Java 8 Notes

Interface Changes in Java 8

- Prior to java 8, **interface in java** can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods

What is the advantage of having Default methods in java 8?

- It provides backward compatibility.
- For example, if several classes such as A, B, C and D implements an interface Test (Interface name is Test) then if we add a new method to the Test, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept "default methods". These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code
- We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code
- To access the default method present in interface we can use the below syntax.
  InterfaceName.super.MethodName()
  Test.super.display() (in this example display is the default method in interface)

What is the advantage of having Static method in Interface in java 8 ?

- As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes
- How to call the interface static method ?
  InterfaceName.methodName()
  Test.addition(); (In this example addition is the static method)
- How it is different from calling the static method of a class
  ClassName.methodName for class
  InterfaceName.methodName for interface
- Java interface static method helps us in providing security by not allowing implementation classes to override them.
- We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use
- Java interface static methods are good for providing utility methods, for example null check, collection sorting etc

Lambda Expressions

- Lamda enables Functional Programming - This is the advantage of Lamda in Java 8
- Lamda enables support for parallel processing
- What is Functional Programming ? - Till now in Java we passed data as method argument. Passing the functionality as method argument is called as Functional programming
- Example: show(()->System.out.println("hi")); Here show is method. We are passing the lamda as parameter. Earlier we have to pass the object and on this object call the method
- In object-oriented programming, data is stored in objects
- In functional programming, data cannot be stored in objects, and it can only be transformed by creating functions
- How to Write a Lamda in Java ?
- Example: We have an interface called Test like below

```
Public interface Test{
        Public void display()
    }
```

If I want to make use of the above interface then create the class and implement this interface and override the display method

Create the object of the class and call display method using the object.

But Using Lamda no need of implementing.

- Similar to the variables in Java

  String s="Ashok it"; Here in this line s is a variable and it is of type String holding the value as "Ashok it";

  To declare the variable we are using the data types similarly every lamda corresponds to interface

  To create the lamda for above interface write like below

  Test testLamda=()->System.out.println("Hi);

  Here how did we write this statement -> as I mentioned each lamda corresponds to one interface so Since we are writing the lamda for Test interface we have mentioned the Left side like Test testLamda

  The java8 compiler looks at the abstract method present in the test interface and accepts the arguments and return the data accordingly. Since the java compiler is intelligent enough to refer the abstract method no need to write the method name, argument and return type

  - That means first write like this Test testLamda=public void display(){

    ```
                                        System.out.println("Hello");
                                    }
    ```

  - Later according to the above point remove the method name, parameters and return type and access modifier because compiler internally refers to abstract method in interface

    Here display method is not taking any arguments hence empty brace()

    And after that if we want to perform any logic we need to write **-> symbol.** If the code implementation is in 2 lines we can use curly brace or else not needed

- Every lamda corresponds to Interface and that interface should have only one abstract method
- If there are more abstract methods then Java 8 compiler doesn't know which abstract method it should refer. Hence we should allow only one abstract method per interface. In order to facilitate this feature the corresponding interface should be marked as @FunctionalInterface.
- In this way if we are writing the lamda then we need to create n number of interfaces so to avoid this Java 8 has introduced java.util.functions package which has 4 types of interfaces and we can make use of those based on the business scenario.
- Example of lamda expression: The below is double abstract method present in Test1 interface then how can I write the lamda. (below is the double method implementation)

  ```
  Public int double(int n){
      Return n*2;
  }
  ```

  Lamda is : Test1 doubleLamda=(n)->return n*2;

- Lamda can replace the anonymous inner class. Example of creating the thread
- Advantage of Lamda is,
  - It enables functional programming. (Functional programming means sending the functionality as method argument)
  - We can replace the anonymous inner class with Lamda expression

Functional Interfaces

- Functional interface is something in which there is only 1 abstract method is present and it can have any number of default or static methods.
- To make or denote the interface as functional interface, we have to annotate it as @FunctionalInterface annotation
- Once we make the interface annotate with @FunctionalInterface then we cannot add more than 1 abstract method.
- @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It's optional but good practice to use it.

- Functional interface corresponds to lamda expression
- A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references
- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface
- Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations. These are:
  - Consumer
  - Predicate
  - Function
  - Supplier
- Consumer, Predicate, and Function, likewise have additions that are provided beneath –
  - Consumer -> Bi-Consumer
  - Predicate -> Bi-Predicate
  - Function -> Bi-Function
- **Consumer**
  - The consumer interface of the functional interface is the one that accepts only one argument. It returns nothing
  - `void accept(T paramT);`
- **Bi-Consumer**
  - The Bi-consumer interface of the functional interface is the one that accepts two arguments. It returns nothing
  - `void accept(T paramT, U paramU);`
- **Predicate**
  - A function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate
  - `boolean test(T paramT);`
- **Bi-Predicate –**
  - Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value
  - `boolean test(T paramT, U paramU);`
- **Function**
  - A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing
  - `R apply(T paramT);`
- **Bi-Function –**
  - The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.
  - `R apply(T paramT, U paramU);`
- **Supplier**
  - The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output
  - T get();
  - There is no Bi Supplier because as it doesn't take any argument and any method can have 1 return type so bi supplier is not possible

➢ Type Inference
- Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable.

➢ Method Reference:

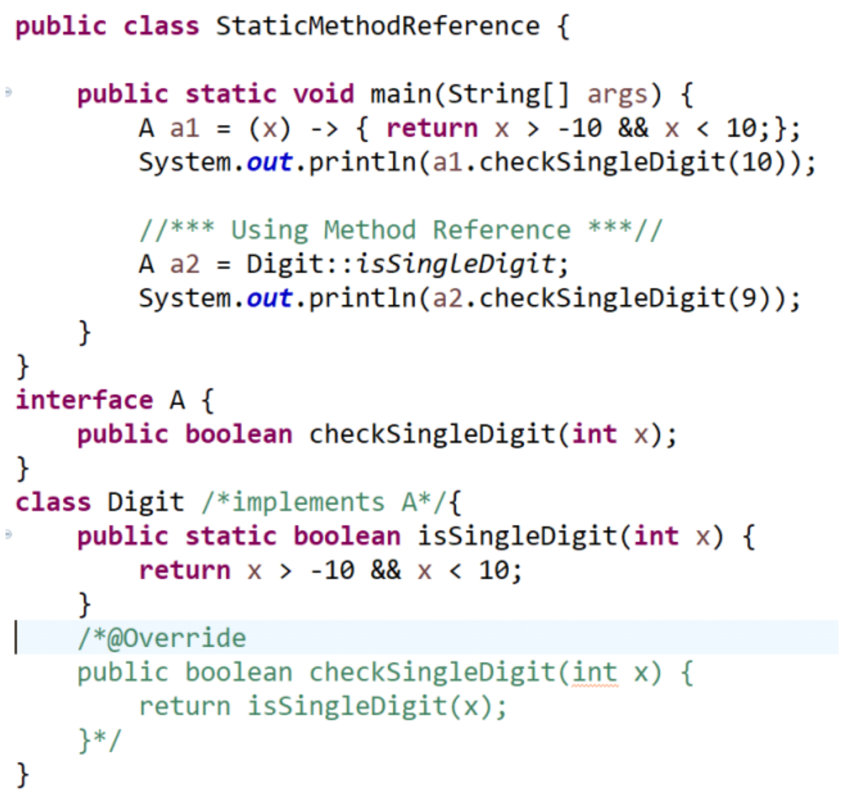
- There are following types of method references in java:
  - Reference to a static method.
  - Reference to an instance method.
  - Reference to a constructor

➢ Method Reference to a static method

- When you are writing the lamda which takes no arguments and if it calling some method and in which it doesn't take arguments then we can use method reference.

