**Why default methods**

The oneliner for this could be “backward compatibility”.If JDK modifies an interface, then all classes which implements this interface will break.

For adding lambda expression in Java 8, JDK needs to add  methods(such as **foreach**) to List or collections Interface, but if you add this method to these interface, it will break millions lines of code as class which implements the interface, need to implement all its methods.

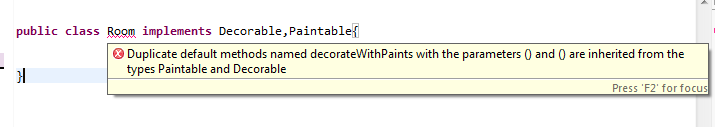
By adding default method in interface, you can provide default implementation of it without affecting implementing classes as it includes implementation of that method and any implementing class which needs that method can override it.

**What about multiple Inheritance?**

Adding default implementation to the interface can give rise to ambiguity in multiple inheritance. As two interface can provide same default method and there can be ambiguity while calling.Java 8 will give you compile time exception when this kind of situation will arise.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public interface Decorable {      default void decorateWithPaints()      {         System.out.println("Decorate using paints");      }     }    public interface Paintable  {      default void decorateWithPaints()      {          System.out.println("Decorate using paints");      }  }    public class Room implements Decorable,Paintable{      } |

so it will give you compile time error as below:

[](https://java2blog.com/wp-content/webpc-passthru.php?src=https://java2blog.com/wp-content/uploads/2014/06/RoomDefaultmultipleInheritance-1.png&nocache=1)

you can solve this compilation error by overriding decorateWithPaints method in Room class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Room implements Decorable,Paintable{        public void decorateWithPaints()      {          System.out.println("Decorate using paints");    }  } |

**Difference between default methods and abstract class**

Introduction of default methods to interface bridge gap between interface and abstract class.Now interface looks very similar to abstract classes but there are still differences. Lets list them

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Abstract class** | **Interface with default methods** |
| State of objects | Abstract class can hold state of object | Interface with default methods can not hold state of objects |
| Access Modifier | Abstract class methods can have public ,protected,private and default modifier | Interface methods are by default public. you can not use any other access modifier with it |
| Constructor | Abstract class can have constructor | Interface  can not have constructor |
| Member variables | It can have member variables |  |

# Java 8 Optional

|  |
| --- |
| package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;    public class JavaFindEmployeeMain {        public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Employee employee = findEmployee(employeeList,"Adam");          System.out.println("Employee name: "+employee.getName());      }        public static Employee findEmployee(List<Employee> employeeList,String name)      {          for(Employee e:employeeList)          {              if(e.getName().equalsIgnoreCase(name))              {                  return e;              }          }          return null;      }      public static List<Employee> createEmployeeList()      {          List<Employee> employeeList=new ArrayList<>();            Employee e1=new Employee("John",21);          Employee e2=new Employee("Martin",22);          Employee e3=new Employee("Mary",31);          Employee e4=new Employee("Stephan",18);          Employee e5=new Employee("Gary",26);            employeeList.add(e1);          employeeList.add(e2);          employeeList.add(e3);          employeeList.add(e4);          employeeList.add(e5);            return employeeList;        }  } |

When you run above program, you will get below output:

Exception in thread “main” java.lang.NullPointerException  
at org.arpit.java2blog.JavaOptionalMain.main(JavaOptionalMain.java:12)

As you can see, "Adam" is not present in employeeList, that’s why we are getting NullPointerException here.

Do you see the issue here? We forgot to check null when we tried to find employee in the list. This occurs more often when you call library function and it returns null and you forget to check it.

## Java 8 Optional

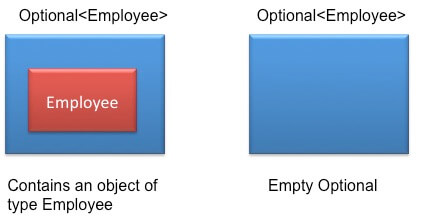
You can use Optional to solve this problem. You can change "JavaOptionalMain" as below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.Optional;    public class JavaOptionalMain {        public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt = findEmployee(employeeList,"Adam");          if(employeeOpt.isPresent())          {              Employee employee = employeeOpt.get();              System.out.println("Employee name: "+employee.getName());          }          else          {              System.out.println("There is no employee with name Adam");          }      }        public static Optional<Employee> findEmployee(List<Employee> employeeList,String name)      {          for(Employee e:employeeList)          {              if(e.getName().equalsIgnoreCase(name))              {                  return Optional.of(e);              }          }          return Optional.empty();      }      public static List<Employee> createEmployeeList()      {          List<Employee> employeeList=new ArrayList<>();            Employee e1=new Employee("John",21);          Employee e2=new Employee("Martin",22);          Employee e3=new Employee("Mary",31);          Employee e4=new Employee("Stephan",18);          Employee e5=new Employee("Gary",26);            employeeList.add(e1);          employeeList.add(e2);          employeeList.add(e3);          employeeList.add(e4);          employeeList.add(e5);            return employeeList;      }  } |

When you run above program, you will get below output:

There is no employee with name Adam

You might think that you could have handled null in JavaFindEmployeeMain as well but when you return Optional from method, it means that missing value can be expected from method.



## Ways to create Optional

There are multiple ways to create Optional.

### Empty Optional

You can create empty optional object using static factory method "empty"

|  |  |
| --- | --- |
| 1  2  3 | Optional<Employee> optCar = Optional.empty(); |

### Optional from a non-null value

You can create Optional from non-null value using static factory method "of"

|  |  |
| --- | --- |
| 1  2  3 | Optional<Employee> optCar = Optional.of(employee); |

If employee is null then above method will throw NullPointerException.

### Optional from null or non-null value

You can create Optional from null or non null value using static factory method "ofNullable"

|  |  |
| --- | --- |
| 1  2  3 | Optional<Employee> optCar = Optional.ofNullable(employee); |

## Getting value from Optional

You can use get() method to retrieve value from Optional but it is least safe. If value is not present then it will throw NoSuchElementException, so you need to make sure you call isPresent() method before you call get() method.

## Check value in Optional

You can check if there is value wrapped inside Optional using isPresent method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public static void main(String[] args)      {              List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt = findEmployee(employeeList,"Adam");          if(employeeOpt.isPresent())          {              Employee employee = employeeOpt.get();              System.out.println("Employee name: "+employee.getName());          }          else          {              System.out.println("There is no employee with name Adam");          }      } |

## Conditional action in Optional

You can use ifPresent method to execute action if value is present in Optional.  
Change main method in JavaOptionalMain as below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt1 = findEmployee(employeeList,"Adam");          Optional<Employee> employeeOpt2 = findEmployee(employeeList,"John");            employeeOpt1.ifPresent((employee)->System.out.println("Employee name: "+employee.getName()+" found in list"));          employeeOpt2.ifPresent((employee)->System.out.println("Employee name: "+employee.getName()+" found in list"));        } |

When you run this program, you will get below output:

Employee name: Dummy  
Employee name: John found in list

As you can see here, if employee name present in the list, then only we are printing employee name else it does not perform any action.

## Default value in Optional using orElse

You can return default value in case there is no value in Optional using orElse method.  
Change main method in JavaOptionalMain as below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt = findEmployee(employeeList,"Adam");          Employee employee1 = employeeOpt.orElse(new Employee("Dummy",0));          System.out.println("Employee name: "+employee1.getName());            Optional<Employee> employeeOpt2 = findEmployee(employeeList,"Martin");          Employee employee2= employeeOpt2.orElse(new Employee("Dummy",0));          System.out.println("Employee name: "+employee2.getName());        } |

When you run this program, you will get below output:

Employee name: Dummy  
Employee name: Martin

Please note that even if value is present in Optional, default object will be crerted.

## Default value in Optional using orElseGet

orElseGet is lazy counter part of orElse.It takes [supplier](https://java2blog.com/java-8-supplier-example/) as parameter and will be called only if value is not present in Optional.  
Change main method in JavaOptionalMain as below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt = findEmployee(employeeList,"Adam");          Employee employee1 = employeeOpt.orElseGet(()->new Employee("Dummy",0));          System.out.println("Employee name: "+employee1.getName());            Optional<Employee> employeeOpt2 = findEmployee(employeeList,"Martin");          Employee employee2 = employeeOpt2.orElseGet(()->new Employee("Dummy",0));          System.out.println("Employee name: "+employee2.getName());        } |

When you run this program, you will get below output:

Employee name: Dummy  
Employee name: Martin

## Throwing exception from Optional

You can use orElseThrow to throw exception in case Optional is empty. It is similar to get() method but in this case, you can choose to throw any Exception rathar than NoSuchMethodException.  
Change main method in JavaOptionalMain as below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args)      {          List<Employee> employeeList = createEmployeeList();          Optional<Employee> employeeOpt = findEmployee(employeeList,"Adam");          Employee employee1 = employeeOpt.orElseThrow(() -> new RuntimeException("Employee not found"));          System.out.println("Employee name: "+employee1.getName());            Optional<Employee> employeeOpt2 = findEmployee(employeeList,"Martin");          Employee employee2 = employeeOpt2.orElseThrow(() -> new RuntimeException("Employee not found"));          System.out.println("Employee name: "+employee2.getName());        } |

## Use of Optional in Java 8 APIs

There are lot of uses of Optional in Java 8 APIs.Let me provide you one example.  
Stream.findFirst() method returns an Optional with the first element of this stream, or an empty Optional if the stream is empty.

A Stream represents a sequence of elements supporting sequential and parallel aggregate operations. Stream does not store data, it operates on source data structures such as List, [Collection](https://java2blog.com/collections-java/), Array etc.

**💻 Awesome Tech Resources:**

* Looking for ⚒️ tech jobs? Go to our [job portal](https://lcsjobs.io/).
* Looking for tech events? Go to [tech events](https://aigents.co/opportunities/events) 🗓️ Calendar.️

Most stream operations accept [functional interfaces](https://java2blog.com/java-8-functional-interface-example/) that make it a perfect candidate for lambda expressions.

If you are not well versed with functional interfaces, lambda expressions, and method references, you may want to read the following tutorials before moving ahead.

1. [Java 8 Functional interfaces](https://java2blog.com/java-8-functional-interface-example/)
2. [Java 8 Lambda expressions](https://java2blog.com/lambda-expressions-in-java-8/)
3. [Java 8 Method references](https://java2blog.com/java-8-method-reference/)

## Types of Stream operations

There are two types of Stream operations.

