Zero to Pro Bootcamp – Level 3

Contents

Myanmar IT Consulting

3.1 Lambda Expression

- Less code
- To provide implementation of functional interface

```
(parameters) -> {statements;}
(or)
 (parameters) -> expression
```

Example – Lambda Expression with Functional Interface

```
@FunctionalInterface
interface MyInterface1{
        void display(String name);
}
interface MyInterface2{
        int addNumber(int n1,int n2);
}
interface MyInterface3{
        boolean checkLogin(String uname,String pass);
}

MyInterface1 test1 = (str)-> System.out.println("Name is " + str);
MyInterface3 test3 = (name,pass)->{
                        if(name.equals("jeon") && pass.equals("123"))
                                return true;
                        else
                                return false;
                };
MyInterface2 sum = (a,b) -> a + b;
MyInterface2 mul = (a,b) -> a * b;
MyInterface2 div = (a,b) -> a / b;

System.out.println("sum: " + sum.operate(200, 50));
System.out.println("mul: " + mul.operate(200, 50));
System.out.println("div: " + div.operate(200, 50));

test1.display("Cherry");
System.out.println((test3.checkLogin("jeon", "123")) ? "Login Success" : "Invalid Login");
System.out.println((test3.checkLogin("abc", "123")) ? "Login Success" : "Invalid Login");
```

Example 2 – Lambda Expression with Collection API

```
List<String> languages = List.of("HTML","JavaScript","Css","Java","Php");

languages.forEach(lang->System.out.println(lang));
```

```
System.out.println("------------------");
languages.forEach(lang->{
        if(lang.contains("Java"))
                System.out.println(lang);
        });
```

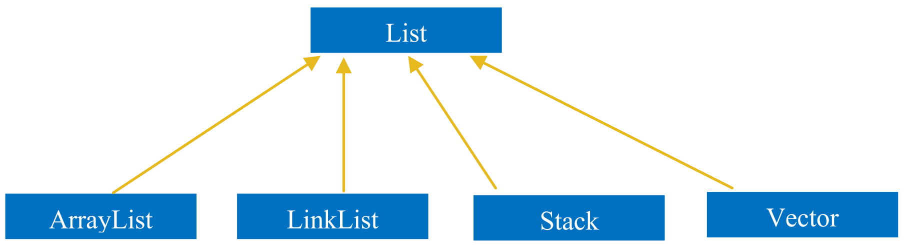
COLLECTION

- Collection is a data structure similar to Arrays. But, it doesn't need to define the size of the collection.
- The data stored in the collection is encapsulated and the access to the data is only possible via predefined methods.
- Java provides a large number of interfaces and classes that define and implement various types of collections.
- Most commonly used interfaces:
    - List *(Elements: any)*
    - Set *(Elements: unique)*
    - Map *(Elements: key-value pair)*

1. List

- Allow duplicated data.
- Stored data zero-based index.
- Since List is an interface, objects cannot be created from it.
- To implement List, its child classes can be used.

```
List<T> al = new ArrayList<>();
List<T> ll = new LinkedList<>();
List<T> v = new Vector<>();
List<T> s = new Stack<>();
```



Some commonly used methods –

add()	Add an element to a list
addAll()	Add all elements of one list to another
Get()	Randomly access element from list
Set()	Change element of list
Remove()	Remove an element from list
clear()	Remove all elements from list
size()	Returns the length of a list
toArray()	Convert a list into an array
Contains()	returns true if a list contains specified element

Example – Ways of Creating List

```
List<Integer> immutableEmptyList = List.of(); // cannot modify
List<Integer> immutableList1 = List.of(100,200,300); // cannot modify
Integer[] arr = {10,20,30,40};
// create with existing data ,cannot modify
List<Integer> immutableList2 = Arrays.asList(arr);
// create empty new arraylist, can modify
List<Integer> mutableEmptyList = new ArrayList<>();
// create new arraylist from existing data, can modify
List<Integer> mutableList = new ArrayList<>(Arrays.asList(arr));
```

ArrayList

- ArrayList is most commonly used.
- It is based on an Array data structure but it does not need to predefine its size.