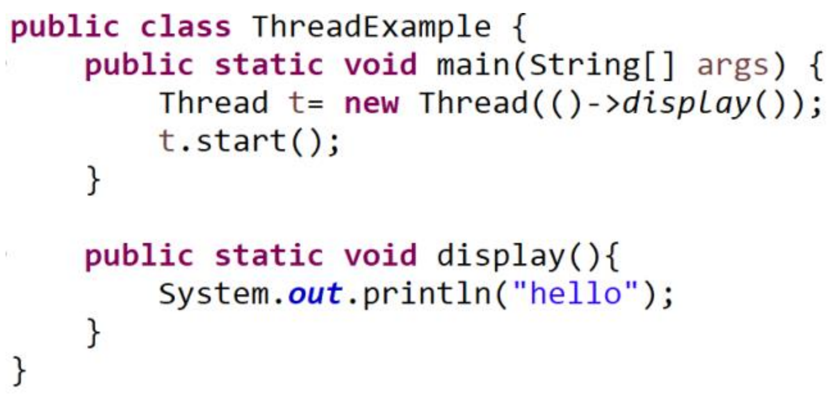
Code without method reference:
Example 1: Lamda Implementation and Lamda with Method Reference.

Look at the below code if you are using without method reference then u need to enable the commented code. If you are using method reference then not required.



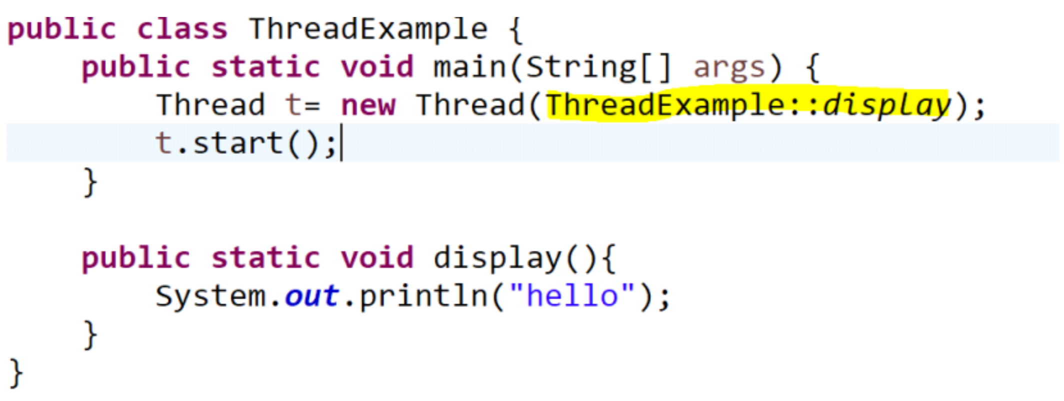
```java
public class StaticMethodReference {

    public static void main(String[] args) {
        A a1 = (x) -> { return x > -10 && x < 10;};
        System.out.println(a1.checkSingleDigit(10));

        //*** Using Method Reference ***//
        A a2 = Digit::isSingleDigit;
        System.out.println(a2.checkSingleDigit(9));
    }
}
interface A {
    public boolean checkSingleDigit(int x);
}
class Digit /*implements A*/{
    public static boolean isSingleDigit(int x) {
        return x > -10 && x < 10;
    }
    /*@Override
    public boolean checkSingleDigit(int x) {
        return isSingleDigit(x);
    }*/
}
```

Example 2:



```java
public class ThreadExample {
    public static void main(String[] args) {
        Thread t= new Thread(()->display());
        t.start();
    }

    public static void display(){
        System.out.println("hello");
    }
}
```

- The above code shows the lamda expression for runnable interface. Passing the Runnable instance to the Thread class
- In this above example we can replace the lamda with method expression like below
- Here also same thing when you implement run method inside run method you will call the display method.

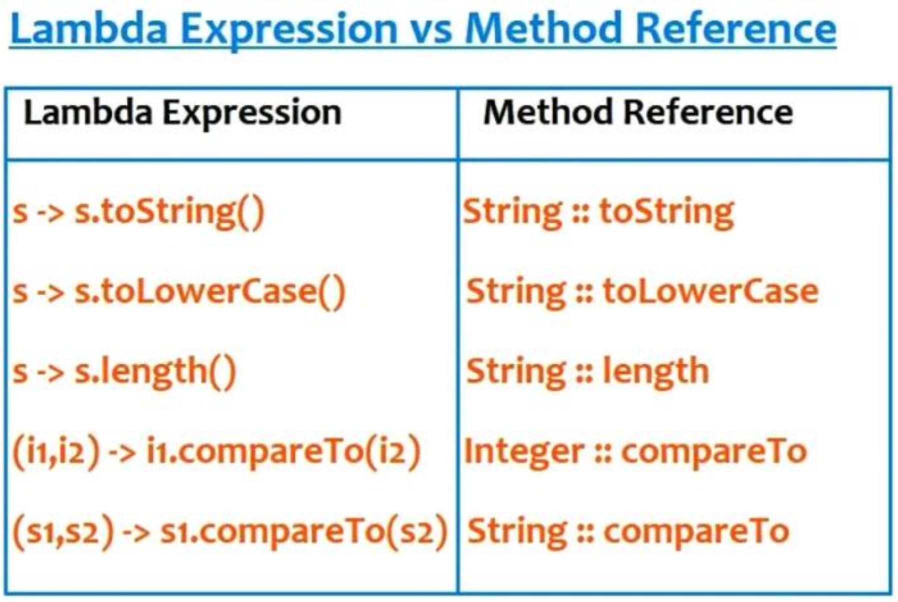


```java
public class ThreadExample {
    public static void main(String[] args) {
        Thread t= new Thread(ThreadExample::display);
        t.start();
    }

    public static void display(){
        System.out.println("hello");
    }
}
```

- In the above code, To write the method reference **Write the class name followed by two colon and then method name (ThreadExample is the class Name:: display)**
- That means **ThreadExample::display** is same as **()->display()**
- Here in this example, lamda doesn't take any arguments and the method doesn't take any arguments

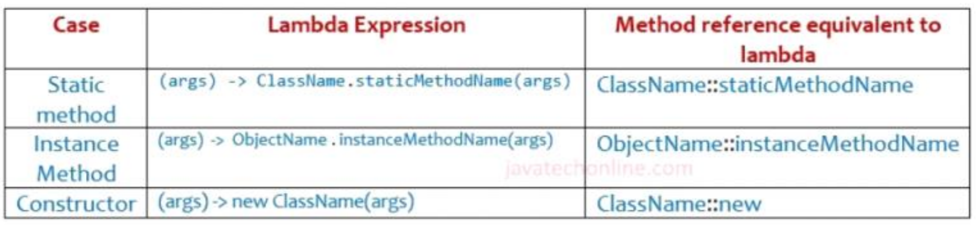
Syntax:
ContainingClass::staticMethodName

Some Examples of Method Reference



Lambda Expression vs Method Reference

Lambda Expression	Method Reference
s -> s.toString()	String :: toString
s -> s.toLowerCase()	String :: toLowerCase
s -> s.length()	String :: length
(i1,i2) -> i1.compareTo(i2)	Integer :: compareTo
(s1,s2) -> s1.compareTo(s2)	String :: compareTo