1. Intermediate operations: return a stream that can be chained with other intermediate operations with dot .
2. Terminal operations: return void or non stream output.

Let’s understand with the help of simple example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package org.arpit.java2blog.stream;    import java.util.Arrays;  import java.util.List;    public class StreamOperations {        public static void main(String[] args) {          List<String> stringList = Arrays.asList("John", "Martin", "Mary", "Steve");            stringList.stream()                     .map((s) -> s.toUpperCase())                     .forEach(System.out::println);      }  } |

**Output:**

JOHN  
MARTIN  
MARY  
STEVE

Here,  
To perform a computation, stream operations are built into a Stream pipeline. Stream pipeline consists of:

1. source
2. zero or more intermediate operations
3. terminal operation.

In our example, Stream pipeline consists of:  
Source: stringList  
1 Intermediate operation: Map  
1 terminal operation: forEach

The below diagram will make it more clear.  
map is intermediate operation and foreach is terminal opertion.

Most stream operations accept parameters as that describes user-defined behaviour, such as [lambda expression](https://java2blog.com/lambda-expressions-in-java-8/)map((s)-&gt;s.toUpperCase()) is passed to map operation.

To get correct behavior, streams parameters should be:  
non-interfering: Stream source should not be modified while execution of Stream pipline. You can learn more about [interference](https://java2blog.com/java-8-parallel-stream/#Interference).  
Stateless: In most cases, lambda expressions should be stateless. Its output should not depend on state that might change during execution of Stream pipeline. I have already covered [stateful lambda expression](https://java2blog.com/java-8-parallel-stream/#Stateful_lambda_expressions) in Parallel Stream tutorial.

Read also: [Java 8 Parallel Stream](https://java2blog.com/java-8-parallel-stream/)

## Stream creation

There are multiple ways to create the Stream.

### Empty Stream

empty() method can be used to create an empty stream.

|  |  |
| --- | --- |
| 1  2  3 | Stream s = Stream.empty() |

It is generally used to return Stream with zero elements rather than null.

### Collection Stream

Stream can be created from Collection by calling .stream() or .parallelStream()

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | List stringList=Arrays.asList("Andy","Peter","Amy","Mary");    stringList.stream()  .map((s)->s.toUpperCase())  .forEach(System.out::println); |

stringList.stream() will return you regular object stream.

### Stream.of

You don’t need to create collection to get a Stream. You can also use [.of()](https://java2blog.com/java-8-stream-of/)

|  |  |
| --- | --- |
| 1  2  3 | Stream streamArray =Stream.of("X","Y","Z"); |

### Stream.generate()

generate() method accepts [Supplier](https://java2blog.com/java-8-supplier-example/) for element generation. It creates infinite Stream and you can limit it by calling limit() function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Stream<Integer> intStream=Stream.generate(() -> 1).limit(5);  intStream.forEach(System.out::println);  // Output  // 1  // 1  // 1  // 1  // 1 |

This will create Integer stream with 10 elements with value 1.

### Stream.iterate()

Stream.iterate() can also be used to generate infinite stream.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Stream<Integer> intStream = Stream.iterate(100 , n -> n+1).limit(5);  intStream.forEach(System.out::println);  // Output  // 100  // 101  // 102  // 103  // 104 |

First parameter of iterate method represents first element of the Stream. All the following elements will be generated by lambda expression n-&gt;n+1 and limit() is used to convert infinite Stream to finite Stream with 5 elements.

## Lazy evaluation

Streams are lazy; intermediate operation are not executed until terminal operation is encounterd.

Each intermediate operation generates a new stream, stores the provided operation or function. When terminal operation is invoked, stream pipeline execution starts and all the intermediate operations are executed one by one.

Let’s understand with the help of example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Stream<String> nameStream = Stream.of("mohan","john","vaibhav","amit");  Stream<String> nameStartJ = nameStream.map(String::toUpperCase)                                      .peek( e -> System.out.println(e))                                    .filter(s -> s.startsWith("J"));    System.out.println("Calling terminal operation: count");  long count = nameStartJ.count();  System.out.println("Count: "+ count);  // Output  // Calling terminal operation: count  // MOHAN  // JOHN  // VAIBHAV  // AMIT  // Count: 1 |

In preceding output, you can see that unless and until terminal operation count is called, nothing was printed on console.

In the preceding example, peek() method is used to print the element of stream. peek() method is generally used for logging and debugging purpose only.

## Order of operations

Let’s see how stream processes the order of operations.  
Could you guess output of the program?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Stream<String> nameStream = Stream.of("mohan","john","vaibhav","amit");  Stream<String> nameStartJ = nameStream.map(          (s) ->          {              System.out.println("Map: "+s);              return s.toUpperCase();            })          .filter(          (s) ->          {               System.out.println("Filter: "+s);               return s.startsWith("J");          }      );    Optional<String> findAny = nameStartJ.findAny();  System.out.println("Final output: "+findAny.get()); |

Output will be:

Map: mohan  
Filter: MOHAN  
Map: john  
Filter: JOHN  
JOHN

Here order of operations might be surprising. A common approach will be to perform intermediate operation on all elements and then perform next operation, but instead each element moves vertically.

This kind of behavior can reduce actual number of operation.  
**For example:**  
In preceding example, Strings vaibhav and amit did not go through map and filter operation as we already got result(findAny()) with String john.

Some of the intermediate operations such as [sorted](https://java2blog.com/java-stream-sorted/) are executed on the entire collection. As succeding operations might depend on the result of sorted operation.

## Primitive Streams

Apart from regular Stream, [Java 8](https://java2blog.com/java-8-tutorial/) also provides primitive Stream for int, long and double.  
Primitive Streams are:

1. IntStream for int
2. LongStream for long
3. DoubleStream for double

All the primitive Streams are similar to regular Stream with following differences.

* It supports few terminal aggregate functions such sum(), average(), etc.
* It accepts specialized function interface such as IntPredicate instead of Predicate, IntConsumer instead of Consumer.

Here is an example of an IntStream.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int sum = Arrays.stream(new int[] {1,2,3})                  .sum();  System.out.println(sum);    // Output  // 6 |

### Convert Stream to IntStream

You may need to convert Stream to IntStream to perform terminal aggregate operations such as sum or average. You can use mapToInt(), mapToLong() or mapToDouble() method to convert Stream to primitive Streams.  
Here is an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Stream.of("10","20","30")        .mapToInt(Integer::parseInt)        .average()        .ifPresent(System.out::println);  // Output  // 20.0 |

### Convert IntStream to Stream

You may need to convert IntStream to Stream to use it as any other datatype. You can use mapToObj() convert primitive Streams to regular Stream.  
Here is an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | String collect = IntStream.of(10,20,30)                            .mapToObj((i)->""+i)                            .collect(Collectors.joining("-"));  System.out.println(collect);  // Output  // 10-20-30 |

## Employee class

Consider a Employee class which has two fields name, age, listOfCities.

Here listOfCities denotes cities in which Employee has lived so far.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | package org.arpit.java2blog.stream;    import java.util.List;    public class Employee implements Comparable<Employee>{        private String name;      private int age;      private List<String> listOfCities;        public Employee(String name, int age,List<String> listOfCities) {          super();          this.name = name;          this.age = age;          this.listOfCities=listOfCities;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public int getAge() {          return age;      }        public void setAge(int age) {          this.age = age;      }        public List<String> getListOfCities() {          return listOfCities;      }        public void setListOfCities(List<String> listOfCities) {          this.listOfCities = listOfCities;      }        @Override      public String toString() {          return "Employee [name=" + name + ", age=" + age + "]";      }        @Override      public int compareTo(Employee o) {          return this.getName().compareTo(o.getName());      }  } |

This Employee class will be used in all succeeding examples.  
Let’s create employeesList on which we are going to perform intermediate and terminal operations.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package org.arpit.java2blog.stream;    import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;    public class StreamGetListOfEmployees {        public static void main(String[] args) {          List<Employee> employeesList=getListOfEmployees();            // Write stream code here      }        public static List<Employee> getListOfEmployees() {            List<Employee> listOfEmployees = new ArrayList<>();            Employee e1 = new Employee("Mohan", 24,Arrays.asList("Newyork","Banglore"));          Employee e2 = new Employee("John", 27,Arrays.asList("Paris","London"));          Employee e3 = new Employee("Vaibhav", 32,Arrays.asList("Pune","Seattle"));          Employee e4 = new Employee("Amit", 22,Arrays.asList("Chennai","Hyderabad"));            listOfEmployees.add(e1);          listOfEmployees.add(e2);          listOfEmployees.add(e3);          listOfEmployees.add(e4);            return listOfEmployees;      }  } |

## Common intemediate operations

### Map()

[Map() operation](https://java2blog.com/java-8-stream-map/) is used to convert Stream<T> to Stream<R>. It produces one output result of type 'R' for each input value of type 'T'. It takes [Function](https://java2blog.com/java-8-function-example/) interface as parameter.  
For example:  
You have stream of list of employees and you need a list of employee names, you simply need to convert Stream<Employee> to Stream<String>.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<String> employeeNames = employeesList.stream()                                  .map(e -> e.getName())                                 .collect(Collectors.toList());  System.out.println(employeeNames);    // Output  // [Mohan, John, Vaibhav, Amit] |

Logical representation of Map operation

You can also use map even if it produces result of same type.  
In case, you want employee name in uppercase, you can use another map() function to convert string to uppercase.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | List<String> employeeNames = employeesList.stream()                                  .map(e -> e.getName())                                  .map(s -> s.toUpperCase())                                  .collect(Collectors.toList());  System.out.println(employeeNames);    // Output  // [MOHAN, JOHN, VAIBHAV, AMIT] |

### Filter()

[Filter() operation](https://java2blog.com/java-8-stream-filter-examples/) is used to filter stream based on conditions. Filter method takes Predicate() interface which returns boolean value.  
Let’s say you want to employees whose name starts with ‘A’.  
You can write following functional code to achieve the same.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | List<String> employeeNames = employeesList.stream()                                  .map(e -> e.getName())                                  .filter(s -> s.startsWith("A"))                                 .collect(Collectors.toList());  System.out.println(employeeNames);    // Output  // [AMIT] |

Logical representation of Filter operation[![StreamFilter]

### sorted()

You can use [sorted()](https://java2blog.com/java-stream-sorted/) method to sort list of objects. sorted method without arguments sorts list in natural order. sorted() method also accepts comparator as parameter to support custom sorting.