Example – Basic Operations

```
ArrayList<Integer> list =new ArrayList<>();
// add an element
list.add(20);
list.add(10);
list.add(50);

System.out.println("Size: " + list.size()); // get number of elements
System.out.println(list);

list.add(1, 40); // add element at specific index
list.forEach(l->System.out.print(l+ " "));
list.set(0, 250); // update element
System.out.println("\nElement at index 0: " + list.get(0)); // get element
System.out.println("Is empty? " + list.isEmpty());

list.remove(1); // remove by index
System.out.println("After removing " + list);

System.out.println("Max value: " + Collections.max(list)); // find max
System.out.println("Min value: " + Collections.min(list)); // find min

list.clear(); // remove all elements
System.out.println("Is empty? " + list.isEmpty());
```

Example – Sorting and Searching

```
String[] data = {"Aung Aung","Jeon","Yuki","Maung Maung"};
ArrayList<String> list = new ArrayList<String>(Arrays.asList(data));

list.add("Jeon"); // allow duplicate data

System.out.println(list);
Collections.sort(list);// sort asc
System.out.println("After sort: " + list);
Collections.reverse(list); //sort desc
System.out.println("After reverse: " + list);

int result = Collections.binarySearch(list, "Jeon");// return index if exist
System.out.println((result < 0) ? "Jeon is not found" : "Jeon is found");
result = Collections.binarySearch(list, "abc");// return -value if not exist
System.out.println((result < 0) ? "abc is not found" : "abc is found");

if(list.contains("Yuki"))
        System.out.println("Yuki is found");

list.remove("Jeon"); // remove by element
System.out.println("After remove " + list);

list.removeIf(s-> (s.endsWith("ung") && s.length() > 9));
System.out.println("After remove " + list);
```

Please visit this link to learn ArrayList in more details –

1. https://beginnersbook.com/java-collections-tutorials/#1
2. https://www.callicoder.com/java-arraylist/

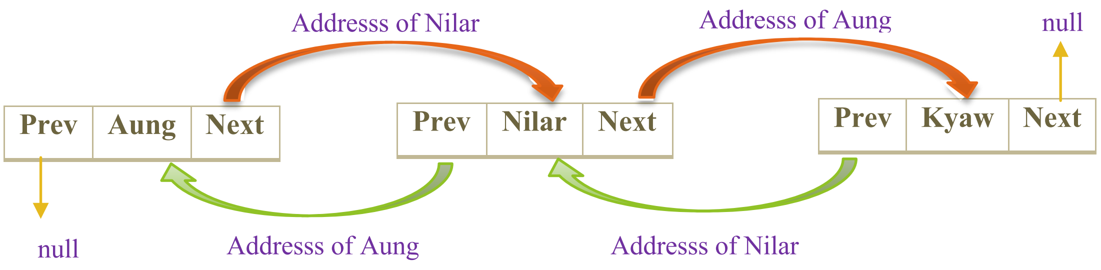
Example – ArrayList with User-defined Object

```
List<Phone> list = new ArrayList<>();

list.add(new Phone("iPhone",1500000));
list.add(new Phone("Samsung",1600000));
```

```
Phone obj = new Phone("Vivo",300000);
list.add(obj);

list.forEach(ph-> System.out.println("Name: " + ph.name + ",Price: " + ph.price));
list.forEach(ph->{
        if(ph.price > 500000)
                System.out.println(ph.name + "(" + ph.price + ")");
});
Phone phone = list.get(1);
System.out.println("Name: " + phone.name + "(" + phone.price + "ks.)");
```

LinkList

- Unlike ArrayList, elements do not need contiguous memory locations.
- Each element has the reference (address/pointer) to the next element.
- It takes more memory space but insert/delete operation is fast.



Example

```
LinkedList<String> lList = new LinkedList<>(); // create LinkedList

lList.add("Nilar");
lList.add(0, "Aung");
lList.add("Kyaw");

System.out.println(lList);

lList.addFirst("Jeon");
lList.addLast("Cherry");

System.out.println(lList);
System.out.println(lList.get(2));
System.out.println(lList.getLast());
```

Output:

```
[Aung, Nilar, Kyaw]

[Jeon, Aung, Nilar, Kyaw, Cherry]

Nilar
```

Stack

- Elements are stored and accessed in Last In First Out manner.

Example

```
Stack<String> cities = new Stack<>();    // create empty Stack
cities.push("Yangon"); // add element
cities.push("Mandalay");
cities.push("Pyin Oo Lwin");
System.out.println(cities);

cities.pop(); // remove object

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

Assignment 1

- Create a list of Student names by using ArrayList. Do the following functions:
  a) Show all the student names.
  b) Sort the list.
  c) Show the sorted list.
  d) Search the specified student and show his position in the list.
  e) Insert the new Student name if it is not already included in the list.
  f) Show the students whose name starts with 'K' or 'k'.
  g) Remove the students whose name ends with 'G' or 'g' show the list.
  h) Clear the list.

Assignment 2

- Assume that a system has three collections to store book, author and book category information.