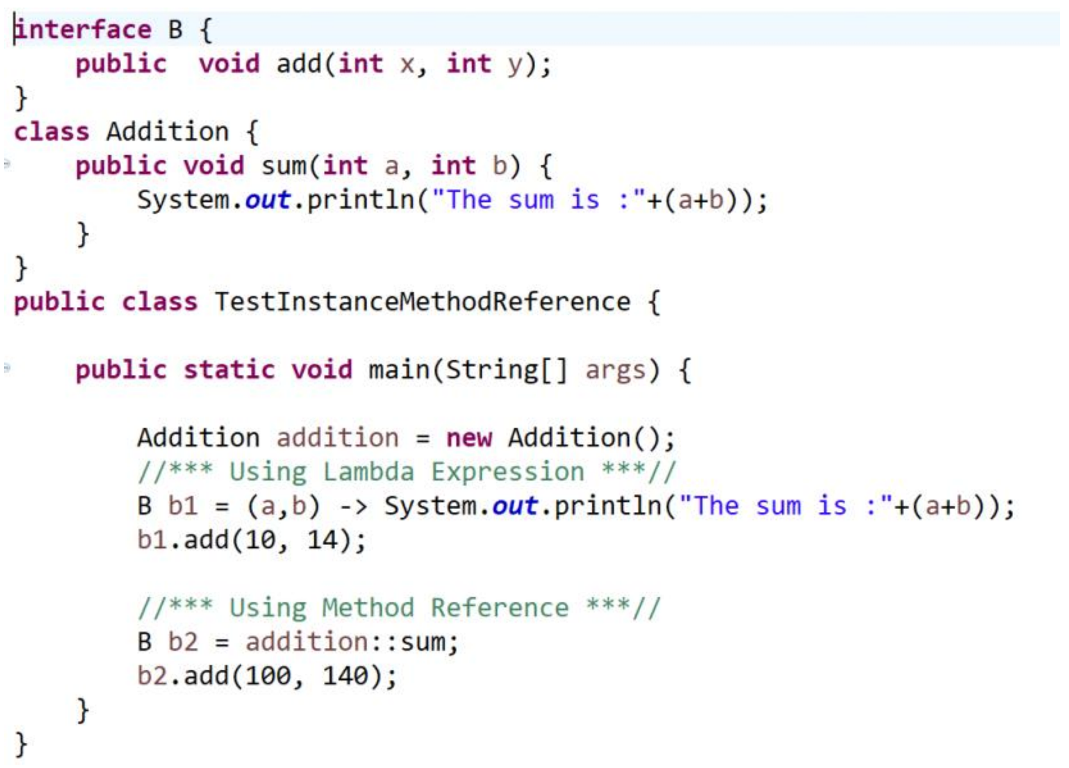


Case	Lambda Expression	Method reference equivalent to lambda
Static method	(args) -> ClassName.staticMethodName(args)	ClassName::staticMethodName
Instance Method	(args) -> ObjectName . instanceMethodName(args)	ObjectName::instanceMethodName
Constructor	(args) -> new ClassName(args)	ClassName::new

Types of Method References

Method reference to an Instance method of a class :

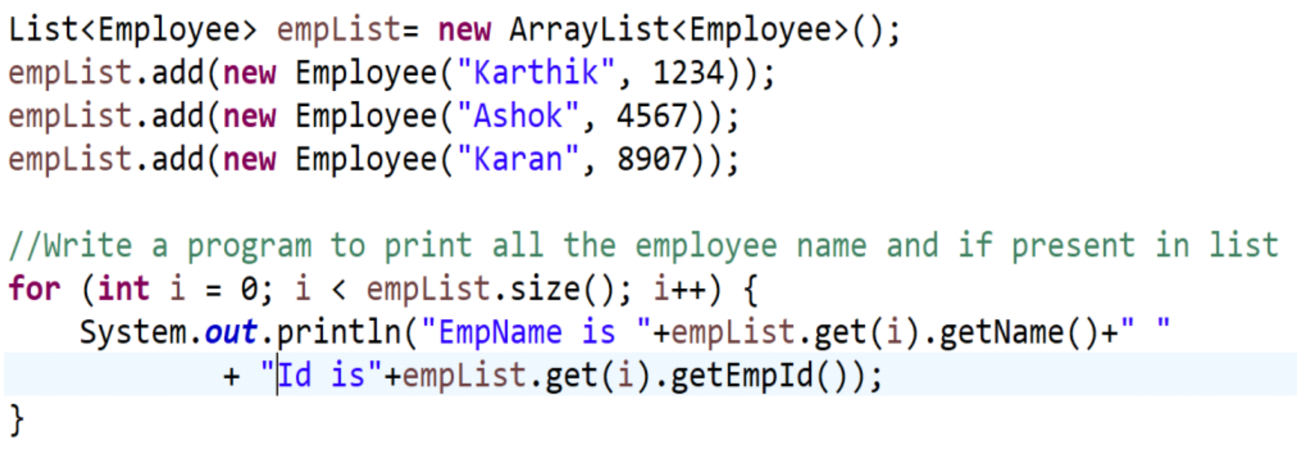
Inorder to refer the instance method we use object name followed by method name



```
interface B {
    public  void add(int x, int y);
}
class Addition {
    public void sum(int a, int b) {
        System.out.println("The sum is :"+(a+b));
    }
}
public class TestInstanceMethodReference {

    public static void main(String[] args) {

        Addition addition = new Addition();
        //*** Using Lambda Expression ***//
        B b1 = (a,b) -> System.out.println("The sum is :"+(a+b));
        b1.add(10, 14);

        //*** Using Method Reference ***//
        B b2 = addition::sum;
        b2.add(100, 140);
    }
}
```

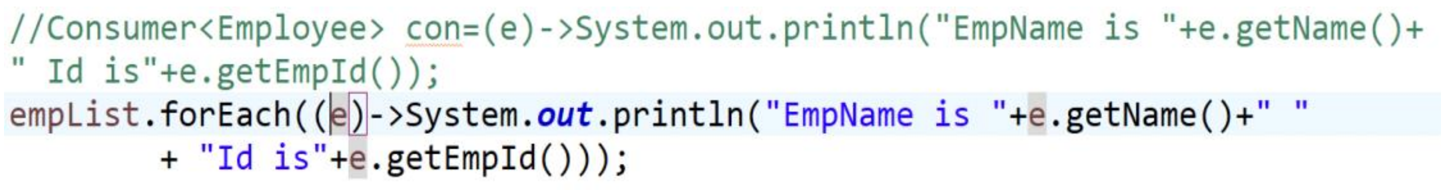
➢ Collection Improvements in Java8

➢ **forEach()**: This forEach method takes the argument as Consumer interface. That means we can write lamda and send lamda as a argument to this method.
Example without Lamda

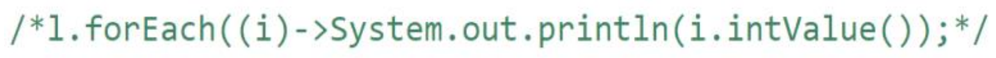


```
List<Employee> empList= new ArrayList<Employee>();
empList.add(new Employee("Karthik", 1234));
empList.add(new Employee("Ashok", 4567));
empList.add(new Employee("Karan", 8907));

//Write a program to print all the employee name and if present in list
for (int i = 0; i < empList.size(); i++) {
    System.out.println("EmpName is "+empList.get(i).getName()+" "
            + "Id is"+empList.get(i).getEmpId());
}
```

Code with ForEach:



```
//Consumer<Employee> con=(e)->System.out.println("EmpName is "+e.getName()+
" Id is"+e.getEmpId());
empList.forEach((e)->System.out.println("EmpName is "+e.getName()+" "
        + "Id is"+e.getEmpId()));
```



```
/*l.forEach((i)->System.out.println(i.intValue());*/
```

In the above example no need to pass what type of e object is, since we are calling the forEach on the collection object, compiler will understand what type of objects that collection will store.