💡 **Did you know?**

Natural order means sorting the list based on comparable interface implemented by list element type.  
**For example:**  
List<Integer> will be sorted on the basis of comparable interface implemented by Integer class.

Here is the sorted() method example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employees = employeesList.stream()                                              .sorted()                                              .collect(Collectors.toList());  System.out.println(employees);    // Output  // [Employee [name=Amit, age=22], Employee [name=John, age=27], Employee [name=Mohan, age=24], Employee [name=Vaibhav, age=32]] |

Here is the sorted() method example with [Comparator](https://java2blog.com/comparator-in-java/) as a parameter.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employees = employeesList.stream()                                .sorted((e1,e2)->e1.getAge() - e2.getAge())                                 .collect(Collectors.toList());  System.out.println(employees);    // Output  // [Employee [name=Amit, age=22], Employee [name=Mohan, age=24], Employee [name=John, age=27], Employee [name=Vaibhav, age=32]] |

You can also rewrite this with [method reference](https://java2blog.com/java-8-method-reference/) as below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employees = employeesList.stream()                                                  .sorted(Comparator.comparing(Employee::getAge))                                                  .collect(Collectors.toList());  System.out.println(employees);    // Output  // [Employee [name=Amit, age=22], Employee [name=Mohan, age=24], Employee [name=John, age=27], Employee [name=Vaibhav, age=32]] |

### limit()

You can use limit() to limit the number of elements in the stream.  
For example:  
limit(3) returns first 3 elements in the list.

Let’s see with the help of an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employees = employeesList.stream()                                       .limit(3)                                    .collect(Collectors.toList());  System.out.println(employees);    // Output  // [Employee [name=Mohan, age=24], Employee [name=John, age=27], Employee [name=Vaibhav, age=32]] |

### Skip()

skip(int n) method is used to discard first n elements from the stream.  
For example:  
skip(3) discards first 3 elements from stream.

Let’s see with help of example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employees = employeesList.stream()                                       .skip(3)                                    .collect(Collectors.toList());  System.out.println(employees);    // Output  // [Employee [name=Amit, age=22]] |

### flatmap()

map() operation generates one output for each input element.

**What if you want more than one output for each input?**  
[flatmap() operation](https://java2blog.com/java-8-stream-flatmap/) is exactly used for this purpose. It is used to map multiple-output for each input.  
**For example:**  
We want to accumulate list of cities in which all employees have lived. One employee could have lived in multiple cities so that we may have more than one city for each employee.

Let’s see with help of example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | List<String> listOfCities = employeesList.stream()                                             .flatMap(e -> e.getListOfCities().stream())                                             .collect(Collectors.toList());    System.out.println("listOfCities: " +listOfCities);    // Output  // listOfCities: [Newyork, Banglore, Paris, London, Pune, Seattle, Chennai, Hyderabad] |

## Common terminal operations

### foreach

[foreach()](https://java2blog.com/java-8-foreach-examples/) is terminal operation which is used to iterate over collection/stream of objects. It takes [consumer](https://java2blog.com/java-8-consumer-example/) as a parameter.

Let’s say you want to print elements of the stream.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | employeesList.stream()               .forEach(System.out::println);    // Output  // Employee [name=Mohan, age=24]  // Employee [name=John, age=27]  // Employee [name=Vaibhav, age=32]  // Employee [name=Amit, age=22] |

### collect

collect() is terminal operation which performs mutable reduction on the elements of Stream using Collector. [Collectors](https://java2blog.com/java-8-collectors-examples/) is utility class which provides inbuilt Collector.  
**For example:**  
Collectors.toList() provides a Collector which converts Stream to a list object.  
Following code accumultates Employee names into a [Arraylist](https://java2blog.com/arraylist-in-java-with-example/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<String> employeeNames = employeesList.stream()                                            .map(Employee::getName)                                            .collect(Collectors.toList());  System.out.println(employeeNames);    // Output  // [Mohan, John, Vaibhav, Amit] |

### Reduce

The reduce operation combines all elements of Stream and produces single result.  
Java 8 has three overloaded version of reduce method.

1. Optional&lt;T&gt; reduce(BinaryOperator&lt;T&gt; accumulator): This method takes BinaryOperator accumulator function. BinaryOperator is BiFunction where both the operands are of same type. First parameter is result till current execution, and second parameter is the current element of the Stream.

Let’s find name of Person with minimum age.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | employeesList.stream()  .reduce( (e1,e2)-> (e1.getAge() < e2.getAge()? e1:e2))  .ifPresent(System.out::println);  // Output  // Employee [name=Amit, age=22] |

1. T reduce(T identity, BinaryOperator<T> accumulator): This method takes identity value and accumulator function. identity value is initial value of the reduction. If Stream is empty,then identity value is the result.  
   Let’s find sum of all ages of Employees

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int sumAge = employeesList.stream()  .mapToInt(Employee::getAge)  .reduce(0, (age1,age2)-> (age1 + age2));    System.out.println("Sum of ages of all Employees: "+sumAge);  // Output  // Sum of ages of all Employees: 105 |

1. &lt;U&gt; U reduce(U identity, BiFunction&lt;U,? super T,U&gt; accumulator, BinaryOperator&lt;U&gt; combiner): This method takes identity value and accumulator function and combiner. Combiner is mainy used in case of [Parallel Streams](https://java2blog.com/java-8-parallel-stream/). Combiner comibnes the result of sub-stream that run in Parallel.

### count

count() is used to count number of elements in the stream.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | long empCountStartJ = employeesList.stream()                                     .map(Employee::getName)                                     .filter(s -> s.startsWith("J"))                                     .count();  System.out.println(empCountStartJ);    // Output  // 1 |

### allMatch()

allMatch() returns true when all the elements in the stream meet provided condition.

This is a short-circuiting terminal operation because operation stops as soon as it encounters any unmatched element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | boolean allMatch = employeesList.stream()                                  .allMatch(e ->e.getAge()>18);    System.out.println("Are all the employess adult: " +allMatch);    // Output  // Are all the employess adult: true |

### nonMatch()

nonMatch() returns true when all the elements in the stream do not meet provided condition.

This is a short-circuiting terminal operation because operation stops as soon as it encounters any matched element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | boolean noneMatch = employeesList.stream()                                   .noneMatch(e ->e.getAge()>60);    System.out.println("Are all the employess below 60: " +noneMatch);    // Output  // Are all the employess below 60: true |

### anyMatch()

anyMatch() returns true when any element in the stream meets provided condition.

This is a short-circuiting terminal operation because operation stops as soon as it encounters any matched element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | boolean anyMatch = employeesList.stream()                                   .anyMatch(e ->e.getAge()>30);    System.out.println("is any employee's age greater than 30: " +anyMatch);    // Output  // is any employee's age greater than 30: true |

### min()

min(Comparator) returns minimum element in the stream based on the provided comparator. It returns an object which contains actual value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Optional<Employee> minEmpOpt = employeesList.stream()                                              .min(Comparator.comparing(Employee::getAge));    Employee minAgeEmp = minEmpOpt.get();  System.out.println("Employee with minimum age is: " +minAgeEmp);    // Output  // Employee with minimum age is: Employee [name=Amit, age=22] |

### max()

max(Comparator) returns maximum element in the stream based on the provided comparator. It returns an object which contains actual value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Optional<Employee> maxEmpOpt = employeesList.stream()                                              .max(Comparator.comparing(Employee::getAge));    Employee maxAgeEmp = maxEmpOpt.get();  System.out.println("Employee with maxium age is: " +maxAgeEmp);    // Output  // Employee with maxium age is: Employee [name=Vaibhav, age=32] |

## Parallel Streams

You can create Parallel Stream using .parallel() method on Stream object in java.  
Here is an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int[] array= {1,2,3,4,5};    System.out.println("=================================");  System.out.println("Using Parallel Stream");  System.out.println("=================================");  IntStream intParallelStream=Arrays.stream(array).parallel();  intParallelStream.forEach((s)->  {      System.out.println(s+" "+Thread.currentThread().getName());  }  ); |

Here is a comprehensive article on [Java 8 Parallel Stream](https://java2blog.com/java-8-parallel-stream/).

## Exercises

Let’s practice some exercises on Stream.

### Exercise 1

Given a list of employees, you need to find all the employees whose age is greater than 30 and print the employee names.(Java 8 APIs only)  
**Answer:**

[Hide Answers](https://java2blog.com/java-8-stream/)

You can simply do it using below statement.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | List<String> employeeFilteredList = employeeList.stream()                                            .filter(e->e.getAge()>30)                                            .map(Employee::getName)                                            .collect(Collectors.toList());  System.out.println(employeeFilteredList);    // Output  // Vaibhav |

### Exercise 2

Given the list of employees, find the count of employees with age greater than 25?  
**Answer:**

[Hide Answers](https://java2blog.com/java-8-stream/)

You can use combination of filter and count to find this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employeeList = createEmployeeList();  long count = employeeList.stream()                           .filter(e->e.getAge()>25)                           .count();  System.out.println("Number of employees with age 25 are : "+count);  // Output  // Number of employees with age 25 are : 2 |

### Exercise 3

Given the list of employees, find the employee whose name is John.