```
static List<String> categoryList = new ArrayList<>();
static List<Author> authorList = new ArrayList<Author>();
static List<Book> bookList = new ArrayList<Book>();
```

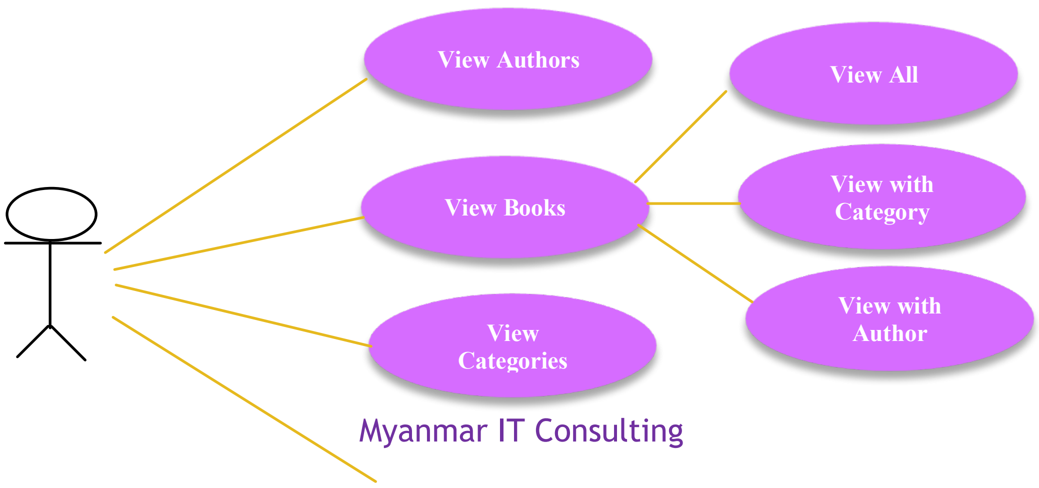
- A book contains these data

```
private int code;
private String title;
private LocalDate publishDate;
private String category;
private Author author;

+ Constructors
+ getter/setter
```

- An author contains these data

```
private String name;
private String country;

+ Constructors
+ getter/setter
```

- A category contains just name.
- This system initially has 3 authors, 4 categories and 3 books.
- The following functions are included in the system.



Add Book

1. View all book information
2. View all author information
3. View all book category information
4. View book list according to author name (user input)
   a. If author name does not exist in author list, display error message " This author does not exist".
   b. Although the author list contains that author name, if there is no book associated to the author, display message "There is no book data for this author".
   c. Only if there is book data for the author, display book list of the author.
5. View book list according to category (user input)
   a. The processes of this function are the same to No.4
6. Add new book (get all book information from user)
   a. If input category name is new, add automatically the category to category list.
   b. If input author is new, display message "This is new author. Please enter author's country name" and get input data and then automatically add the new author to author list.
   c. When the new book has been added to the book list, verify the data of the corresponding collections are updated or not.

3. Set

- Unlike List, Set interface does not allow duplicated value.
- Since Set is an interface, it cannot be instantiated. Some of its subclasses –
    - HashSet
    - LinkedHashSet
    - TreeSet

```
Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
```

HashSet

- It is commonly used.
- Elements are inserted in a random order.
- Only allows one null element.

Example

```
HashSet<String> set1 = new HashSet<>(); // create empty hashset
set1.add("Orange"); // add new object
set1.add("Apple");
set1.add(null);
set1.add("Mango");
set1.add("Strawberry");

System.out.println(set1);

set1.remove("Apple"); // remove object

System.out.println(set1);

Set<String> set2 = Set.of("Potato","Pineapple");
set1.addAll(set2); // union two sets

System.out.println(set1);
```

Output:

```
[null, Apple, Strawberry, Mango, Orange]

[null, Strawberry, Mango, Orange]

[null, Potato, Strawberry, Mango, Pineapple, Orange]
```

LinkedHashSet

- Unlike HashSet, it maintains elements ordered as the insertion order.

Example

```
LinkedHashSet<String> set1 = new LinkedHashSet<>();
set1.add("Orange");
```

```
set1.add("Apple");
set1.add(null);
set1.add("Mango");
set1.add("Strawberry");

set1.forEach(s->System.out.println(s));

set1.remove(null); //remove an object
set1.removeIf(s->s.contains("o")); // remove based on condition

System.out.println("After remove, " + set1);
```

TreeSet

- Doesn't allow null elements.
- Elements stored in ascending order.

Example

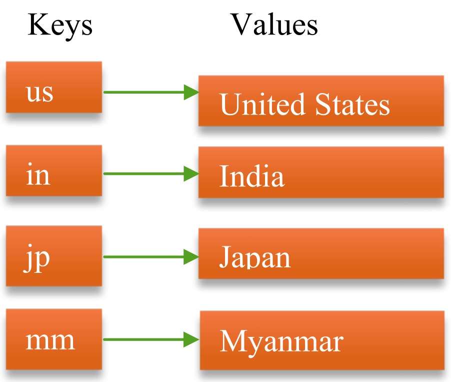
```
TreeSet<String> set1 = new TreeSet<>();
set1.add("Orange");
set1.add("Apple");
set1.add(null); // cause error
set1.add("Mango");
set1.add("Strawberry");

