stream16.forEach(System.***out***::println);

}

}

**ParallelStream concept**

SteamsParallel.java

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** SteamsParallel {

**public** **static** **void** main(String[] args) {

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

**for** (**int** i = 1; i < 10; i++) {

list.add(i);

}

// Here creating a parallel stream

Stream<Integer> stream = list.parallelStream();

Integer[] evenNumbersArr = stream.filter(i -> i % 2 == 0).toArray(Integer[]::**new**);

**for**(**int** i=0;i<evenNumbersArr.length;i++)

System.***out***.print(evenNumbersArr[i]+",");

}

}

SteamsParallel2.java

/\*

The only thing to keep in mind to create parallel stream is to call the parallelStream() method on the collection else by default the sequential stream gets returned by stream() method.

1.1.1 Parallel Streams Performance Implications

Parallel Stream has equal performance impacts as like its advantages.

Since each sub-stream is a single thread running and acting on the data, it has overhead compared to the sequential stream

Inter-thread communication is dangerous and takes time for coordination

1.2 When to use Parallel Streams?

They should be used when the output of the operation is not needed to be dependent on the order of elements present in source collection (i.e. on which the stream is created)

Parallel Streams can be used in case of aggregate functions

Parallel Streams quickly iterate over the large-sized collections

Parallel Streams can be used if developers have performance implications with the Sequential Streams

If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in

Now, open up the Eclipse Ide and I will explain further the parallel streams in Java8 programming

\*/

**import** java.util.ArrayList;

**import** java.util.List;

**class** Employee{

**private** String name;

**private** **int** salary;

**public** Employee() {

}

**public** Employee(String name, **int** salary) {

**this**.name = name;

**this**.salary = salary;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

}

**public** **class** SteamsParallel2 {

**public** **static** **void** main(String[] args) {

**long** t1, t2;

List<Employee> eList = **new** ArrayList<Employee>();

**for**(**int** i=0; i<100; i++) {

eList.add(**new** Employee("A", 20000));

eList.add(**new** Employee("B", 3000));

eList.add(**new** Employee("C", 15002));

eList.add(**new** Employee("D", 7856));

eList.add(**new** Employee("E", 200));

eList.add(**new** Employee("F", 50000));

}

/\*\*\*\*\* Here We Are Creating A 'Sequential Stream' & Displaying The Result \*\*\*\*\*/

t1 = System.*currentTimeMillis*();

System.***out***.println("Sequential Stream Count?= " + eList.stream().filter(e -> e.getSalary() > 15000).count());

t2 = System.*currentTimeMillis*();

System.***out***.println("Sequential Stream Time Taken?= " + (t2-t1) + "\n");

/\*\*\*\*\* Here We Are Creating A 'Parallel Stream' & Displaying The Result \*\*\*\*\*/

t1 = System.*currentTimeMillis*();

System.***out***.println("Parallel Stream Count?= " + eList.parallelStream().filter(e -> e.getSalary() > 15000).count());

t2 = System.*currentTimeMillis*();

System.***out***.println("Parallel Stream Time Taken?= " + (t2-t1));

}

}

**Additional Examples:**

**Streams.java**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**import** java.util.stream.Stream;

**public** **class** Streams {

**public** **static** **void** main(String[] args) {

List<String> l1 = **new** ArrayList<>();

l1.add("Amitabh");

l1.add("Shekhar");

l1.add("Aman");

List<String> l2 = **new** ArrayList<>();

l2.add("Rahul");

l2.add("Shahrukh");

List<String> l3 = **new** ArrayList<>();

l3.add("Salman");

l3.add("Yana");

l3.add("Lokesh");

List<List<String>> ll = **new** ArrayList<List<String>>();

ll.add(l1);

ll.add(l2);

ll.add(l3);

Stream<List<String>> stream = ll.parallelStream();

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

Stream<String> st = stream.flatMap(List::stream);

// st.forEach(System.out::println);

List<String> newList = st.collect(Collectors.*toList*());

// List<String> newList =

//

System.***out***.println(newList);

}

}

Streams1.java

**import** java.util.stream.Stream;

//Stream.of(val1, val2, val3….)

**public** **class** Streams1 {

**public** **static** **void** main(String[] args) {

Stream<Integer> stream = Stream.*of*(1, 2, 3, 4, 5, 6, 7, 8, 9);

stream.forEach(p -> System.***out***.print(p));

System.***out***.println("");

Stream<Integer> stream2 = Stream.*of*(1, 2, 3, 4, 5, 6, 7, 8, 9);

stream2.forEach(System.***out***::print);

}

}

Streams2.java

**import** java.util.stream.Stream;

//2. Stream.of(arrayOfElements)

**public** **class** Streams2 {

**public** **static** **void** main(String[] args)

{

Stream<Integer> stream = Stream.*of*( **new** Integer[]{1,2,3,4,5,6,7,8,9} );

stream.forEach(p -> System.***out***.println(p));

}

}

**Streams3.java**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

//3. List.stream()

**public** **class** Streams3 {

**public** **static** **void** main(String[] args)

{

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

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

stream.forEach(p -> System.***out***.println(p));