[Hide Answers](https://java2blog.com/java-8-stream/)

**Answer:**  
It is again very simple logic, change the main function in above class as following.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Employee> employeeList = createEmployeeList();  Optional<Employee> e1 = employeeList.stream()            .               .filter(e->e.getName().equalsIgnoreCase("John"))                                            .findAny();    if(e1.isPresent())        System.out.println(e1.get()); |

### Exercise 4

Given a list of employees, You need to find highest age of employee?  
**Answer:**

[Hide Answers](https://java2blog.com/java-8-stream/)

It is again very simple logic, change the main function in above class as following.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | List<Employee> employeeList = createEmployeeList();  OptionalInt max = employeeList.stream()                                .mapToInt(Employee::getAge)                                .max();    if(max.isPresent())         System.out.println("Maximum age of Employee: "+max.getAsInt());  // Output:  // Maximum age of Employee: 32 |

### Excercise 5

Given a list of employees, you need sort employee list by age? Use java 8 APIs only  
**Answer:**

[Hide Answers](https://java2blog.com/java-8-stream/)

You can simply use sort method of list to sort the list of employees.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | List<Employee> employeesList = createEmployeeList();  List<Employee> sortedList = employeesList.stream()                                           .sorted((e1,e2)->e1.getAge()-e2.getAge())                                           .collect(Collectors.toList());  sortedList.forEach(System.out::println);    // Output  // Employee [name=Amit, age=22]  // Employee [name=Mohan, age=24]  // Employee [name=John, age=27]  // Employee [name=Vaibhav, age=32] |

### Excercise 6

Given the list of Employees, you need to join the all employee names with ","?  
**Answer:**

[Hide Answers](https://java2blog.com/java-8-stream/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | List<Employee> employeeList = createEmployeeList();  List<String> employeeNames = employeeList                       .stream()                       .map(Employee::getName)                       .collect(Collectors.toList());  String employeeNamesStr = String.join(",", employeeNames);  System.out.println("Employees are: "+employeeNamesStr);  // Output  // Amit,Mohan,John,Vaibhav |

Output:

Employees are: John,Martin,Mary,Stephan,Gary

### Excercise 7

Given the list of employees, you need to group them by name

[Hide Answers](https://java2blog.com/java-8-stream/)

**Answer:**

You can use [Collections.groupBy()](https://java2blog.com/how-to-do-groupby-in-java/) to group list of employees by employee name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class MaximumUsingStreamMain {  public static void main(String args[])  {  List<Employee> employeeList = createEmployeeList();  Map<String, List<Employee>> map = employeeList.stream()                                .collect(Collectors.groupingBy(Employee::getName));  map.forEach((name,employeeListTemp)->System.out.println("Name: "+name+" ==>"+employeeListTemp));  }    public static List<Employee> createEmployeeList()  {  List<Employee> employeeList=new ArrayList<>();    Employee e1=new Employee("John",21);  Employee e2=new Employee("Martin",19);  Employee e3=new Employee("Mary",31);  Employee e4=new Employee("Mary",18);  Employee e5=new Employee("John",26);    employeeList.add(e1);  employeeList.add(e2);  employeeList.add(e3);  employeeList.add(e4);  employeeList.add(e5);    return employeeList;  }  } |

Output:

Name: John ==>[Employee Name: John age: 21, Employee Name: John age: 26] Name: Martin ==>[Employee Name: Martin age: 19] Name: Mary ==>[Employee Name: Mary age: 31, Employee Name: Mary age: 18]

Sorting………..

1. sorted(): Returns a stream having elements sorted by natural order
2. sorted(Comparator<? super T> comparator): Returns a stream having elements sorted by provided comparator

💡 **Did you know?**

If you sort list of Integers using Stream.sorted() method then [Comparable](https://java2blog.com/comparable-in-java/) interface, implemented by Integer class, will define natural ordering of list of Integers .

Let’s understand how can we use Stream.sorted() to sort list of elements.

## Sort List of Integers

We can simply use sorted() method to sort list of integers.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<Integer> result = listOfIntegers.stream()                                           .sorted()                                           .collect(Collectors.toList()); |

Here, List of Integers is sorted by [Comparable](https://java2blog.com/comparable-in-java/) interface implemented by Integer class.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public final class Integer extends Number implements Comparable<Integer> {  ....  /\*\*       \* Compares two {@code Integer} objects numerically.       \*       \* @param   anotherInteger   the {@code Integer} to be compared.       \* @return  the value {@code 0} if this {@code Integer} is       \*          equal to the argument {@code Integer}; a value less than       \*          {@code 0} if this {@code Integer} is numerically less       \*          than the argument {@code Integer}; and a value greater       \*          than {@code 0} if this {@code Integer} is numerically       \*           greater than the argument {@code Integer} (signed       \*           comparison).       \* @since   1.2       \*/      public int compareTo(Integer anotherInteger) {          return compare(this.value, anotherInteger.value);      }        /\*\*       \* Compares two {@code int} values numerically.       \* The value returned is identical to what would be returned by:       \*       \* Integer.valueOf(x).compareTo(Integer.valueOf(y))       \*       \*       \* @param  x the first {@code int} to compare       \* @param  y the second {@code int} to compare       \* @return the value {@code 0} if {@code x == y};       \*         a value less than {@code 0} if {@code x < y}; and       \*         a value greater than {@code 0} if {@code x > y}       \* @since 1.7       \*/      public static int compare(int x, int y) {          return (x < y) ? -1 : ((x == y) ? 0 : 1);      } |

As you can see, Integers are compared on this basis of (x &lt; y) ? -1 : ((x == y) ? 0 : 1) logic.

You can pass Comparator.reverseOrder() to sorted method to reverse sort list of Integers.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<Integer> reverseOrder = listOfIntegers.stream()                                                 .sorted(Comparator.reverseOrder())                                                 .collect(Collectors.toList()); |

Comparator.reverseOrder() is [static](https://java2blog.com/static-keyword-in-java/) method and provides a comparator that imposes reverse of natural ordering.

Let’s see complete example to sort list of integers.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package org.arpit.java2blog;    import java.util.Arrays;  import java.util.Comparator;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfIntegers {        public static void main(String[] args) {          List<Integer> listOfIntegers = Arrays.asList(new Integer[] {40,34,21,37,20});            List<Integer> result = listOfIntegers.stream()                                                  .sorted()                                                  .collect(Collectors.toList());          System.out.println(result);            List<Integer> reverseOrder = listOfIntegers.stream()                                                       .sorted(Comparator.reverseOrder())                                                       .collect(Collectors.toList());          System.out.println(reverseOrder);      }    } |

**Output:**

[20, 21, 34, 37, 40] [40, 37, 34, 21, 20]

## Sort List of Strings

We can simply use sorted() method to sort list of Strings.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<String> result = listOfStrings.stream()                                           .sorted()                                           .collect(Collectors.toList()); |

Here, List of Strings is sorted by [Comparable](https://java2blog.com/comparable-in-java/) interface implemented by String class.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public final class String      implements java.io.Serializable, Comparable<String>, CharSequence {  ....  /\*\*       \* Compares two strings lexicographically.       \*       \* @param   anotherString   the {@code String} to be compared.       \* @return  the value {@code 0} if the argument string is equal to       \*          this string; a value less than {@code 0} if this string       \*          is lexicographically less than the string argument; and a       \*          value greater than {@code 0} if this string is       \*          lexicographically greater than the string argument.       \*/      public int compareTo(String anotherString) {          int len1 = value.length;          int len2 = anotherString.value.length;          int lim = Math.min(len1, len2);          char v1[] = value;          char v2[] = anotherString.value;            int k = 0;          while (k < lim) {              char c1 = v1[k];              char c2 = v2[k];              if (c1 != c2) {                  return c1 - c2;              }              k++;          }          return len1 - len2;      } |

As you can see, Integers are compared on this basis of (x &lt; y) ? -1 : ((x == y) ? 0 : 1) logic.

You can pass Comparator.reverseOrder() to sorted method to reverse sort list of Integers.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<String> reverseOrder = listOfStrings.stream()                                                 .sorted(Comparator.reverseOrder())                                                 .collect(Collectors.toList()); |

Comparator.reverseOrder() is [static](https://java2blog.com/static-keyword-in-java/) method and provides a comparator that imposes reverse of natural ordering.

Let’s see complete example to sort list of integers.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package org.arpit.java2blog;    import java.util.Arrays;  import java.util.Comparator;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfStrings {        public static void main(String[] args) {          List<String> listOfLanguages = Arrays.asList(new String[] { "Python", "C++", "Java", "PHP" });            List<String> sortedListOfLanguages = listOfLanguages.stream()                                                                .sorted()                                                                .collect(Collectors.toList());          System.out.println(sortedListOfLanguages);            List<String> sortedListOfLanguagesRev = listOfLanguages.stream()                                                                   .sorted(Comparator.reverseOrder())                                                                   .collect(Collectors.toList());          System.out.println(sortedListOfLanguagesRev);      }    } |

**Output:**

[C++, Java, PHP, Python] [Python, PHP, Java, C++]

## Sort List of custom objects

### Sort list of Students by natural order

Create a class named Student.java

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package org.arpit.java2blog;    public class Student{        String name;      int age;        public Student(String name, int age) {          super();          this.name = name;          this.age = age;      }      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      public int getAge() {          return age;      }      public void setAge(int age) {          this.age = age;      }    *@Override*      public String toString() {          return "Student [name=" + name + ", age=" + age + "]";      }  } |

Let’s use Stream’s sorted() to sort the list of Students now.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfStudents {        public static void main(String[] args) {          List<Student> studentsList = getListOfStudents();            List<Student> sortedStudentsList= studentsList.stream()                                                            .sorted()                                                            .collect(Collectors.toList());          System.out.println(sortedStudentsList);      }        public static List<Student> getListOfStudents()      {          List<Student> studentList=new ArrayList<>();            Student s1=new Student("Peter",21);          Student s2=new Student("Harshal",18);          Student s3=new Student("Andy",17);          Student s4=new Student("Mary",20);          Student s5=new Student("Peter",19);            studentList.add(s1);          studentList.add(s2);          studentList.add(s3);          studentList.add(s4);          studentList.add(s5);            return studentList;      }  } |

Let’s run above program.

Exception in thread “main” java.lang.ClassCastException: org.arpit.java2blog.Student cannot be cast to java.lang.Comparable  
at java.util.Comparators$NaturalOrderComparator.compare(Comparators.java:47)  
at java.util.TimSort.countRunAndMakeAscending(TimSort.java:355)  
at java.util.TimSort.sort(TimSort.java:220)  
at java.util.Arrays.sort(Arrays.java:1512)  
at java.util.stream.SortedOps$SizedRefSortingSink.end(SortedOps.java:353)  
at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:483)  
at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:472)  
at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)  
at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)  
at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:566)  
at org.arpit.java2blog.SortListOfStudents.main(SortListOfStudents.java:14)

**Did you know why we got exception here?**  
We got the ClassCastException exception because we did not implement Comparable interface in Student class.

Let’s implement [Comparable](https://java2blog.com/comparable-in-java/) interface in Student class.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog;    public class Student implements Comparable<Student>{        String name;      int age;        public Student(String name, int age) {          super();          this.name = name;          this.age = age;      }      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      public int getAge() {          return age;      }      public void setAge(int age) {          this.age = age;      }    *@Override*      public String toString() {          return "Student [name=" + name + ", age=" + age + "]";      }    *@Override*      public int compareTo(Student o) {          return this.getName().compareTo(o.getName());      }    } |

Run SortListOfStudents again, and you will get below output.