Iterator<String> itr = set1.iterator();
itr.forEachRemaining(s->System.out.println(s)); // print using iterator

Set<String> set2 = set1.descendingSet(); //get descending TreeSet
System.out.println(set2);
```

Output:

```
Apple
Mango
Orange
Strawberry
[Strawberry, Orange, Mango, Apple]
```

4. Map

- Elements of Map are stored in key/value pairs.
- It does not allow duplicated keys.
- Values can be access via the keys.
- Some classes that implements Map interface
  - HashMap
  - LinkedHashMap
  - TreeMap



```
Map<K,V> hm = new HashMap<> ();
Map<K,V> tm = new TreeMap<> ();
Map<K,V> lhm = new LinkedHashMap<>();
```

Remark –

- The Map interface contains 3 different sets –
  - Set of keys
  - Set of values
  - Set of key/value associations

HashMap

- It is basic implementation of Map interface and most commonly used.
- It permits null values and the null key.

Example – Creating, Adding, Union

```
Map<String, String> foods = new HashMap<>(); // create empty map

foods.put("Orange", "Fruit"); // put a new object if key not already exist.
foods.put("Grape", "Fruit");

Map<String,String> another1 = Map.of("Mango","Fruit","Potato","Vegetable");

foods.putAll(another1); // union
System.out.println(foods);

Map<String,String> another2 = Map.ofEntries(
                                Map.entry("Coffee", "Juice"),
                                Map.entry("Lemon Tea", "Juice")
                        );

foods.putAll(another2);
foods.forEach((k,v)->System.out.println(k + "->" + v));

foods.putIfAbsent("kiwi", "Fruit"); // put if key is not already exist
System.out.println("After add new key: " + foods);

System.out.println("Contain key('Coffee'): " + foods.containsKey("Coffee")); // check key
System.out.println("Contain value('Snack'): " + foods.containsValue("Snack")); // check value

Set<String> keys = foods.keySet(); // get only keys
System.out.println("All keys: " + keys);

Collection<String> values = foods.values(); // get only values
System.out.println("All values: " + values);
```

Example – Removing Elements

```
foods.remove("Mango"); // remove
```

```
System.out.println("After remove 'Mango', " + foods);

foods.keySet().removeIf(k-> k.contains("Tea")); // remove base on condition(key)
System.out.println("After remove key('Tea'), " + foods);

foods.values().removeIf(v-> v.equalsIgnoreCase("fruit")); // remove base on condition(value)
System.out.println("After remove value('fruit'): " + foods);
```

Example – Updating Elements

```
foods.replace("Mango", "PineApple"); // replace
System.out.println("After update: " + foods.get("Mango"));

foods.compute("Coffee", (k,v)-> v.toUpperCase()); // update if already key exist
System.out.println("After update: " + foods.get("Coffee"));

foods.compute("Cake", (k,v) -> "Snack"); // put new if not exist key
System.out.println("After update: " + foods);

foods.computeIfAbsent("Orange", k -> "Juice"); // no effect if already key exist
System.out.println("After update: " + foods.get("Orange"));

foods.computeIfAbsent("Banana", k -> "Fruit");  // put new if not exist key
System.out.println("After update: " + foods);

foods.computeIfPresent("Orange", (k,v) -> "Juice"); // update if key already exist
System.out.println("After update: " + foods.get("Orange"));

foods.computeIfPresent("Corn", (k,v) -> "Vegetable"); // no effect if not exist key
System.out.println("After update: " + foods.get("Corn"));
```

Please visit these links to learn HashMap in more details -

1. https://www.programiz.com/java-programming/library/hashmap
2. https://www.callicoder.com/java-hashmap/

Example – with User-defined Objects

```
class Employee{
        private int id;
        private String name;
        private String city;

        //constructors
        // getter/setter
}
Map<Integer, Employee> employees = new HashMap<>();

employees.put(1001, new Employee(1001,"Nyi Nyi","Yangon"));
employees.put(1002, new Employee(1002,"Htet Htet","Pyin Oo Lwin"));
employees.put(1003, new Employee(1003,"Naung Naung","Yangon"));
employees.put(1004, new Employee(1004,"Aung Aung","Mandalay"));

employees.forEach((k,emp) ->{
        System.out.println(emp.getId() + " " + emp.getName() + " " + emp.getCity());
});