}

}

**Stream4.java**

**import** java.util.Date;

**import** java.util.stream.Stream;

//4. Stream.generate() or Stream.iterate()

**public** **class** Stream4 {

**public** **static** **void** main(String[] args)

{

Stream<Date> stream ;

stream = Stream.*generate*(() -> { **return** **new** Date(); }).limit(5);

stream.forEach(p -> System.***out***.println(p));

}

}

Stream5.java

**import** java.util.stream.IntStream;

**import** java.util.stream.Stream;

//5. String chars or String tokens

**public** **class** Stream5 {

**public** **static** **void** main(String[] args)

{

IntStream stream = "12345\_aBCdefg".chars();

stream.forEach(p -> System.***out***.println(p));

System.***out***.println("=============");

//OR

Stream<String> stream2 = Stream.*of*("A$B$C".split("\\$"));

stream2.forEach(p -> System.***out***.println(p));

}

}

StreamConvert1.java

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

//1. Convert Stream to List – Stream.collect( Collectors.toList() )

**public** **class** StreamConvert1 {

**public** **static** **void** main(String[] args){

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

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

System.***out***.println(list);

List<Integer> newList = **new** ArrayList<Integer>();

/\*for(Integer e:list) {

int i=e;

if(i%2==0)

newList.add(i);

}

System.out.println(newList);

for(int i=0;i<list.size();i++) {

int e=list.get(i);

if(e%2==0)

newList.add(e);

}

System.out.println(newList);

\*/

Iterator<Integer> it = list.iterator();

**while**(it.hasNext()) {

**int** e = it.next();

**if**(e%2==0)

newList.add(e);

}

System.***out***.println(newList);

/\*Stream<Integer> stream = list.stream();

List<Integer> evenNumbersList;

evenNumbersList = stream.filter(i -> i%2 == 0).collect(Collectors.toList());

System.out.print(evenNumbersList);

\*/

}

}

StreamConvert2.java

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

//2. Convert Stream to array – Stream.toArray( EntryType[]::new )

**public** **class** StreamConvert2 {

**public** **static** **void** main(String[] args){

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

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::**new**);

**for**(Integer e:evenNumbersArr)

System.***out***.print(e+",");

}

}

StreamInterOP.java

**import** java.util.ArrayList;

**import** java.util.List;

/\*

Intermediate operations

Intermediate operations return the stream itself so you can chain multiple method calls in a row.

1.1. Stream.filter()

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach) on the result.

memberNames.stream().filter((s) -> s.startsWith("A"))

.forEach(System.out::println);

1.2. Stream.map()

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type.

memberNames.stream().filter((s) -> s.startsWith("A"))

.map(String::toUpperCase)

.forEach(System.out::println);

1.3. Stream.sorted()

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

memberNames.stream().sorted()

.map(String::toUpperCase)

.forEach(System.out::println);

\*/

**public** **class** StreamInterOP {

**public** **static** **void** main(String[] args) {

List<String> memberNames = **new** ArrayList<>();

memberNames.add("Amitabh");

memberNames.add("Shekhar");

memberNames.add("Aman");

memberNames.add("Rahul");

memberNames.add("Shahrukh");

memberNames.add("Salman");

memberNames.add("Yana");

memberNames.add("Lokesh");

System.***out***.println("+++++++++++++++++++++++");

memberNames.stream().filter((s) -> s.startsWith("A"))

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

System.***out***.println("+++++++++++++++++++++++");

memberNames.stream().filter((s) -> s.startsWith("A"))

.map(String::toUpperCase)

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

System.***out***.println("+++++++++++++++++++++++");

memberNames.stream().sorted()

.map(String::toUpperCase)

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

}

}

StreamShortCircuit.java

**import** java.util.ArrayList;

**import** java.util.List;

/\*

Stream short-circuit operations

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

Though, stream operations are performed on all elements inside a collection satisfying a predicate, It is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, you will do with if-else block. In internal iteration, there are certain methods you can use for this purpose. Let’s see example of two such methods:

1. Stream.anyMatch()

This will return true once a condition passed as predicate satisfy. It will not process any more elements.

boolean matched = memberNames.stream()

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

System.out.println(matched);

Output: true

2. Stream.findFirst()

It will return first element from stream and then will not process any more element.

String firstMatchedName = memberNames.stream()

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

.findFirst().get();

System.out.println(firstMatchedName);

Output: Lokesh

\*/

**public** **class** StreamShortCircuit {

**public** **static** **void** main(String[] args) {

List<String> memberNames = **new** ArrayList<>();

memberNames.add("Amitabh");

memberNames.add("Shekhar");

memberNames.add("Aman");

memberNames.add("Rahul");

memberNames.add("Shahrukh");

memberNames.add("Salman");

memberNames.add("Yana");

memberNames.add("Lokesh");

**boolean** matched = memberNames.stream().anyMatch((s) -> s.startsWith("A"));

System.***out***.println(matched);

String firstMatchedName = memberNames.stream().filter((s) -> s.startsWith("L")).findFirst().get();

System.***out***.println(firstMatchedName);

}

}

StreamTerminalOp.java

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Optional;

**import** java.util.stream.Collectors;

/\*

Terminal operations

Terminal operations return a result of a certain type instead of again a Stream.