➢ **removeif():**
The removeIf() method of ArrayList is used to remove all of the elements of this ArrayList that satisfies a given

predicate filter which is passed as a parameter to the method
Syntax:
    public boolean removeIf(Predicate filter)

```
System.out.println(empList.removeIf(p->p.getName().startsWith("K")));
System.out.println(empList);
```

➢ Java Stream API

- **Stream**: Stream is an interface introduced in Java8 under package called java.util.stream
    - A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result
    - If we want to represent a group of objects as a single entity then we should go for collection.
    - But if we want to process objects from the collection then we should go for streams

    - **Characteristics of streams**
        - Streams work perfectly with lambdas.
        - Streams don't store their elements.
        - Streams are immutable.
        - Streams are not reusable.
        - Streams don't support indexed access to their elements.
        - Streams are easily parallelizable.
        - Stream operations are lazy when possible.

    - We can create a stream from a source, perform operations on it and produce the result. Source may be a collection or an array or an I/O resource. Stream does never modify the source.
    - Remember that Streams doesn't store the data. We can't add or remove elements from streams. Therefore, we can't call them the data structures. They do only operations on the data.
    - Stream's operations are primarily divided in to two categories : Intermediate operations & Terminal operations.
    - Stream works on a concept of 'Pipeline of Operations'. A pipeline of operations consists of three things : a source, zero or more intermediate operations and a terminal operation.
    - In order to gain the performance while processing the large amount of data, Stream has a concept of parallel processing without writing any multi-threaded code.
    - All elements of a stream are not populated at a time. They are lazily populated as per demand because intermediate operations are not evaluated until terminal operation is invoked.
    - A stream is represented by the java.util.stream.Stream<T> interface. This works with objects only.
    - There are also specializations to work with primitive types, such as IntStream, LongStream, and DoubleStream

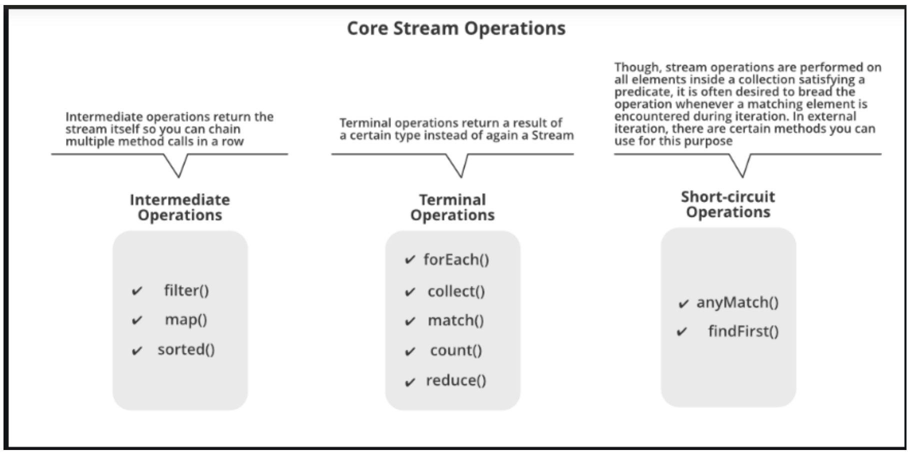
    - If we want to use the concept of streams then stream() is the method to be used. Stream is available as an interface.
      Stream s = c.stream();
      Here C represents collection object.

- **Creating the Stream**
    - **The first one is creating a stream** from a java.util.Collection implementation using the stream() method:
    - List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
      Stream<String> stream = words.stream();

    - **The second one is creating a stream from individual values:**
    - Stream<String> stream = Stream.of("hello","hola", "hallo", "ciao");

    - **The third one is creating a stream from an array:**

    - String[] words = {"hello", "hola", "hallo", "ciao"};
      2Stream<String> stream = Stream.of(words);


Core Stream Operations
Intermediate operations return the stream itself so you can chain multiple method calls in a row
Terminal operations return a result of a certain type instead of again a Stream
Though, stream operations are performed on all elements inside a collection satisfying a predicate, it is often desired to bread the operation whenever a matching element is encountered during iteration. In external iteration, there are certain methods you can use for this purpose
Intermediate Operations
✓ filter()
✓ map()
✓ sorted()
Terminal Operations
✓ forEach()
✓ collect()
✓ match()
✓ count()
✓ reduce()
Short-circuit Operations
✓ anyMatch()
✓ findFirst()

Intermediate Operation in Stream:

- Java 8 Stream intermediate operations return another Stream which allows you to call multiple operations in the form of a query
- Stream intermediate operations do not get executed until a terminal operation is invoked.
- All Intermediate operations are lazy, so they're not executed until a result of processing is actually needed.
- Here is the list of all Stream intermediate operations:
  - filter()
  - map()
  - flatMap()
  - distinct()
  - sorted()
  - peek()
  - limit()
  - skip()

filter()

- Returns a stream consisting of the elements of this stream that
- match the given predicate.
- Syntax:
  - Stream filter(Predicate predicate)

map()

- Returns a stream consisting of
- the results of applying the given function to the elements of this stream.
- method is an intermediate operation which is stateless and non-interfering with other elements in the Stream
- This method used to transform one set of values into another set of values by applying given function
- Transformation :- That's when map function is applied to Stream of T type (Stream<T>) then it get converted to Stream of R type (Stream<R>)
- One-to-One mapping :- map() method produces single value for each of the elements in the input stream hence it is referred as One-to-One mapping
- Example 1 :- a map function to square the input values can be applied to Stream of Integer consisting of natural numbers, then new Stream of Integer is returned consisting of its square values

```
// use map function to convert to Square value
List<Integer> squareValues = listOfNaturalNumbers
        .stream() // 1. get stream
        .map(n -> n*n) // 2. map intermediate operation
        .collect(Collectors.toList()); // 3. collect terminal operation
```

- Example 2 :- another example is finding ranks of Student from the input List of Students
- Note :- Number of elements returned in the new Stream after applying map function will always be equal to number of elements in the Original Stream

```
// getting ranks of each Student from List
List<Integer> rankList = studentList
        .stream() // 1. get stream
        .map(student -> student.rank) // 2. map intermediate operation
        .collect(Collectors.toList()); // 3. collect terminal operation
```

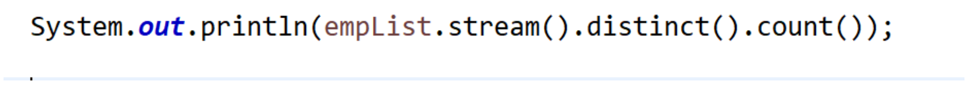
- Method signature :- <R> Stream<R> map(Function<? super T, ? extends R> mapper)

➢ **distinct()**
- Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
- For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.
- Syntax:

1 ◆ Stream distinct()

Write a Lamda to count how many distinct employees are present.