Employee emp = employees.get(1004);
System.out.println("Id: " + emp.getId());
System.out.println("Name: " + emp.getName());
System.out.println("City: " + emp.getCity());

employees.forEach((k,v)->{
        if(v.getCity().equalsIgnoreCase("yangon"))
                System.out.println(v.getName());
});
```

LinkedHashMap

- Unlike HashMap, it maintains insertion order.

Example

```
LinkedHashMap<Integer, String> students = new LinkedHashMap<>();

students.put(11, "Aung Aung");
students.put(5, "Kyaw Kyaw");
students.put(8, "Honey");
students.putIfAbsent(10, "Cherry");

students.forEach((k,v)->{
        System.out.println("Rno: " +k);
        System.out.println("Name: " + v);
});

students.replace(8, "Honey Htun");
System.out.println("After replace: " + students);

students.remove(11);
System.out.println("After remove: " + students);
```

Please visit these links to learn LinkedHashMap in more detail.

1. https://www.callicoder.com/java-linkedhashmap/
2. https://www.programiz.com/java-programming/linkedhashmap

TreeMap

- TreeMap doest not allow the null key.
- It sorts the elements in the ascending order of its keys

Example

```
TreeMap<String, String> fileExtensions = new TreeMap<>();

fileExtensions.put("java", ".java");
fileExtensions.put("php", ".php");
fileExtensions.put("c++", ".cpp");
fileExtensions.put("html", ".html");
fileExtensions.putIfAbsent("javascript", ".js");

System.out.println(fileExtensions);
fileExtensions.remove("c++");

fileExtensions.forEach((k,v)->System.out.println(k + " => " + v));
```

Please visit these links to learn TreeMap in more details

1. https://www.callicoder.com/java-treemap/
2. https://www.programiz.com/java-programming/treemap

- Create list of Student objects and make these operations –
  - Insert a new student
  - Sorting
  - Display all students
  - Update student data
  - Search student with roll no
  - Delete a specific student according to roll
- Use the following collection types to store student objects
  - Set
  - Map (use the student’s roll no as Key)

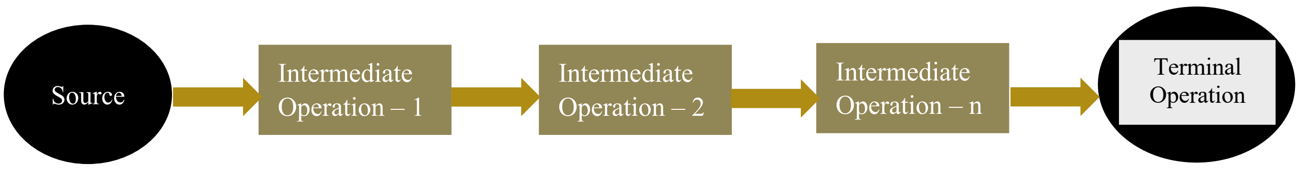
STREAM

What is Stream?

- A sequence of elements from a source.
- The source of elements can be collections, array or other data sources.
- Not a data structure.
- Do not support indexed access
- Less visual clutter in code
- Automatic iterations

How to work Stream

- Three stages –
  - Create a stream
  - Perform intermediate operations
  - Perform terminal operations


Source
Intermediate Operation – 1
Intermediate Operation – 2
Intermediate Operation – n
Terminal Operation

- All the classes and interfaces of this API is provided in java.util.stream package.

Example

```
int[] prices = {1700,3500,3800,2200,5000,13500,500,1500,1000};
IntStream streams = Arrays.stream(prices); //create stream
System.out.println("*** Prices > 1500 ***");
streams.filter(p -> p > 1500)  // intermediate operation
            .sorted()
            .forEach(p -> System.out.println(p)); // terminal operation
```

Stream Operations

- Stream Operations come in two types –
  - **Intermediate operation**
    - It takes a stream as input and return a stream (can be chained).
      - filter()
      - map()
      - flatMap()
      - skip()
      - distinct()
      - sorted()
      - limit()
  - **Terminal operation**
    - It is used to produce end result. It does not return a new Stream instance (cannot be chained).

- forEach()	- count()
- toArray()	- anyMatch()
- reduce()	- allMatch()
- collect()	- noneMatch()
- min()	- findFirst()
- max()	- findAny()

Filtering

- It takes predicate as an argument and returns elements of the stream that match the given predicate.
- Predicate is a functional interface, which accepts an argument and returns a Boolean.

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

Example –