1. Stream.forEach()

This method helps in iterating over all elements of a stream and perform some operation on each of them. The operation is passed as lambda expression parameter.

memberNames.forEach(System.out::println);

2. Stream.collect()

collect() method used to receive elements from a steam and store them in a collection and mentioned in parameter function.

List<String> memNamesInUppercase = memberNames.stream().sorted()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.print(memNamesInUppercase);

3. Stream.match()

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

boolean matchedResult = memberNames.stream()

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

System.out.println(matchedResult);

matchedResult = memberNames.stream()

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

System.out.println(matchedResult);

matchedResult = memberNames.stream()

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

System.out.println(matchedResult);

4. Stream.count()

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

long totalMatched = memberNames.stream()

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

.count();

5. Stream.reduce()

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

Optional<String> reduced = memberNames.stream()

.reduce((s1,s2) -> s1 + "#" + s2);

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

\*/

**public** **class** StreamTerminalOp {

**public** **static** **void** main(String[] args) {

List<String> memberNames = **new** ArrayList<>();

memberNames.add("Amitabh");

memberNames.add("Shekhar");

memberNames.add("Aman");

memberNames.add("Rahul");

memberNames.add("Shahrukh");

memberNames.add("Salman");

memberNames.add("Yana");

memberNames.add("Lokesh");

memberNames.forEach(System.***out***::println);

List<String> memNamesInUppercase = memberNames.stream().sorted().map(String::toUpperCase)

.collect(Collectors.*toList*());

System.***out***.print(memNamesInUppercase);

**boolean** matchedResult = memberNames.stream().anyMatch((s) -> s.startsWith("A"));

System.***out***.println(matchedResult);

matchedResult = memberNames.stream().allMatch((s) -> s.startsWith("A"));

System.***out***.println(matchedResult);

matchedResult = memberNames.stream().noneMatch((s) -> s.startsWith("A"));

System.***out***.println(matchedResult);

**long** totalMatched = memberNames.stream().filter((s) -> s.startsWith("A")).count();

System.***out***.println(totalMatched);

Optional<String> reduced = memberNames.stream().reduce((s1, s2) -> s1 + "#" + s2);

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

}

}

**StringIntersection.java**

**public** **class** StringIntersection {

**public** **static** String printDiffChars(**char** a[], **char** b[]) {

String s = "";

**boolean** charIsPresent = **false**;

**for** (**int** i = 0; i < a.length; i++) {

charIsPresent = **false**;

**for** (**int** j = 0; j < b.length; j++) {

**if** (a[i] == b[j]) {

charIsPresent = **true**;

**break**;

}

}

**if** (charIsPresent == **false**)

s = s + a[i];

}

**for** (**int** i = 0; i < b.length; i++) {

charIsPresent = **false**;

**for** (**int** j = 0; j < a.length; j++) {

**if** (b[i] == a[j]) {

charIsPresent = **true**;

**break**;

}

}

**if** (charIsPresent == **false**)

s = s + b[i];

}

**return** s;

}

**public** **static** **void** main(String[] args) {

String s1="abzycdxf";

String s2="bceadmn";

String intection = *printDiffChars*(s1.toCharArray(),s2.toCharArray());

System.***out***.println(intection);

}

}

DateTimeDemo1.java

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**import** java.time.LocalTime;

**import** java.time.Month;

**import** java.time.Period;

**import** java.time.Year;

**import** java.time.ZoneId;

**import** java.time.ZonedDateTime;

**public** **class** DateTimeDemo1{

**public** **static** **void** main(String[] args){

LocalDate date = LocalDate.*now*();

System.***out***.println(date);

System.***out***.println(date.getDayOfMonth());

System.***out***.println(date.getMonth());

System.***out***.println(date.getYear());

LocalTime time = LocalTime.*now*();

System.***out***.println(time);

System.***out***.println(time.getHour());

System.***out***.println(time.getMinute());

System.***out***.println(time.getSecond());

System.***out***.println(time.getNano());

LocalDateTime localDateTime = LocalDateTime.*now*();

System.***out***.println(localDateTime);

LocalDateTime dateTime = LocalDateTime.*of*(1990, Month.***FEBRUARY***,5,8,20);

System.***out***.println(dateTime);

ZoneId zone = ZoneId.*systemDefault*();

System.***out***.println(zone);

ZoneId zoneDetroit = ZoneId.*of*("US/Michigan");

ZonedDateTime zoneDateTime = ZonedDateTime.*now*(zoneDetroit);

System.***out***.println("date time of US/Michigan ");

System.***out***.println(zoneDateTime);

LocalDate today = LocalDate.*now*();

LocalDate birthDay = LocalDate.*of*(1990, 02, 05);

Period p = Period.*between*(birthDay, today);

System.***out***.println(p.getYears());

System.***out***.println(p.getMonths());

System.***out***.println(p.getDays());

**int** y=1980;

Year year = Year.*of*(y);

**if**(year.isLeap())

System.***out***.println("is leap year");

**else**

System.***out***.println("is not leap year");

}

}