◆ In this case we need to override equals and hash code method in Employee class and provide the implementation for these methods on any of the Employee variables (like Name or Emp id etc)



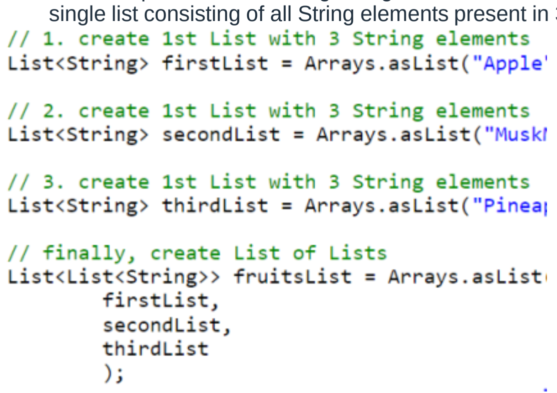
```
System.out.println(empList.stream().distinct().count());
```

flatMap()

- This Stream method is an **intermediate operation** which is **stateless** and **non-interfering** with other elements in the Stream
- **map** method does only **transformation**; but **flatMap** does **mapping** as well as **flattening** and this is main difference between these 2 map method of Stream API
- Suppose, we have **List of List of String elements**, in this case direct transformation isn't possible. So, we have to map first and then flatten to get **List of String elements**
- **Transformation** and **Flattening :-** When flatMap function is applied to **Stream of Stream of T** type (**Stream<Stream<T>>**) then it get converted to **Stream of R** type (**Stream<R>**) i.e.; transform to another stream and next flatten it
- **One-to-Many mapping :-** flatMap() method produces **stream of values** for each of the elements in the **input stream** hence it is referred as **One-to-Many** mapping
- **Note :-** Number of elements returned in the **new Stream** after transformation and flattening will always be **equal to** sum of elements in all sub-**Streams**
- Method signature :- **<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**

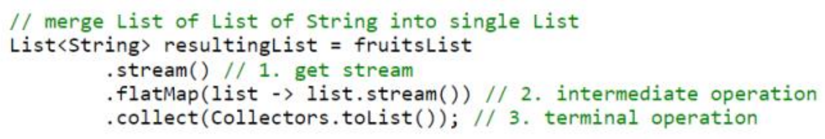
Flattening

- Flattening is basically converting all sub-lists into single list
- That's **Collection<Collection<T>>** to **Collection<T>**
- For example, 3 list containing String elements and these 3 lists is added to the outer list, then applying flatMap produces one single list consisting of all String elements present in 3 sub-lists



```
// 1. create 1st List with 3 String elements
List<String> firstList = Arrays.asList("Apple'

// 2. create 1st List with 3 String elements
List<String> secondList = Arrays.asList("Muskl

// 3. create 1st List with 3 String elements
List<String> thirdList = Arrays.asList("Pinea

// finally, create List of Lists
List<List<String>> fruitsList = Arrays.asList
        firstList,
        secondList,
        thirdList
        );
```

➢



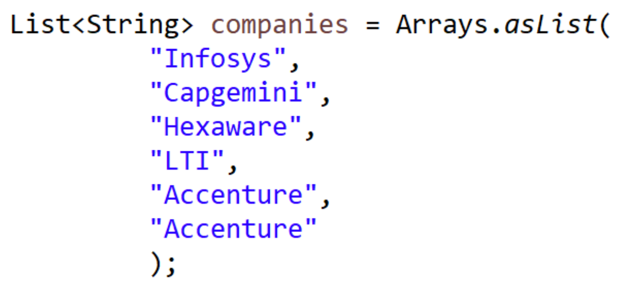
```
// merge List of List of String into single List
List<String> resultingList = fruitsList
        .stream() // 1. get stream
        .flatMap(list -> list.stream()) // 2. intermediate operation
        .collect(Collectors.toList()); // 3. terminal operation
```

➢ **sorted()**
- Returns a stream consisting of the elements of this stream, sorted according to the natural order.
- If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
- Note: For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made.

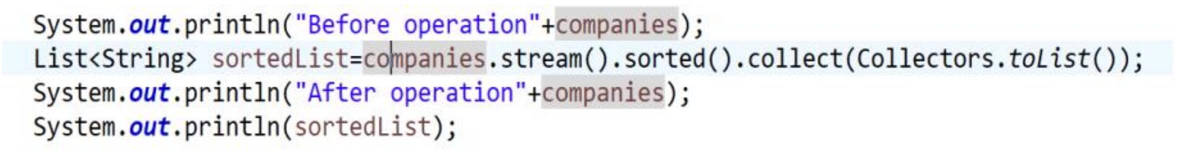
- We can even pass the comparator to this sorted method to get the desired order
- Syntax:

1 Stream sorted()

Write a Lamda to display the company names in Sorted order

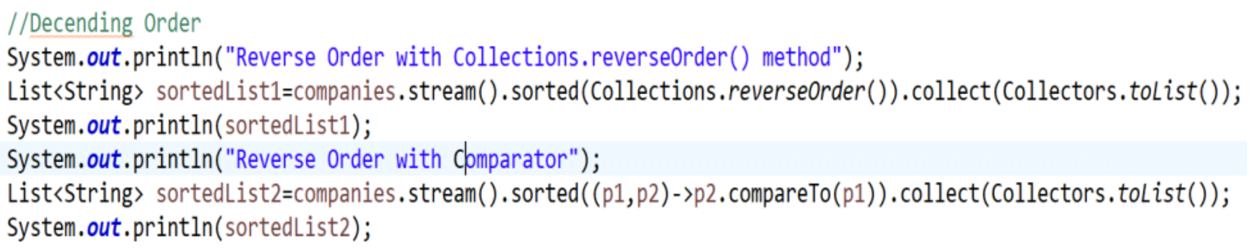


```
List<String> companies = Arrays.asList(
        "Infosys",
        "Capgemini",
        "Hexaware",
        "LTI",
        "Accenture",
        "Accenture"
        );
```



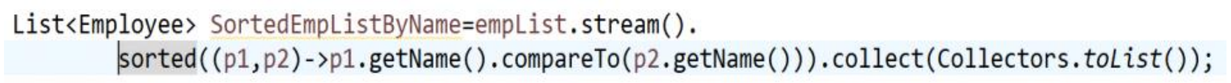
```
System.out.println("Before operation"+companies);
List<String> sortedList=companies.stream().sorted().collect(Collectors.toList());
System.out.println("After operation"+companies);
System.out.println(sortedList);
```

To get in Reverse Order we can pass comparator or Collections.reverseOrder



```
//Decending Order
System.out.println("Reverse Order with Collections.reverseOrder() method");
List<String> sortedList1=companies.stream().sorted(Collections.reverseOrder()).collect(Collectors.toList());
System.out.println(sortedList1);
System.out.println("Reverse Order with Comparator");
List<String> sortedList2=companies.stream().sorted((p1,p2)->p2.compareTo(p1)).collect(Collectors.toList());
System.out.println(sortedList2);
```

- **Write a Program to Sort Employees by Name.**



```
List<Employee> SortedEmpListByName=empList.stream().
        sorted((p1,p2)->p1.getName().compareTo(p2.getName())).collect(Collectors.toList());
```

- **limit()**
  - Returns a stream with the limited size given. It will truncate the remaining elements from the stream.
  - Note: limit() is suitable for sequential streams and cannot give good performance results for parallel streams.
  - Syntax:

1 ◆ Stream limit(long maxSize)

- **skip()**
  - This method skips the given n elements and returns a Stream. This is the most useful when want to perform any operations on last n records or lines from a List or Stream.
  - Syntax:

1 ◆ Stream skip(long n)

Terminal Operations:

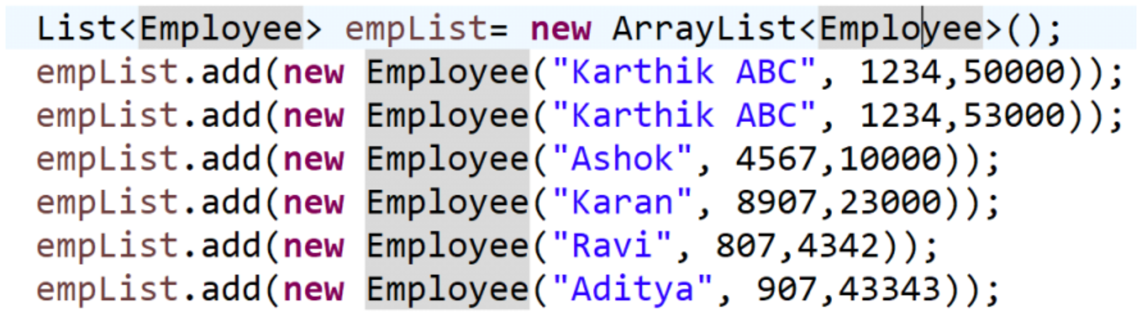
- Java-8 Stream terminal operations produces a non-stream, result such as primitive value, a collection or no value at all. Terminal operations are typically preceded by intermediate operations which return another Stream which allows operations to be connected in a form of a query.
- Here is the list of all Stream terminal operations:
  - toArray()
  - collect()

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

➢ **collect()**

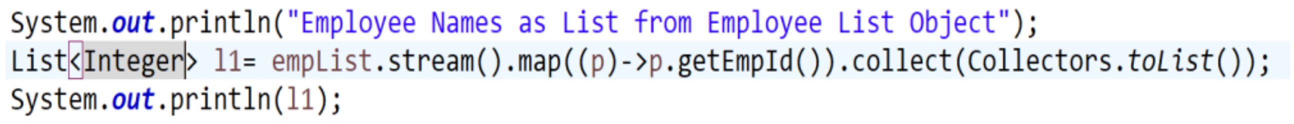
- This Stream method is a terminal operation which reads given stream and returns a collection like List or Set (or Map)
- collect() method accepts Collectors which is a final class with various utility methods to perform reduction on the given Stream
- Different collections that can be used in collect method are:
  1.Collectors.toList() to convert to List
  2.Collectors.toSet() to convert to Set
  3.Collectors.toCollection() to convert any Collection class like ArrayList/HashSet
  4.Collectors.toMap() to convert to Map

➢ **Write A Lamda to get all emp Id into List object from EmpList ?**



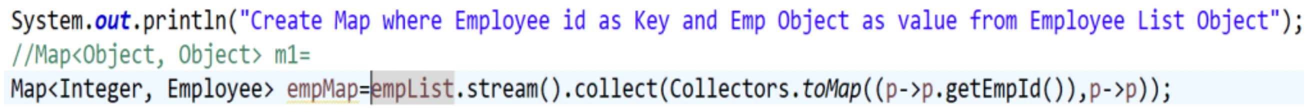
```java
List<Employee> empList= new ArrayList<Employee>();
empList.add(new Employee("Karthik ABC", 1234,50000));
empList.add(new Employee("Karthik ABC", 1234,53000));
empList.add(new Employee("Ashok", 4567,10000));
empList.add(new Employee("Karan", 8907,23000));
empList.add(new Employee("Ravi", 807,4342));
empList.add(new Employee("Aditya", 907,43343));
```

➢ **Creating Set:**



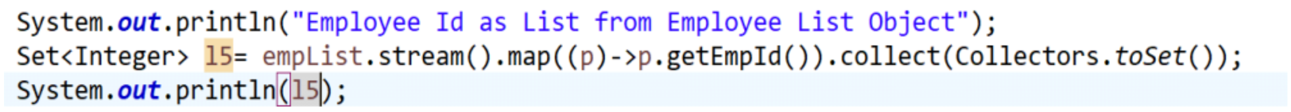
```java
System.out.println("Employee Names as List from Employee List Object");
List<Integer> l1= empList.stream().map((p)->p.getEmpId()).collect(Collectors.toList());
System.out.println(l1);
```

➢ **Creating Map:**



```java
System.out.println("Create Map where Employee id as Key and Emp Object as value from Employee List Object");
//Map<Object, Object> m1=
Map<Integer, Employee> empMap=empList.stream().collect(Collectors.toMap((p->p.getEmpId()),p->p));
```

➢ **Creating Set:**



```java
System.out.println("Employee Id as List from Employee List Object");
Set<Integer> l5= empList.stream().map((p)->p.getEmpId()).collect(Collectors.toSet());
System.out.println(l5);
```

➢ **forEach():**

- forEach() method is introduced in Collection and Map in addition to Stream, so we can iterate through elements of Collection, Map or Stream
- It is a terminal operation which is used to iterate through all elements present in the Stream
- Performs an action for each element of this stream
- Input to the forEach() method is Consumer which is Functional Interface
- For Sequential stream pipeline, this method follows original order of the source
- But for Parallel stream pipeline, this method doesn't guarantee original order of the source
  Syntax:

void forEach(Consumer action)

- **Write a Lamda to Print all the Employee names using Stream.**

  -
    ```
    empList.stream().forEach((p)->System.out.println(p.getName()));
    ```

- **Write a Lamda to filter even numbers and print to console using lambda expression**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

```
numbers // original source
.stream() // 1. get stream from source
.filter(i -> i%2 == 0) // 2. filter intermediate operation
.forEach(num -> System.out.println(num)); // 3. forEach terminal operation
```

- **count()**
  - This Stream method is a terminal operation which counts number of elements present in the stream
    Method signature :- long count()
    Example:

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

// get count
long count1 = numbers.stream().count();
```

- **Write a Lamda to Count how many employees are present whose name starts with K.**

```
Predicate<Employee> predicate=(p)->{
    System.out.println("Hi");
    if(p.getName().startsWith("K"))
        return true;
    else
        return false;
};
//System.out.println(predicate.test(11));;
System.out.println(empList.stream().filter(predicate).count());
```

(Or)

```
System.out.println(empList.stream().filter((p)->p.getName().startsWith("K")).count());
```

- **min()**
  - Returns the minimum value in the stream wrapped in an Optional or an empty one if the stream is empty.
  - It is a terminal operation which reads given stream and returns maximum element according to provided Comparator
  - This is the special case of reduction i.e.; Stream.reduce();
  - Method signature :- Optional<T> max(Comparator<? super T> comparator)
  - The min and Max method accepts a Comparator reference so we can write a lamda for compareTo method and pass that as a argument to min and max method

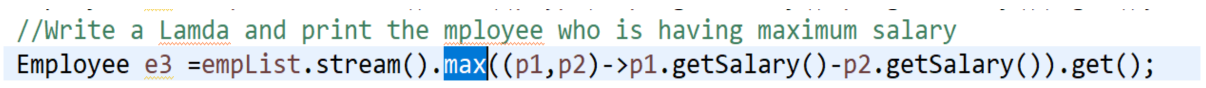
```
// list of integer numbers
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

// get minimum number from list of integers
int minNumber = numbers
        .stream()
        .min(Comparator.comparing(Integer::valueOf))
        .get();
```

- **Write a Lamda and print the employee who is having minimum salary ?**

```
//Write a Lamda and print the mployee who is having minimum salary
System.out.println(empList.stream().min((p1,p2)->p1.getSalary()-p2.getSalary()));

Employee e4 =empList.stream().min((p1,p2)->p1.getSalary()-p2.getSalary()).get();
```

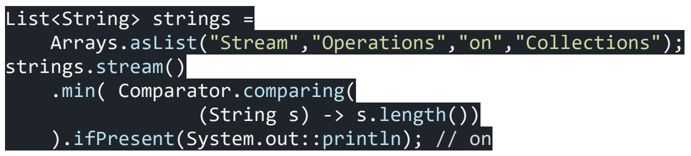
- **max()**
  - Returns the maximum value in the stream wrapped in an Optional or an empty one if the stream is empty.
  - This Stream method is a terminal operation which reads given stream and returns maximum element according to provided Comparator
  - This is the special case of reduction i.e.; Stream.reduce();
  - Method signature :- Optional<T> max(Comparator<? super T> comparator)