[Student [name=Andy, age=17], Student [name=Harshal, age=18], Student [name=Mary, age=20], Student [name=Peter, age=21], Student [name=Peter, age=19]]

As you can see, list of Students is sorted by student name.

### Sort list of Students by reverse natural order

Sort list of Students by name in descending order.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfStudents {        public static void main(String[] args) {          List<Student> studentsList = getListOfStudents();            List<Student> sortedListOfStudent2 = listOfStudents.stream()                                                             .sorted(Comparator.reverseOrder())                                                             .collect(Collectors.toList());          System.out.println(sortedStudentsList);      }        public static List<Student> getListOfStudents()      {          List<Student> studentList=new ArrayList<>();            Student s1=new Student("Peter",21);          Student s2=new Student("Harshal",18);          Student s3=new Student("Andy",17);          Student s4=new Student("Mary",20);          Student s5=new Student("Peter",19);            studentList.add(s1);          studentList.add(s2);          studentList.add(s3);          studentList.add(s4);          studentList.add(s5);            return studentList;      }  } |

**Output:**

[Student [name=Peter, age=21], Student [name=Peter, age=19], Student [name=Mary, age=20], Student [name=Harshal, age=18], Student [name=Andy, age=17]]

### Sort list of Students by age using comparator

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfStudents {        public static void main(String[] args) {          List<Student> studentsList = getListOfStudents();            List<Student> studentsListByAge = studentsList.stream()                                                           .sorted((s1,s2) -> s1.getAge()-s2.getAge())                                                           .collect(Collectors.toList());          System.out.println(studentsListByAge);      }        public static List<Student> getListOfStudents()      {          List<Student> studentList=new ArrayList<>();            Student s1=new Student("Peter",21);          Student s2=new Student("Harshal",18);          Student s3=new Student("Andy",17);          Student s4=new Student("Mary",20);          Student s5=new Student("Peter",19);            studentList.add(s1);          studentList.add(s2);          studentList.add(s3);          studentList.add(s4);          studentList.add(s5);            return studentList;      }  } |

**Output:**

[Student [name=Andy, age=17], Student [name=Harshal, age=18], Student [name=Peter, age=19], Student [name=Mary, age=20], Student [name=Peter, age=21]]

You can also use Comparator.comparing(Function&lt;? super T, ? extends U&gt; keyExtractor) to sort it on the basis of age.  
Comparator.comparing() accepts a [function](https://java2blog.com/java-8-function-example/) that maps a sort key from a type, and returns a Comparator that compares by that sort key.  
Let’s say you want to sort list of Students on the basis of age. You can extracts sort key age from Student object and Comparator.comparing() will return a comparator which will sort by that sort key.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Function<Student,Integer> fun = (s) -> s.getAge();      List<Student> studentsListByAge = studentsList.stream()                                                 .sorted(Comparator.comparing(fun))                                                 .collect(Collectors.toList()); |

Here Comparator.comparing() returns a new Comparator to sort based on age.

We can also use [method reference](https://java2blog.com/java-8-method-reference/) here as we are just calling s.getAge() in the [functional interface](https://java2blog.com/java-8-functional-interface-example/).

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Function<Student,Integer> fun = Student::getAge      List<Student> studentsListByAge = studentsList.stream()                                              .sorted(Comparator.comparing(fun))                                                  .collect(Collectors.toList()); |

Let’s say you want to sort list of Students on the basis of age in descending order. You can pass another comparator to Comparator.comparing() to make custom sorting based on sort key.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<Student> sortedListOfStudent5 = listOfStudents.stream()                                                             .sorted(Comparator.comparing(Student::getAge,(age1,age2) -> age2 - age1))                                                             .collect(Collectors.toList()); |

Here,  
Comparator.comparing() has two arguments.  
Student::getAge to define sort key.  
(age1,age2) -&gt; age2 - age1) to define custom sorting on the basis of sort key.

Here is the complete example.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.Comparator;  import java.util.List;  import java.util.function.Function;  import java.util.stream.Collectors;    public class SortListOfStudents {        public static void main(String[] args) {          List<Student> sList = getListOfStudents();            Function<Student,Integer> fun = (s) -> s.getAge();          List<Student> sListByAge = sList.stream()                                           .sorted(Comparator.comparing(fun))                                           .collect(Collectors.toList());          System.out.println("Sorted list by age ascending: "+sListByAge);            List<Student> sListByAgeRev = sList.stream()                                             .sorted(Comparator.comparing(Student::getAge                                                                           ,(age1,age2) -> age2 - age1))                                             .collect(Collectors.toList());            System.out.println("Sorted list by age descending: "+sListByAgeRev);      }        public static List<Student> getListOfStudents()      {          List<Student> studentList=new ArrayList<>();            Student s1=new Student("Peter",21);          Student s2=new Student("Harshal",18);          Student s3=new Student("Andy",17);          Student s4=new Student("Mary",20);          Student s5=new Student("Peter",19);            studentList.add(s1);          studentList.add(s2);          studentList.add(s3);          studentList.add(s4);          studentList.add(s5);            return studentList;      }  } |

**Output:**

Sorted list by age ascending: [Student [name=Andy, age=17], Student [name=Harshal, age=18], Student [name=Peter, age=19], Student [name=Mary, age=20], Student [name=Peter, age=21]] Sorted list by age descending: [Student [name=Peter, age=21], Student [name=Mary, age=20], Student [name=Peter, age=19], Student [name=Harshal, age=18], Student [name=Andy, age=17]]

### Sort list of Students by name and age

Let’s say you want to list of students by name and if name is same, then you need to sort by age.

You can use Comparator.thenComparing() with Comparator.comparing() to achieve the same.

Let’s see with the help of example.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.Comparator;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfStudents {        public static void main(String[] args) {          List<Student> sList = getListOfStudents();            List<Student> sListByNameAge = sList.stream()                                             .sorted(Comparator.comparing(Student::getName)                                                               .thenComparing(Student::getAge))                                             .collect(Collectors.toList());            System.out.println(sListByNameAge);      }        public static List<Student> getListOfStudents()      {          List<Student> studentList=new ArrayList<>();            Student s1=new Student("Peter",21);          Student s2=new Student("Harshal",18);          Student s3=new Student("Andy",17);          Student s4=new Student("Mary",20);          Student s5=new Student("Peter",19);            studentList.add(s1);          studentList.add(s2);          studentList.add(s3);          studentList.add(s4);          studentList.add(s5);            return studentList;      }  } |

**Output:**

[Student [name=Andy, age=17], Student [name=Harshal, age=18], Student [name=Mary, age=20], Student [name=Peter, age=19], Student [name=Peter, age=21]]

As you can see, there are two students named Peter in the list then, sorted by age.

## Excercise

Given a list of Employee objects, you need to sort them of Employee’s name in descending order and return a sorted list of Employee(List<Employee>)  
Here is the definition of Employee class.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package org.arpit.java2blog;    public class Employee {        String name;        public Employee(String name) {          super();          this.name = name;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }    *@Override*      public String toString() {          return "Employee [name=" + name + "]";      }  } |

[Hide Answers](https://java2blog.com/java-stream-sorted/)

We can use pass comparator to sorted() method or Comparator.comparing() to sort list of employee by name in descending order.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.Comparator;  import java.util.List;  import java.util.stream.Collectors;    public class SortListOfEmployees {    public static void main(String[] args) {  List<Employee> eList=getListOfEmployees();    List<Employee> eListDesc1 = eList.stream()                          .sorted((e1,e2) -> e2.getName().compareTo(e1.getName()))                                  .collect(Collectors.toList());    System.out.println(eListDesc1);    List<Employee> eListDesc2 = eList.stream().sorted(Comparator.comparing(Employee::getName  ,(s1,s2)->s2.compareTo(s1)))                                                .collect(Collectors.toList());  System.out.println(eListDesc2);    }  private static List<Employee> getListOfEmployees() {    List<Employee> listOfEmployees=new ArrayList<>();    Employee e1=new Employee("Shahid");  Employee e2=new Employee("Anchit");  Employee e3=new Employee("Chetan");  Employee e4=new Employee("Vikrant");    listOfEmployees.add(e1);  listOfEmployees.add(e2);  listOfEmployees.add(e3);  listOfEmployees.add(e4);    return listOfEmployees;  }    } |

**Output:**

[Employee [name=Vikrant], Employee [name=Shahid], Employee [name=Chetan], Employee [name=Anchit]] [Employee [name=Vikrant], Employee [name=Shahid], Employee [name=Chetan], Employee [name=Anchit]]

# Java 8 Collectors examples

In this post,  we are going to see java 8 Collectors examples. You can do various operations such as average, count, groupby, sort the list with the help of Collectors. I am not providing theory here, I think you will learn better with the help of examples.  
**Examples:**

#### Counting:

Counting is used to count number of elements in the stream.It returns Collector instance which can be accepted by collect method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package org.arpit.java2blog;  import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8CollectorsExample {        public static void main(String[] args) {          List intList=Arrays.asList(10,20,30,40,50);          // Counting          long  count = intList.stream().collect(Collectors.counting());          System.out.println(count);        }  } |

When you run above code, you will get below output:

**💻 Awesome Tech Resources:**

* Looking for ⚒️ tech jobs? Go to our [job portal](https://lcsjobs.io/).
* Looking for tech events? Go to [tech events](https://aigents.co/opportunities/events) 🗓️ Calendar.️

|  |  |
| --- | --- |
| 1  2  3 | 5 |

#### AveragingInt :

AveragingInt is used to find average of stream elements as int datatype. It returns Collector instance which can be accepted by collect method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package org.arpit.java2blog;    import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8CollectorsExample {        public static void main(String[] args) {          List intList=Arrays.asList(10,20,30,40,50);          // Averaging int          Double result1 = intList.stream().collect(Collectors.averagingInt(v->v));          System.out.println(result1);            Double result2 = intList.stream().collect(Collectors.averagingInt(v->v\*v));          System.out.println(result2);      }  } |

When you run above code, you will get below output:

30.0  
1100.0

Lets understand how did you get 30 for case 1 :  
**(10+20+30+40+50/5)= 150/5 =30.0**  
Now you must wondering how did we get 1100 for 2nd case:  
**(10\*10 + 20\*20 + 30\*30 + 40\*40 + 50\*50)/5=5500/5 = 1100.0**  
If you want to understand more about **v-> v\*v** , you can go through Java 8 lambda expressions Similarly we have different function for different data types such as AveragingDouble, AveragingLong.