```
class User{
        private String name;
        private String role;
        // constructor
        // getter/setter
}
List<User> users = List.of(
                    new User("Kyaw Kyaw","Admin"),
                    new User("Aung Aung","Staff"),
                    new User("Maung Maung","Staff"),
                    new User("Yuri","Customer"),
                    new User("Jeon","Customer")
          );

users
    .stream()
    .filter(u -> u.getRole().equals("Staff")) // filtering
    .forEach(u -> System.out.println(u.getName() + "(" + u.getRole() + ")")); // iterating
```

Predicate allows method reference.

Example – with Method Reference

```
class User{
      boolean IsStaff() {
                return this.role.equals("Staff");
          }
}

  users
          .stream()
          .filter(User::IsStaff)
          .forEach(u -> System.out.println(u.getName() + "(" + u.getRole() + ")"));
```

Example – Filtering with Multiple Criteria

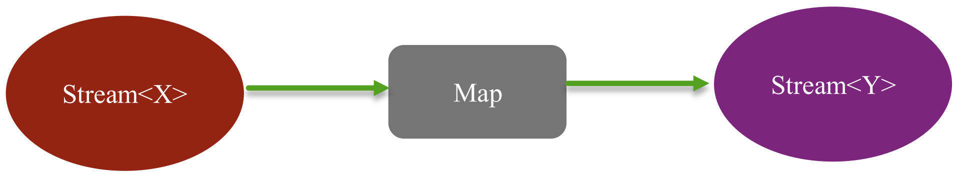
```
List<String> staffs = users
                          .stream()
                          .filter(u -> u.getName().contains("aung") && u.getRole().equals("Staff"))
                          .map(u -> u.getName()) // fetching only name
                          .collect(Collectors.toList()); // convert to List
System.out.println(staffs);
```

Example – Multiple Filter Operations

```
Stream<Integer> numbers = Stream.iterate(1, i -> i + 1)
                                      .limit(15);

Predicate<Integer> evens = x -> x%2 == 0; // create predicate

numbers
         .filter(a -> a > 4 && a <= 10)
         .filter(evens)
         .forEach(System.out::println);
```

Mapping

- It is used to convert Stream of one type to another.
- It takes mapper function as argument and generates the new Stream.
- Mapping function is functional interface which takes one input to one output.



```
map() method syntax

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

- Stream interface has three similar methods which produce IntStream, LongStream and DoubleStream respectively after the map operation.

```
Similar methods

IntStream mapToInt(ToIntFunction<? super T> mapper)
LongStream mapToLong(ToLongFunction<? super T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

Example – 1

```
List<Employee> empList = Arrays.asList(
                                        new Employee("Kyaw Kyaw",9800,"Yangon"),
                                        new Employee("Aung Aung",6000,"Mandalay"),
                                        new Employee("Mg Mg",10000,"Mandalay"),
                                        new Employee("Yuri",6000,"Yangon"),
                                        new Employee("Jeon",7800,"Monywa")
                              );

List<String> distinctCities = empList
                                        .stream()
                                        .map(e -> e.getCity()) // get only city name
                                        .distinct()   // unique
                                        .collect(Collectors.toList());

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

Example – 2

```
int[] salaries = empList
                    .stream()
                    .filter(e -> e.getSalary() > 6000)
                    .mapToInt(e -> e.getSalary())
                    .toArray();
for(var s: salaries)
    System.out.println(s);
```

Example – 3

```
List<Integer> salaries = empList
                            .stream()
                            .map(e -> e.getSalary())
                            .filter(s -> s < 10000)
                            .limit(3)
                            .collect(Collectors.toList());
System.out.println(salaries);
```

Example – 4

```
int maxSalary = empList
                    .stream()
                    .mapToInt(e -> e.getSalary())
                    .max()      // find max salary
                    .getAsInt();
int minSalary = empList
                    .stream()
                    .mapToInt(e -> e.getSalary())
                    .min()     // find min salary
                    .getAsInt();

System.out.println("Maximum salary is " + maxSalary);
System.out.println("Minimum salary is " + minSalary);
```

flatMap()

- It converts a stream of collections into a single “flat” stream.
- Flattening is referred to as merging multiple collections/arrays into one.

```
flatMap = Flattening(flat) + Mapping(map)
```

Example

```
List<String> drinks = Arrays.asList("Cola","Milk Tea","Pessi");
List<String> foods = List.of("Burger","Kyay Oho","Noodles","Cake");
List<String> desserts = List.of("Ice cream","Cake");

List<List<String>> items = new ArrayList<>();
items.add(drinks);
items.add(foods);
items.add(desserts);

System.out.println("Before flat: " + items);

Set<String> flatList = items
                        .stream()
                        .flatMap(v -> v.stream())
                        .collect(Collectors.toSet());
System.out.println("After flat: " + flatList);
```

Output

```
Before flat:
[[Cola, Milk Tea, Pessi], [Burger, Kyay Oho,
Noodles, Cake], [Ice cream, Cake]]

After flat:
 [Burger, Cola, Pessi, Kyay Oho, Cake, Milk
Tea, Ice cream, Noodles]
```

Matching

- Terminal Operation.
- Take a predicate and return Boolean result.
    1. anyMatch() – return true if at least one element matches the given predicate.
    2. allMatch() – return true if all elements match the given predicate.
    3. noneMatch() – opposite of allMathch().