- **Write a Lamda and print the Employee who is having maximum salary ?**



```
//Write a Lamda and print the mployee who is having maximum salary
Employee e3 =empList.stream().max((p1,p2)->p1.getSalary()-p2.getSalary()).get();
```

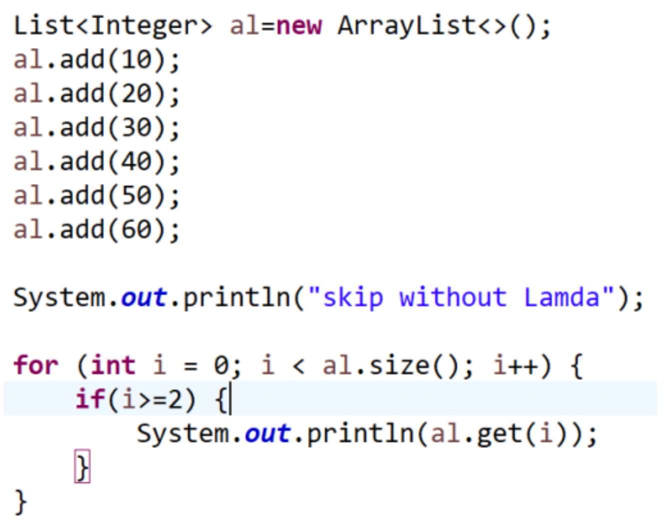
  - When we talk about primitives, it is easy to know which the minimum or maximum value is. But when we are talking about objects (of any kind), Java needs to know how to compare them to know which one is the maximum and the minimum. That's why the Stream interface needs a Comparator for max() and min()



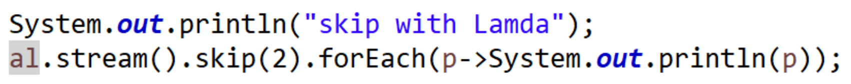
```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .min( Comparator.comparing(
                 (String s) -> s.length())
    ).ifPresent(System.out::println); // on
```

sum() returns the sum of the elements in the stream or zero if the stream is empty:

- **skip():**
  - This Stream method is an intermediate operation which skip/discards first n elements of the given stream
  - skip() method returns new stream consisting of remaining elements after skipping specified n elements from the original stream
  - Specified n can't be negative, otherwise IllegalArgumentException is thrown
  - If specified n is higher than the original size of the Stream then an empty Stream is returned
  - Method signature :- Stream<T> skip(long n)



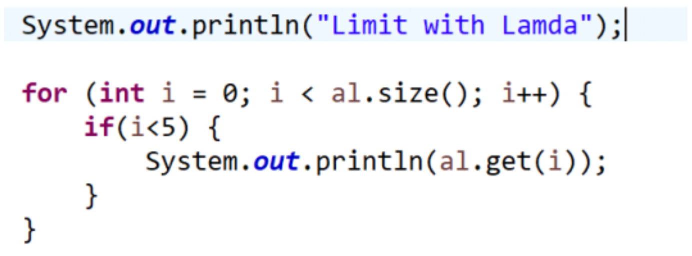
```
List<Integer> al=new ArrayList<>();
al.add(10);
al.add(20);
al.add(30);
al.add(40);
al.add(50);
al.add(60);

System.out.println("skip without Lamda");

for (int i = 0; i < al.size(); i++) {
    if(i>=2) {
        System.out.println(al.get(i));
    }
}
```



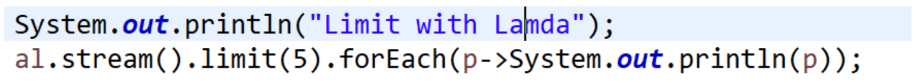
```
System.out.println("skip with Lamda");
al.stream().skip(2).forEach(p->System.out.println(p));
```

- **limit() method :**
  - This Stream method is an intermediate operation which limits to first n elements of the given stream
  - limit() method returns new stream consisting of elements not longer than the specified max size in the encounter order
  - Specified n can't be negative, otherwise IllegalArgumentException is thrown
  - Stream limit() method is stateful which means it is interfering as it has to keep the state of the items that are being picked up

- limit() method doesn't consume entire stream. As soon as, limit reaches the maximum number of elements it ignores the remaining elements. Hence, it is the short-circuit operation.
- Method signature :- Stream<T> limit(maxSize n)



```
System.out.println("Limit with Lamda");

for (int i = 0; i < al.size(); i++) {
    if(i<5) {
        System.out.println(al.get(i));
    }
}
```



```
System.out.println("Limit with Lamda");
al.stream().limit(5).forEach(p->System.out.println(p));
```

findFirst() method :

- This Stream method is a terminal operation which returns Optional instance describing first element of the given Stream
- If provided Stream has encounter-order then first element is returned (encounter-order depends on the source or intermediate operations)
- But if provided Stream has no-encounter-order then any element may be returned
- If provided Stream is empty or intermediate operation's result is empty then empty Optional is returned i.e.; Optional.empty()
- Note: Above behaviour is true for both sequential and parallel streams

findAny() method :

- This Stream method is a terminal operation which returns Optional instance describing any element from the given Stream
- If provided Stream has encounter-order then most of the times first element is returned but this behavior isn't guaranteed
- For Stream with no-encounter-order, any element may be returned
- The behaviour of this operation is explicitly nondeterministic, as it is free to select any element in the stream. For stable result, use findFirst() method as explained in above section
- If provided Stream is empty or intermediate operation's result is empty then empty Optional is returned i.e.; Optional.empty()

-

- Method signature :- Optional<T> findAny()

Stream forEachOrdered() method

Encounter order :-

- The order in which stream elements are supplied
- For example, if the original source to stream is ArrayList which preserves insertion order, then stream follows insertion order of ArrayList
- This Stream method is a terminal operation which is used to iterate through all elements present in the Stream in the encounter order
- Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order
- Input to the forEachOrdered() method is Consumer which is Functional Interface
- Irrespective of whether Stream is Sequential or Parallel, this method iterates through all elements in the encounter order, if one such exists i.e.; as stream elements supplied to forEachOrdered method
- forEachOrdered() method is non-interfering with other elements in the Stream
- Method signature :- void forEachOrdered(Consumer<? super T> action)
- **Stream forEachOrdered() method example :**
- Here, we will take a look at an example for forEachOrdered() method of Stream API
- We will iterate through all elements present using forEachOrdered method after obtaining Sequential

stream as well as Parallel stream
- Also, we will note time difference for processing in Sequential & Parallel order which is 20 ms & 14 ms respectively
- Another thing to note here is that, elements in List are printed as per original insertion order in both Sequential & Parallel stream

➢ **Stream reduce() method :**

- This Stream method is a terminal operation which performs reduction on the given stream and returns a reduced (or single) value
- There are 3 variants of reduce() method
- Method signature 1 :- Optional<T> reduce(BinaryOperator<T> accumulator)
- Method signature 2 :- T reduce(T identity, BinaryOperator<T> accumulator)
- Method signature 3 :- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
- reduce() method helps to derive sum of the Stream, searching max/mix element amongst Stream elements or finding average and performing String concatenation
- We can utilize this method for Map and reduce functionality
- This method performs a reduction on the elements of this stream, using an associative accumulation function. It returns an Optional describing the reduced value, if any.
- T reduce(T identity, BinaryOperator<T> accumulator)
-
- This method takes two parameters: the identity and the accumulator. The identity element is both the initial value of the reduction and the default result if there are no elements in the stream. The accumulator function takes two parameters: a partial result of the reduction and the next element of the stream. It returns a new partial result. The Stream.reduce method returns the result of the reduction.