#### joining

Joining method is used to concatenate with delimiter, you can also pass prefix and suffix.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package org.arpit.java2blog;  import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8CollectorsExample {        public static void main(String[] args) {          List NameList=Arrays.asList("Arpit","John","Martin");          // Counting          String  stringWithHyphen = NameList.stream().collect(Collectors.joining("-"));          System.out.println("String with hyphen : "+stringWithHyphen);          String  stringWithHyphenAndPrefixAndSuffix = NameList.stream().collect(Collectors.joining("-","==","=="));          System.out.println("String with hyphen , suffix and prefix :  "+stringWithHyphenAndPrefixAndSuffix);      }  } |

When you run above code, you will get below output:

String with hyphen : Arpit-John-Martin  
String with hyphen , suffix and prefix : ==Arpit-John-Martin==

#### summingint:

summingInt is used to find sum of stream elements as int datatype. It returns Collector instance which can be accepted by collect method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package org.arpit.java2blog;    import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8CollectorsExample {        public static void main(String[] args) {          List intList=Arrays.asList(10,20,30,40,50);          // Averaging int          Double result1 = intList.stream().collect(Collectors.summingInt(v->v));          System.out.println(result1);            Double result2 = intList.stream().collect(Collectors.summingInt(v->v\*v));          System.out.println(result2);      }  } |

When you run above code, you will get below output:

150  
5500

Similarly we have different function for different data types such as summingDouble, summingLong.

#### collectingAndThen:

collectingAndThen: is used to get a Collector instance and perform finishing function on top of it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package org.arpit.java2blog;    import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8CollectorsExample {        public static void main(String[] args) {          List intList=Arrays.asList(10,20,30,40,50);          // collectingAndThen          int result1 = intList.stream().collect(Collectors.collectingAndThen(Collectors.summingInt(v->(int)v),result->result/2));          System.out.println(result1);      }  } |

When you run above code, you will get below output:

75

# Collectors.groupby example : How to do group by in java

We have already seen some examples on Collectors in previous post.  In this post, we are going to see Java 8 Collectors groupby example. Groupby is another feature added in java 8 and it is very much similar to SQL/Oracle.  
Lets understand more with the help of example: Lets create our model class country as below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | package org.arpit.java2blog;    public class Country{    String countryName;  long population;    public Country() {    super();  }  public Country(String countryName,long population) {    super();    this.countryName = countryName;    this.population=population;  }  public String getCountryName() {    return countryName;  }  public void setCountryName(String countryName) {    this.countryName = countryName;  }  public long getPopulation() {    return population;  }  public void setPopulation(long population) {    this.population = population;  }  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result     + ((countryName == null) ? 0 : countryName.hashCode());  result = prime \* result + (int) (population ^ (population >>> 32));  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)    return true;  if (obj == null)    return false;  if (getClass() != obj.getClass())    return false;  Country other = (Country) obj;  if (countryName == null) {    if (other.countryName != null)     return false;  } else if (!countryName.equals(other.countryName))    return false;  if (population != other.population)    return false;  return true;  }    public String toString()  {    return "{"+countryName+","+population+"}";  }  } |

Lets create main class in which we will use Collectors.groupBy to do group by.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | package org.arpit.java2blog;    import java.math.BigDecimal;  import java.util.Arrays;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class Java8CollectorsGroupBy {  public static void main(String[] args) {      List items = Arrays.asList(new Country("India", 20000),      new Country("China", 40000), new Country("Nepal", 30000),      new Country("India", 50000), new Country("China", 10000));      // Group by countryName    Map<String, List> groupByCountry = items.stream().collect(      Collectors.groupingBy(Country::getCountryName));      System.out.println(groupByCountry.get("India"));      // Group by CountryName and calculates the count    Map<String, Long> counting = items.stream().collect(      Collectors.groupingBy(Country::getCountryName,Collectors.counting()));      // Group by countryName and sum up the population    System.out.println(counting);    Map<String, Long> populationCount = items.stream().collect(      Collectors.groupingBy(Country::getCountryName,        Collectors.summingLong(Country::getPopulation)));    System.out.println(populationCount);  }  } |

When you run above class, you will get below output:

[{India,20000}, {India,50000}] {China=2, Nepal=1, India=2}  
{China=50000, Nepal=30000, India=70000}

**Reference:**  
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

# Java Stream List to Map

In this post, we will see how to convert List to Map using Stream in java 8.

Collectors’s toMap() can be used with Stream to convert List to Map in java.

Consider a class Named Movie which have 3 fields – id, name and genre

**💻 Awesome Tech Resources:**

* Looking for ⚒️ tech jobs? Go to our [job portal](https://lcsjobs.io/).
* Looking for tech events? Go to [tech events](https://aigents.co/opportunities/events) 🗓️ Calendar.️

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package org.arpit.java2blog;    public class Movie {        private String name;      private String genre;        public Movie(String name, String genre) {          super();          this.name = name;          this.genre = genre;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public String getGenre() {          return genre;      }        public void setGenre(String genre) {          this.genre = genre;      }    *@Override*      public String toString() {          return "Movie [name=" + name + ", genre=" + genre + "]";      }  } |

Create a list of movies and convert with to map with name as key and genre as value.

Create a class Named ConvertListToMapMain

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class ConvertListToMapMain {        public static void main(String[] args) {          List<Movie> moviesList=getListOfMovies();            Map<String, String> moviesMap = moviesList.stream()                    .collect(Collectors.toMap((m)->m.getName(), (m)->m.getGenre()));          System.out.println(moviesMap);      }        public static List<Movie> getListOfMovies()      {          List<Movie> moviesList=new ArrayList<>();            Movie m1=new Movie("3 idiots","Comedy");          Movie m2=new Movie("Interstellar","SciFi");          Movie m3=new Movie("Forest gump","Comedy");          Movie m4=new Movie("Matrix","SciFi");          Movie m5=new Movie("The Hangover","Comedy");            moviesList.add(m1);          moviesList.add(m2);          moviesList.add(m3);          moviesList.add(m4);          moviesList.add(m5);            return moviesList;      }  } |

**Output:**

{Interstellar=SciFi, Matrix=SciFi, The Hangover=Comedy, Forest gump=Comedy, 3 idiots=Comedy}

moviesMap contains name as key and genre as value.  
We can also use [method references](https://java2blog.com/java-8-method-reference/) instead of lamda expressions.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4 | Map<String, String> moviesMap = moviesList.stream()                    .collect(Collectors.toMap(Movie::getName, Movie::getGenre())); |

### **What happens in case of a duplicate key?**

In case, duplicate keys are present, then it will throw java.lang.IllegalStateException exception.

Let’s say we have two movies with name Matrix

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public static List<Movie> getListOfMovies()      {          List<Movie> moviesList=new ArrayList<>();            Movie m1=new Movie("3 idiots","Comedy");          Movie m2=new Movie("Interstellar","SciFi");          Movie m3=new Movie("Forest gump","Comedy");          Movie m4=new Movie("Matrix","SciFi");          Movie m5=new Movie("Matrix","Comedy");            moviesList.add(m1);          moviesList.add(m2);          moviesList.add(m3);          moviesList.add(m4);          moviesList.add(m5);            return moviesList;      } |

When you run ConvertListToMapMain again, you will get below output:  
**Output:**

Exception in thread “main” java.lang.IllegalStateException: Duplicate key Sci Fi  
at java.util.stream.Collectors.lambda$throwingMerger$0(Collectors.java:133)  
at java.util.HashMap.merge(HashMap.java:1254)  
at java.util.stream.Collectors.lambda$toMap$58(Collectors.java:1320)  
at java.util.stream.ReduceOps$3ReducingSink.accept(ReduceOps.java:169)  
at java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1382)  
at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:482)  
at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:472)  
at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)  
at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)  
at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:566)  
at org.arpit.java2blog.ConvertListToMapMain.main(ConvertListToMapMain.java:14)

To resolve this issue, we need to pass merge BinaryOperator function to Collectors.toMap().

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Map<String, String> moviesMap = moviesList.stream()          .collect(Collectors.toMap((m)->m.getName(), (m)->m.getGenre(),                  (oldValue,newValue) -> newValue          )); |

When you run ConvertListToMapMain again, you will get below output:  
**Output:**

{Matrix=Comedy, Interstellar=SciFi, Forest gump=Comedy, 3 idiots=Comedy}

### **What if you want specific Map such as**[TreeMap](https://java2blog.com/treemap-in-java-with-examples/)**?**

You can pass [constructor method reference](https://java2blog.com/java-8-method-reference/) to Collectors.toMap() to get specific map.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Map<String, String> moviesMap = moviesList.stream()                    .collect(Collectors.toMap( (m)->m.getName(), (m)->m.getGenre(),                            (oldValue,newValue) -> newValue                            ,TreeMap::new                            )); |

When you run ConvertListToMapMain again, you will get below output:  
**Output:**

{3 idiots=Comedy, Forest gump=Comedy, Interstellar=SciFi, Matrix=Comedy}

As you can see, we have TreeMap which is sorted by name.

## Get multiMap with Collectors.groupingBy()

In case, you want movie grouped by genre as key and name as value, then you can use [Collectors.groupingBy()](https://java2blog.com/how-to-do-groupby-in-java/) and Collectors.mapping as below:

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class ConvertListToMapMain {        public static void main(String[] args) {          List<Movie> moviesList=getListOfMovies();            Map<String, List<String>> moviesGenMap = moviesList.stream()                    .collect(Collectors.groupingBy(Movie::getGenre                                                  ,Collectors.mapping(Movie::getName, Collectors.toList())));          System.out.println(moviesGenMap);      }        public static List<Movie> getListOfMovies()      {          List<Movie> moviesList=new ArrayList<>();            Movie m1=new Movie("3 idiots","Comedy");          Movie m2=new Movie("Interstellar","SciFi");          Movie m3=new Movie("Forest gump","Comedy");          Movie m4=new Movie("Matrix","SciFi");          Movie m5=new Movie("The Hangover","Comedy");            moviesList.add(m1);          moviesList.add(m2);          moviesList.add(m3);          moviesList.add(m4);          moviesList.add(m5);            return moviesList;      }    } |

**Output:**

{Comedy=[3 idiots, Forest gump, The Hangover], SciFi=[Interstellar, Matrix]}

As you can see, moviesGenMap has genre as key and list of movies name as value.