Note – allMatch() and noneMatcho() return true on an empty stream.

Example

```
List<Integer> numbers = List.of(2,4,6,8,10,11);

        boolean allEven = numbers.stream().allMatch(i -> i%2 == 0); // false
        boolean oneEven = numbers.stream().anyMatch(i -> i%2 == 0); // true
        boolean noneEven = numbers.stream().noneMatch(i -> i%2 == 0); // false
```

Reduction

- It combines elements of a stream and produces a single value.
- Built-in reducing operations - average, sum, min, max, and count etc.

Example

```
List<Employee> empList = Arrays.asList(
                        new Employee("Kyaw Kyaw", 9800, "Yangon"),
                        new Employee("Aung Aung", 6000, "Mandalay"),
                        new Employee("Mg Mg", 10000, "Mandalay"),
                        new Employee("Yuri", 6000, "Yangon"),
                        new Employee("Jeon", 7800, "Monywa"));

int total = empList.stream().mapToInt(Employee::getSalary).sum();
double avg = empList.stream().mapToDouble(Employee::getSalary).average().getAsDouble();
int max = empList.stream().mapToInt(e -> e.getSalary()).max().getAsInt();
long count = empList.stream().filter(e -> e.getSalary() > 6000).count();
Employee empMax = empList.stream().max(Comparator.comparingInt(e -> e.getSalary())).get();
Employee empMin = empList.stream().min((e1,e2) -> e1.getSalary() - e2.getSalary()).get();
```

- Custom reducing operations can be created with Stream.reduce() method.

Example

```
int totalSalary = empList.stream()
                        .map(Employee::getSalary)
                        .reduce(0,(s1, s2) -> s1 + s2); // 0 is initVal

int minSalary = empList
                        .stream()
                        .mapToInt(e -> e.getSalary())
                        .reduce(Integer::min) //Optional<Integer>
                        .getAsInt(); //Integer

Employee empMaxSal = empList
                                .stream()
                                .reduce((e1,e2) -> e1.getSalary() < e2.getSalary() ? e2: e1)
                                .get();
```

Collecting

- It is used to collect the output of stream operations into the collection such as List, Set or Map.
- Collectors class provide different methods like toList(), toSet() and toMap() to collect the result of steam.

Example

```
 class Student{
        private int rno;
        private String name;
        private String city;
        // constructors
        // getter/setter
}
Student[] students = {
        new Student(10,"cherry","yangon"),
        new Student(2,"cherry","mandalay"),
        new Student(5,"khaing","monywa"),
        new Student(1,"htet","mandalay"),
        new Student(11,"htet yadana","yangon")
};

Set<String> hashSet = Arrays.stream(students)
                                .map(s -> s.getName())
                                .collect(Collectors.toSet());
TreeSet<Integer> treeSet = Arrays.stream(students)
                                .map(s -> s.getRno())
```

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


Map<Integer,String> hashmap1 = Arrays.stream(students)
                                        .collect(Collectors.toMap(Student::getRno,Student::getName));

Map<Integer,Student> hashmap2 = Arrays.stream(students)
                                                  .filter(s -> !s.getCity().equals("yangon"))
                                                  .collect(Collectors.toMap(
                                                                      s -> s.getRno(), //key
                                                                      s -> s  // value
                                                  ));

System.out.println(hashSet); // [cherry, htet yadana, khaing, htet]
System.out.println(treeSet); // [1, 2, 5, 10, 11]
System.out.println(hashmap1); // {1=htet, 2=cherry, 5=khaing, 10=cherry, 11=htet yadana}
System.out.println(hashmap2);
//{1=(rno=1, name=htet, city=mandalay), 2=(rno=2, name=cherry, city=mandalay), 5=(rno=5,
name=khaing, city=monywa)}
```

Group By

- groupingBy() is similar to “GROUP BY” clause in the SQL language.

Example – Count, Sum, Average