➢ **The mapping() method**

mapping() is a static method of the Collectors class that returns a Collector. It converts a Collector accepting elements of one type to a Collector that accepts elements of another type

➢ **Optional**

- A NullpointerException is a common issue in java applications. To prevent this, we normally add frequent NULL checks in our code to check if a variable is not empty before we use it in our program. Optional provides a better approach to handle such situations
- Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.
- Java Optional class provides a way to deal with null values. It is used to represent a value is present or not. Java 8 added a new class Optional available in java.util package.
- Advantages of Java 8 Optional:
  - Null checks are not required.
  - No more NullPointerException at run-time.
  - We can develop clean and neat APIs.
  - No more Boiler plate code
- We can keep the method return type as Optional.
- Look at the below program where we get the null pointer exception.

```
    public static String m1() {
        return null;
    }
    public static void main(String[] args) {

        String s="Karthik";

        s=m1();
        if(s.contains("K"));{
            System.out.println("True");
        }
```

In the Above program we get null pointer exception when we check the condition if(s.contains("k")). This null pointer exception can be avoided using the below code

```
public static Optional<String> m1() {
    System.out.println("package1 A class m1 method");
    return Optional.ofNullable(null);
}
public static void main(String[] args) {

    String s="Karthik";

    Optional<String> op= m1();
    if(op.isPresent())
    {
        System.out.println(op.get());
    }
```

- Using the above code we can create the Optional object by calling the OfNullable method.
- **OfNullable**() : if the value is null then it returns empty optional, if not create the optional object by the element that we passed as generics.
- **isPresent():** This method returns true if the optional is not empty.
- **get()**: To get the value from the optional we should use the get() method call this method after when isPresent() returns true;
- ifPresent(Consumer c): This method can be used to perform some operation if the value from the optional object is not null.
- `m1().ifPresent(p->System.out.println(p));`
- **filter(predicate):** We can perform the filter operation if the optional is not empty
- `Optional<String> a= m1().filter((p)->p.startsWith("B"));`
- **orElse():** If the value of the optional object is null then using the below method we can return some default String value. In the below example when the value from the m1() method is null or empty then it returns "Hi"
- `String m=m1().orElse("Hi");`
- **orElseGet(Supplier):** here in the below example, m1() method is returning optional and the value inside optional is null then using the orElseget method we can write the supplier interface lamda and return the different value. So the output of the below m2 variable is **"Value from orElseGet"**

```java
public static Optional<String> m1() {
    System.out.println("package1 A class m1 method");
    return Optional.ofNullable(null);
}
public static void main(String[] args) {

    String m2=m1().orElseGet(()->"Value from orElseGet");
    System.out.println(m2);
```

Modified the program and m1() method is returning "abc" instead null then orElseget(Supplier) method return abc because the supplier gets invoked only when the option object value is null

```java
public static Optional<String> m1() {
    System.out.println("package1 A class m1 method");
    return Optional.ofNullable("abc");
}
public static void main(String[] args) {

    String m2=m1().orElseGet(()->"Value from orElseGet");
    System.out.println(m2);
```

➢ Local Date Time:

java.time.LocalDateTime class, introduced in Java 8, represents a local date-time object without timezone information. The LocalDateTime class in Java is an immutable date-time object that represents a date in the yyyy-MM-dd-HH-mm-ss.zzz format

now()

Use the now() method to get the current local date-time. Note that we can get the current local timestamp in another zone by passing the zone id

LocalDateTime now = LocalDateTime.now(); *//Current timestamp in UTC* LocalDateTime utcTimestamp =

➢ Create LocalDateTime with Values

To create a local timestamp with a specific date and time information – use of(year, month, day, hour, minute, second, nanosecond) method that is an overloaded method with optional arguments

```java
LocalDateTime localDateTime1 =
        LocalDateTime.of(2019, 03, 28, 14, 33, 48, 640000);

LocalDate date = LocalDate.of(2109, 03, 28);
LocalTime time = LocalTime.of(10, 34);
```

➢ Combine LocalDate and LocalTime

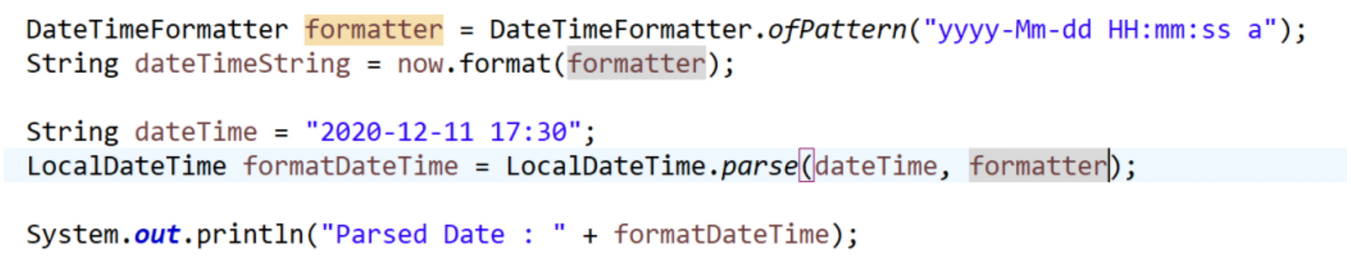
If we have separate instances of LocalDate and LocalTime classes, then we can combine them to obtain the instance of LocalDateTime

```java
LocalDate date = LocalDate.of(2109, 03, 28);
LocalTime time = LocalTime.of(10, 34);

LocalDateTime localDateTime5 = LocalDateTime.of(date, time);
```

Formatting the Date:

```java
//Formatting LocalDateTime to string in specified format
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-Mm-dd HH:mm:ss a");
String dateTimeString = now.format(formatter);
```

String to LocaDateTime



```java
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-Mm-dd HH:mm:ss a");
String dateTimeString = now.format(formatter);

String dateTime = "2020-12-11 17:30";
LocalDateTime formatDateTime = LocalDateTime.parse(dateTime, formatter);

System.out.println("Parsed Date : " + formatDateTime);
```

Concurrency API Enhancements

CompletableFuture

As mentioned before, CompletableFuture is one of the most important enhancements in Java 8. It implements the interfaces Future and CompletionStage, providing several functionalities related to futures out of the box. It is much easier to handle futures and promises using Java as it was before Java 8

Creation

It is possible to create a completable directly using its constructor:

```java
CompletableFuture completableFuture = new CompletableFuture();

```

or using factory methods where the sync or async mode is specified and its task or process is also passed:

```java
CompletableFuture completableFuture = CompletableFuture.supplyAsync( ( ) -> {
   // big computation task
   return "100";
} );

```

Getting results

In order to get the results of a CompletableFuture we have several options:

- Using the get() method:
-

```java
System.out.println( "get  " + cf.get() );

```

Will wait **for ever** until the CompletableFuture is completed or cancelled.

- Using the getNow(String fallback) method:
-

```java
System.out.println( "get now " + cf.getNow( "now" ) );

```

If the result of the computation is not present yet, the fallback passed as parameter is returned

ExecutorService invokeAll() API

invokeAll() method executes the given list of Callable tasks, returning a list of Future objects holding their status and results when all complete.

IO Enhancements:

Files.find()

- The path, starting file or folder.
- The maxDepth defined the maximum number of directory levels to search. If we put 1, which means the search for top-level or root folder only, ignore all its subfolders; If we want to search for all folder levels, put Integer.MAX_VALUE.
- The BiPredicate<Path, BasicFileAttributes> is for condition checking or filtering.
- The FileVisitOption tells if we want to follow symbolic links, default is no. We can put FileVisitOption.FOLLOW_LINKS to follow symbolic links