## Excercise

Given a list of Employee objects, you need to convert list to Map with name as key and age as value. In case, there are duplicate keys, then you need to preserve old value.  
Here is the definition of Employee class.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | package org.arpit.java2blog;    public class Employee {        String name;      int age;        public Employee(String name,int age) {          super();          this.name = name;          this.age=age;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public int getAge() {          return age;      }        public void setAge(int age) {          this.age = age;      }  } |

[Hide Answers](https://java2blog.com/java-stream-list-to-map/)

We can use Collectors.toMap() to convert list of employees to Map. As we need to preserve old values in case of duplicate key, we have used merge function as (oldValue,newValue)->oldValue

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.Map;  import java.util.stream.Collectors;    public class ExcerciseListToMap {    public static void main(String[] args) {  List<Employee> eList=getListOfEmployees();    Map<String, Integer> mapNameToAge = eList.stream()                                  .collect(Collectors.toMap(Employee::getName                                                     ,Employee::getAge                                                     ,(oldValue,newValue)->oldValue                                   ));  System.out.println(mapNameToAge);    }  private static List<Employee> getListOfEmployees() {    List<Employee> listOfEmployees=new ArrayList<>();    Employee e1=new Employee("Amit",24);  Employee e2=new Employee("Anchit",27);  Employee e3=new Employee("Vaibhav",32);  Employee e4=new Employee("Amit",22);    listOfEmployees.add(e1);  listOfEmployees.add(e2);  listOfEmployees.add(e3);  listOfEmployees.add(e4);    return listOfEmployees;  }    } |

**Output:**

{Amit=24, Vaibhav=32, Anchit=27}

# Difference between map and flatMap in java

In this post, we will see the difference between map and flatMap in java.I have already covered Stream’s map and [flatMap](https://www.java2blog.com/java-8-stream-flatmap/) methods  in previous articles.

As you might know, Stream’s map and flatMap method both can be applied on Stream<T> and return Stream<R> as output.

The actual difference is, map operation produces one output value for one input value but flatMap operation produces zero or more number of values for each input value.  
Let’s understand with the help of the simple example.

**💻 Awesome Tech Resources:**

* Looking for ⚒️ tech jobs? Go to our [job portal](https://lcsjobs.io/).
* Looking for tech events? Go to [tech events](https://aigents.co/opportunities/events) 🗓️ Calendar.️

## Stream map vs flatMap

You have list of String and you want to make them uppercase, in this case, you can simply use map function as below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package org.arpit.java2blog.map;    import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8MapExample {      public static void main(String[] str)      {          List<String> listOfCountries=Arrays.asList("India","China","Nepal","Bhutan");            List<String> listOfCountriesUppercase=listOfCountries.stream()                                                .map(String::toUpperCase)                                                .collect(Collectors.toList());          listOfCountriesUppercase.forEach(System.out::println);      }  } |

When you run above program, you will get below output:

INDIA

CHINA

NEPAL

BHUTAN

map works well in case of simple cases as above.

#### What if you have more complex data structure such as List of List of String(List<List<String>>)

Let’s say you have a list of list of cities(List<List<String>>) and you want to find all the cites which start with "T".  
You can not directly use filter on List<List<String>>, it’s not going to work.  
You need to use flatmap to flatten the List and then use the filter on it to get the results.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package org.arpit.java2blog.flatMap;    import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;    public class Java8FlatMapExample {      public static void main(String[] str)      {          List<List<String>> listofListOfCities=Arrays.asList(Arrays.asList("Delhi","Mumbai"),                  Arrays.asList("Beijing","Shanghai","Tianjin"),                  Arrays.asList("Kathmandu","Lalitpur"),                  Arrays.asList("Thimphu","Phuntsholing"));            List<String> listOfCitiesUppercase=listofListOfCities.stream()                                                .flatMap(citiesByCountries -> citiesByCountries.stream())                                                .filter(s -> s.startsWith("T"))                                                .collect(Collectors.toList());          listOfCitiesUppercase.forEach(System.out::println);      }  } |

When you run above program, you will get below output:

Tianjin  
Thimphu

## Java Parallel Stream introduction

[Java 8](https://java2blog.com/java-8-tutorial/) introduces the concept of parallel stream to do parallel processing. As we have more number of cpu cores nowadays due to cheap hardware costs, parallel processing can be used to perform operation faster.

Let’s understand with help of simple example

**💻 Awesome Tech Resources:**

* Looking for ⚒️ tech jobs? Go to our [job portal](https://lcsjobs.io/).
* Looking for tech events? Go to [tech events](https://aigents.co/opportunities/events) 🗓️ Calendar.️

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package org.arpit.java2blog.java8;    import java.util.Arrays;  import java.util.stream.IntStream;    public class Java8ParallelStreamMain {        public static void main(String[] args) {            System.out.println("=================================");          System.out.println("Using Sequential Stream");          System.out.println("=================================");          int[] array= {1,2,3,4,5,6,7,8,9,10};          IntStream intArrStream=Arrays.stream(array);          intArrStream.forEach(s->          {              System.out.println(s+" "+Thread.currentThread().getName());          }                  );            System.out.println("=================================");          System.out.println("Using Parallel Stream");          System.out.println("=================================");          IntStream intParallelStream=Arrays.stream(array).parallel();          intParallelStream.forEach(s->          {              System.out.println(s+" "+Thread.currentThread().getName());          }                  );      }  } |

When you run above program, you will get below output

=================================  
Using Sequential Stream  
=================================  
1 main  
2 main  
3 main  
4 main  
5 main  
6 main  
7 main  
8 main  
9 main  
10 main  
=================================  
Using Parallel Stream  
=================================  
7 main  
6 ForkJoinPool.commonPool-worker-3  
3 ForkJoinPool.commonPool-worker-1  
9 ForkJoinPool.commonPool-worker-2  
2 ForkJoinPool.commonPool-worker-3  
5 ForkJoinPool.commonPool-worker-1  
10 ForkJoinPool.commonPool-worker-2  
1 ForkJoinPool.commonPool-worker-3  
8 ForkJoinPool.commonPool-worker-2  
4 ForkJoinPool.commonPool-worker-1

If you notice the output,main thread is doing all the work in case of sequential stream. It waits for current iteration to complete and then work on next iteration.  
In case of Parallel stream,4 threads are spawned simultaneously and it internally using Fork and Join pool to create and manage threads.Parallel streams create ForkJoinPool instance via static ForkJoinPool.commonPool() method.

Parallel Stream takes the benefits of all available CPU cores and processes the tasks in parallel. If number of tasks exceeds the number of cores, then remaining tasks wait for currently running task to complete.

## Parallel Streams are cool, so should you use it always?

**A big No!!**  
It is easy to convert sequential [Stream](https://java2blog.com/java-8-stream-filter-examples/) to parallel Stream just by adding .parallel, does not mean you should always use it.  
There are lots of factors you need to consider while using parallel streams otherwise you will suffer from negative impacts of parallel Streams.

Parallel Stream has much higher overhead than sequential Stream and it takes good amount of time to coordinate between threads.  
You need to consider parallel Stream if and only if:

* You have large dataset to process.
* As you know that Java uses ForkJoinPool to achieve parallelism, ForkJoinPool forks sources stream and submit for execution, so your source stream should be splittable.  
  For example:  
  [ArrayList](https://java2blog.com/arraylist-in-java-with-example/) is very easy to split, as we can find a middle element by its index and split it but LinkedList is very hard to split and does not perform very well in most of the cases.
* You are actually suffering from performance issues.
* You need to make sure that all the shared resources between threads need to be synchronized properly otherwise it might produce unexpected results.

Simplest formula for measuring parallelism is "NQ" model as provided by **Brian Goetz** in his presentation.

**NQ Model**:

N x Q > 10000

where,  
N = number of items in dataset  
Q = amount of work per item

It means if you have a large number of datasets and less work per item(For example: Sum), parallelism might help you run program faster and vice versa is also true. So if you have less number of datasets and more work per item(doing some computational work), then also parallelism might help you in achieving results faster.

Let’s see with the help of another example.

In this example, we are going to see how CPU behaves when you perform long computations in case of parallel Stream and sequential stream.We are doing some arbit calculations to make CPU busy.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package org.arpit.java2blog.java8;  import java.util.ArrayList;  import java.util.List;    public class PerformanceComparisonMain {        public static void main(String[] args) {            long currentTime=System.currentTimeMillis();          List<Integer> data=new ArrayList<Integer>();          for (int i = 0; i < 100000; i++) {              data.add(i);          }            long sum=data.stream()                  .map(i ->(int)Math.sqrt(i))                  .map(number->performComputation(number))                  .reduce(0,Integer::sum);            System.out.println(sum);          long endTime=System.currentTimeMillis();          System.out.println("Time taken to complete:"+(endTime-currentTime)/(1000\*60)+" minutes");        }        public static int performComputation(int number)      {          int sum=0;          for (int i = 1; i < 1000000; i++) {              int div=(number/i);              sum+=div;            }          return sum;      }  } |

When you run above program, you will get below output.

117612733  
Time taken to complete:6 minutes

But we are not interested in output here, but how CPU behaved when above operation performed.  


As you can see CPU is not fully utilized in case of Sequential Stream.

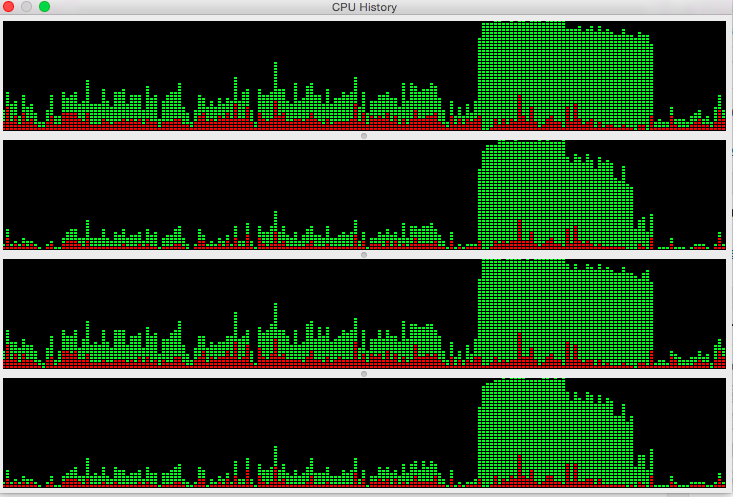
Let’s change at 16 line no. and make the stream parallel and run the program again.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long sum=data.stream()                  .parallel()                  .map(i ->(int)Math.sqrt(i))                  .map(number->performComputation(number))                  .reduce(0,Integer::sum); |

You will get below output when you run Stream in parallel.