```
List<Employee> empList = Arrays.asList(
                                        new Employee("Kyaw Kyaw",9800,"Yangon"),
                                        new Employee("Aung Aung",6000,"Mandalay"),
                                        new Employee("Mg Mg",10000,"Mandalay"),
                                        new Employee("Yuri",6000,"Yangon"),
                                        new Employee("Jeon",7800,"Monywa")
                              );
// count
Map<String, Long> counting = empList
                                           .stream()
                                           .collect(
                                                 Collectors
                                                    .groupingBy(e -> e.getCity(),Collectors.counting())
                                           );
// sum
Map<String,Integer> sum = empList
                                        .stream()
                                        .collect(
                                            Collectors.groupingBy(e -> e.getCity(),
                                                 Collectors.summingInt(Employee::getSalary))
                                        );
// avg
Map<String,Double> average = empList
```

```
                                             .stream()
                                             .collect(
                                                 Collectors.groupingBy(Employee::getCity,
                                                     Collectors.averagingDouble(Employee::getSalary))
                                         );
System.out.println(counting); // {Monywa=1, Mandalay=2, Yangon=2}
System.out.println(sum); // {Monywa=7800, Mandalay=16000, Yangon=15800}
System.out.println(average); // {Monywa=7800.0, Mandalay
```

Output

```
10000 ks.
        Mg Mg(Mandalay)
6000 ks.
        Aung Aung(Mandalay)
        Yuri(Yangon)
7800 ks.
        Jeon(Monywa)
9800 ks.
        Kyaw Kyaw(Yangon)
```

Example – display employee data in each salary

```
// group by salary
Map<Integer, List<Employee>> groupBySalary =
                                     empList
                                         .stream()
                                         .collect(
                                             Collectors.groupingBy(Employee::getSalary)
                                         );
groupBySalary.forEach((k,v)->{
        System.out.println(k + " ks.");
        v.forEach(a -> System.out.println("\t" + a.getName() + "(" + a.getCity() + ")"));
});
```

Example – find employee name in each city

```
Map<String, List<String>> groupByCity =
        empList
            .stream()
            .collect(
                    Collectors.groupingBy(Employee::getCity,
                    Collectors.mapping(Employee::getName, Collectors.toList()))
            );
System.out.println(groupByCity);
// {Monywa=[Jeon], Mandalay=[Aung Aung, Mg Mg], Yangon=[Kyaw Kyaw, Yuri]}
```

Example – find employes who stay in same city

```
Map<String,List<Employee>> data =
        empList
            .stream()
            .collect(
                Collectors
                    .collectingAndThen(
                            Collectors.groupingBy(Employee::getCity),
                                        map -> map.entrySet() // get Map<String,List<Emp>>
                                                  .stream()
                                                  .filter(e -> e.getValue().size() > 1) // filter List<emp> size
                                                  .collect(Collectors.toMap(v -> v.getKey(), v -> v.getValue())
                                        )
                              )
                    );

System.out.println(data);
// {Mandalay=[Name: Aung Aung,Salary: 6000, Name: Mg Mg,Salary: 10000], Yangon=[Name:
Kyaw Kyaw,Salary: 9800, Name: Yuri,Salary: 6000]}
```

Assignment 1

Create Salespeople list which contain the following information

name	city	comm
Peel	London	0.12
Serres	San Jose	0.13
Motika	London	0.11
Rifkin	Barcelona	0.15
Axelrod	New York	0.10

Display the following information by using suitable stream operations.

1. Names and cities of all salespeople in London with a commission above .10.
2. Salesperson details not having commission .10, .13 or .15.
3. Different city names.
4. The top of (salespeople 3) records.
5. Details of the salespeople who live in ‘Rome’.
6. The number of salespeople who stay in London.
7. List of salespeople in descending order of commission.

Zero to Pro Bootcamp – Level 3

Assignment 2

Create Employee list which contains the following data –

name	city	department	salary	birthday
Htet Htet	Yangon	Electronics	900000	1991-10-16
Cherry	Yangon	Electronics	820000	1994-08-14
Kyaw Kyaw	Yangon	Electronics	780000	1988-09-02
Aung Aung	Mandalay	IT	600000	1995-01-11
Jeon	Mandalay	IT	600000	1997-09-01
Hsu Hsu	Pyin Oo Lwin	IT	920000	1994-12-10
Aye Aye	Yangon	HR	500000	1992-10-10
Gay Gay	Taung Gyi	HR	400000	1996-05-12
Phway Phway	Monywa	HR	300000	1995-09-03
Ko Ko	Monywa	IT	500000	1992-11-11

Find the following information by using the suitable stream operations.

1. The minimum salary of employees.
2. The youngest employee information
3. Count all employee whose birthday is greater than or equal '1995-02-12'.
4. Total salary of all employees.
5. Fetch the three max salaries.
6. The average salary of department 'HR'.
7. Employee which gets smallest salary.
8. The highest salary of employee in each city.
9. List of employee who got the same salary.
10. List of employee names in each department.

s

THANK YOU