117612733  
Time taken to complete:3 minutes

Let’s check CPU history when we ran program using parallel stream.



As you can see parallel stream used all 4 CPU cores to perform computation.

## Custom Thread pool in Parallel Stream

The parallel stream by default uses ForkJoinPool.commonPool which has one less thread than number of processor. This means parallel stream uses all available processors because it uses the main thread as well.

In case, you are using multiple parallel streams, then they will share same ForkJoinPool.commonPool .This means you may not be able to use all the processors assigned to each parallel stream.

To solve this issue, you can create own thread pool while processing the stream.

**Java**

|  |  |
| --- | --- |
| 1  2  3 | ForkJoinPool fjp = new ForkJoinPool(parallelism); |

This will create ForkJoinPool with target parallelism level. If you don’t pass parallelism, it will equal to the number of processors by default.

Now you can submit parallel stream to this custom ForkJoinPool.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ForkJoinPool fjp1 = new ForkJoinPool(5);  Callable<Integer> callable1 = () -> data.parallelStream()                     .map(i -> (int) Math.sqrt(i))                     .map(number -> performComputation(number))                     .peek( (i) -> {                        System.out.println("Processing with "+Thread.currentThread()+" "+ i);                        })                      .reduce(0, Integer::sum);            try {              sumFJ1 = fjp1.submit(callable1).get();          } catch (InterruptedException | ExecutionException e) {              e.printStackTrace();          } |

Let’s understand with the help of example.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.Callable;  import java.util.concurrent.ExecutionException;  import java.util.concurrent.ForkJoinPool;    public class PerformanceComparisonMain {        public static void main(String[] args) {            List<Integer> data = new ArrayList<Integer>();          for (int i = 0; i < 10; i++) {              data.add(i);          }            System.out.println("================");          System.out.println("Parallel stream 1");          System.out.println("================");          long sum1 =data.parallelStream()          .map(i -> (int) Math.sqrt(i))          .map(number -> performComputation(number))          .peek( (i) -> {             System.out.println("Processing with "+Thread.currentThread()+" "+ i);               })          .reduce(0, Integer::sum);            System.out.println("Sum: "+sum1);            System.out.println("================");          System.out.println("Parallel stream 2");          System.out.println("================");            long sum2 = data.parallelStream()                  .map(i -> ((int) Math.sqrt(i)\*10))                  .map(number -> performComputation(number))                  .peek( (i) -> {                     System.out.println("Processing with "+Thread.currentThread()+" "+ i);                       })                  .reduce(0, Integer::sum);            System.out.println("Sum: "+sum2);            System.out.println("================");          System.out.println("Parallel stream with custom thread pool 1");          System.out.println("================");            ForkJoinPool fjp1 = new ForkJoinPool(5);          long sumFJ1 = 0;            Callable<Integer> callable1 = () -> data.parallelStream()                     .map(i -> (int) Math.sqrt(i))                     .map(number -> performComputation(number))                     .peek( (i) -> {                        System.out.println("Processing with "+Thread.currentThread()+" "+ i);                        })                      .reduce(0, Integer::sum);            try {              sumFJ1 = fjp1.submit(callable1).get();          } catch (InterruptedException | ExecutionException e) {              e.printStackTrace();          }            System.out.println("Sum: "+sumFJ1);            System.out.println("================");          System.out.println("Parallel stream with custom thread pool 2");          System.out.println("================");            Callable<Integer> callable2 = () -> data.parallelStream()                  .map(i -> (int) Math.sqrt(i)\*10)                  .map(number -> performComputation(number))                  .peek( (i) -> {                     System.out.println("Processing with "+Thread.currentThread()+" "+ i);                       })                  .reduce(0, Integer::sum);            long sumFJ2 = 0;            ForkJoinPool fjp2 = new ForkJoinPool(4);            try {              sumFJ2 = fjp2.submit(callable2).get();          } catch (InterruptedException | ExecutionException e) {              e.printStackTrace();          }            System.out.println("Sum: "+sumFJ2);      }        public static int performComputation(int number) {          int sum = 0;          for (int i = 1; i < 100000; i++) {              int div = (number / i);              sum += div;            }          return sum;      }  } |

When you run the program, you will get below output:

================  
Parallel stream 1  
================  
Processing with Thread[ForkJoinPool.commonPool-worker-2,5,main] 3  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 1  
Processing with Thread[ForkJoinPool.commonPool-worker-2,5,main] 5  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 1  
Processing with Thread[ForkJoinPool.commonPool-worker-2,5,main] 3  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 3  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 0  
Processing with Thread[ForkJoinPool.commonPool-worker-2,5,main] 1  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 3  
Processing with Thread[main,5,main] 3  
Sum: 23  
================  
Parallel stream 2  
================  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 66  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 111  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 66  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 27  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 27  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 66  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 0  
Processing with Thread[ForkJoinPool.commonPool-worker-1,5,main] 27  
Processing with Thread[ForkJoinPool.commonPool-worker-3,5,main] 66  
Processing with Thread[main,5,main] 66  
Sum: 522  
================  
Parallel stream with custom thread pool 1  
================  
Processing with Thread[ForkJoinPool-1-worker-1,5,main] 3  
Processing with Thread[ForkJoinPool-1-worker-3,5,main] 3  
Processing with Thread[ForkJoinPool-1-worker-4,5,main] 1  
Processing with Thread[ForkJoinPool-1-worker-3,5,main] 5  
Processing with Thread[ForkJoinPool-1-worker-4,5,main] 0  
Processing with Thread[ForkJoinPool-1-worker-1,5,main] 3  
Processing with Thread[ForkJoinPool-1-worker-3,5,main] 3  
Processing with Thread[ForkJoinPool-1-worker-2,5,main] 1  
Processing with Thread[ForkJoinPool-1-worker-3,5,main] 3  
Processing with Thread[ForkJoinPool-1-worker-2,5,main] 1  
Sum: 23  
================  
Parallel stream with custom thread pool 2  
================  
Processing with Thread[ForkJoinPool-2-worker-1,5,main] 66  
Processing with Thread[ForkJoinPool-2-worker-3,5,main] 66  
Processing with Thread[ForkJoinPool-2-worker-1,5,main] 66  
Processing with Thread[ForkJoinPool-2-worker-3,5,main] 111  
Processing with Thread[ForkJoinPool-2-worker-0,5,main] 66  
Processing with Thread[ForkJoinPool-2-worker-1,5,main] 27  
Processing with Thread[ForkJoinPool-2-worker-3,5,main] 0  
Processing with Thread[ForkJoinPool-2-worker-2,5,main] 27  
Processing with Thread[ForkJoinPool-2-worker-0,5,main] 66  
Processing with Thread[ForkJoinPool-2-worker-1,5,main] 27  
Sum: 522

As you can see, first two parallel streams are using ForkJoinPool.commonPool and next 2 are using custom thread pools i.e. ForkJoinPool-1 and ForkJoinPool-2

## Things you should keep in mind while using Parallel Stream

### Stateful lambda expressions

You should avoid using stateful lambda expressions in stream operations.A Stateful lambda expressions is one whose output depends on any state that might change during execution of stream operations.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.Arrays;  import java.util.Collections;  import java.util.List;    public class ListOfIntegersStatefulLambda {        public static void main(String[] args) {            List<Integer> listOfIntegers = Arrays.asList(new Integer[] {40,34,21,37,20});          List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());          listOfIntegers.parallelStream()                    // You shou! It uses a stateful lambda expression.                  .map(e -> {                      syncList.add(e);                      return e;                  })          .forEachOrdered(e -> System.out.print(e + " "));            System.out.println("");            syncList.stream().forEachOrdered(e -> System.out.print(e + " "));          System.out.println("");      }  } |

**Output:**

40 34 21 37 20  
40 34 37 20 21

forEachOrdered processes the elements in order imposed bt stream. .map(e -> {syncList.add(e); return e;}) is stateful lambda and order in which .map(e -> {syncList.add(e); return e;}) adds element to  
syncList can vary, so you should not use stateful lambda operations while using parallel stream.

### Interference

Lambda expression in stream operation should not modify source of stream.  
Following code tries to add element to list of integer and throw concurrentModification exception.

**Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package org.arpit.java2blog;    import java.util.ArrayList;  import java.util.List;    public class ListOfIntegersStatefulLambda {        public static void main(String[] args) {            List<Integer> listOfIntegers = new ArrayList<>();          Integer[] intArray =new Integer[] {40,34,21,37,20};          for(Integer in:intArray)          {              listOfIntegers.add(in);          }          listOfIntegers.parallelStream()            .peek( i -> listOfIntegers.add(7))          .forEach(e -> System.out.print(e + " "));            System.out.println("");        }  } |

**Output:**

34 21 40 20 37 Exception in thread “main” java.util.ConcurrentModificationException: java.util.ConcurrentModificationException  
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)  
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)  
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)  
at java.lang.reflect.Constructor.newInstance(Constructor.java:423)  
at java.util.concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:593)  
at java.util.concurrent.ForkJoinTask.reportException(ForkJoinTask.java:677)  
at java.util.concurrent.ForkJoinTask.invoke(ForkJoinTask.java:735)  
at java.util.stream.ForEachOps$ForEachOp.evaluateParallel(ForEachOps.java:159)  
at java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateParallel(ForEachOps.java:173)  
at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:233)  
at java.util.stream.ReferencePipeline.forEach(ReferencePipeline.java:485)  
at org.arpit.java2blog.ListOfIntegersStatefulLambda.main(ListOfIntegersStatefulLambda.java:19)  
Caused by: java.util.ConcurrentModificationException  
at java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1388)  
at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:482)  
at java.util.stream.ForEachOps$ForEachTask.compute(ForEachOps.java:290)  
at java.util.concurrent.CountedCompleter.exec(CountedCompleter.java:731)  
at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)  
at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1056)  
at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)  
at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)

Please note that all the intermediate operations are lazy, execution of streams begins when foreach is invoked. As argument of peek tries to modifies stream source during execution of stream, which causes Java to throw ConcurrentModificationException

## Conclusion

You have learnt about parallel streams when to use parallel streams with examples. You should be careful while using parallel streams. Parallel streams are very powerful if used in the correct context.