**Java 8 tutorial :**

you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations , Streams, functional interfaces, map extensions and the new Date API.

**What are new features which got introduced in Java 8?**

There are lots of new features which added in Java 8. Here is the list of important features:

> Lambda Expression

> Stream API

> Default methods in the interface

> Functional Interface

> Optional

> Method references

> Date API

> Nashorn, JavaScript Engine

**What are main advantages of using Java 8?**

> More compact code

> Less boiler plate code

> More readable and reusable code

> More testable code

> Parallel operations

**Lambda Expressions in Java 8**

Java 8 has introduced a new feature called Lambda expressions. As this change will bring functional programming into Java. Other languages such as Scala already have this feature so this is not new to programming world, it is new to java.

Before understanding Lambda expressions, Lets first understand **Functional Interface.**

**What is Functional Interface?**

Functional interfaces are those interfaces that have only one abstract method in it. It can have more than one default or static method and can override the method from java.lang.object.

Let’s create a functional interface:

**@FunctionalInterface**

public interface Decorable {

// **one abstract method**

void decorateWithCurtains();

// **default method**

default void decorateWithPaints() {

System.out.println("Decorating using paints");

}

// **Overriding method of java.lang.Object**

@Override

public int hashCode();

}

Java can itself identify **Functional Interface** but you can also denote interface as Functional Interface by annotating it with @FunctionalInterface.

**Some popular Functional Interfaces are:**

> java.lang.Runnable

> java.util.concurrent.Callable

> java.awt.event.ActionListener

> java.util.Comparator

**Lambda expression** represents an anonymous function. It comprises of a set of parameters, a lambda operator (->) and a function body . You can call it function without name,

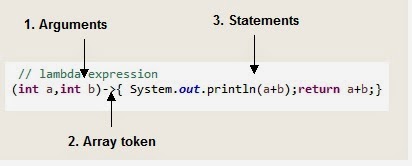
The connection between Lambda Expression and Functional Interface:

Lambda expression can be applied for the abstract method of functional Interface which is being implemented or being instantiated anonymously.

**Structure of Lambda Expressions**

(Argument List) ->{expression;} or

(Argument List) ->{statements;}



1. **Argument list or parameters**

> Lambda expression can have zero or more arguments.

()->{System.out.println(“Hello”)}; //Without argument, will print hello

(int a)->{System.out.println(a)}; // One argument, will print value of a

(int a,int b)-> {a+b};//two argument, will return sum of these two integers

> You can choose to not declare the type of arguments as it can be inferred from context.

(a,b)->{a+b}; // two argument, will return sum of these two numbers

> you can not declare one argument’s type and do not declare type for other argument.

(int a,b)->{a+b}; // Compilation error

> When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

a->{System.out.println(a)}; // Will print value of number a

1. **Array token (->)**
2. **Body**

Body can have expression or statements.

If there is only one statement in body,curly brace is not needed and return type of the anonymous function is same as of body expression

If there are more than one statements, then it should be in curly braces and return type of anonymous function is same as value return from code block, void if nothing is returned.

*// old way*

**new** Thread(**new** Runnable() {

   @Override

   public **void** run() {

    System.out.println("Thread is started");

   }

  }).start();

*// using lambda Expression*

**new** Thread(()->System.out.println("Thread is started")).start();

}

**Code to sort list of movies by name using comparator**

Old way :

List<Movie> listOfMovies = **new** ArrayList<>();

    listOfMovies.add(m1);

    listOfMovies.add(m2);

    System.out.println("Before Sort by name : ");

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

      Movie movie = (Movie) listOfMovies.get(i);

      System.out.println(movie);

    }

*// Sort by movieName*

*// Anonymous Comparator*

*// old way*

    Collections.sort(listOfMovies, **new** Comparator<Movie>() {

      @Override

      public **int** compare(Movie o1, Movie o2) {

**return** o1.getMovieName().compareTo(o2.getMovieName());

      }

    });

**Lambda way :**

The problem with Anonymous Comparator is of syntax. Each time you want to sort the list using a comparator, you have to remember the bulky syntax.

So generally the main problem with Anonymous classes is syntax. For very simple operation, we need to write complex code. To solve this problem, JDK has introduced a new feature called Lambda Expressions. how lambda expression will reduce this complex code.

*// Sort by movieName*

*// Anonymous Comparator*

*// old way*

    Collections.sort(listOfMovies, **new** Comparator<Movie>() {

      @Override

      public **int** compare(Movie o1, Movie o2) {

**return** o1.getMovieName().compareTo(o2.getMovieName());

      }

    });

*// Using lambda expression*

    Collections.sort(listOfMovies, (o1, o2) -> o1.getMovieName().compareTo(o2.getMovieName()));

    System.out.println("After Sort by name: ");

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

      Movie movie = (Movie) listOfMovies.get(i);

      System.out.println(movie);

    }

  }

for using Comparator. So in spite of writing Anonymous comparator, our expression became very easy.



So we have passed 2 arguments o1 and o2, we didn’t pass type because it can be inferred from context.

We have only one statement here, so no need to put it in curly braces.

**HelloWorld Lambda Expression Example**

public **interface** HelloWorld {

**void** sayHello();}

public **class** HelloWorldMain {

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

{

*// Lambda Expression*

     HelloWorld helloWorld=()->System.out.println("Hello using Lambda Expression");

    helloWorld.sayHello();

}

}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**Without Lambda Expression**

1. **interface** Drawable{
2. **public** **void** draw();  }
3. **public** **class** LambdaExpressionExample {
4. **public** **static** **void** main(String[] args) {
5. **int** width=10;
6. //without lambda, Drawable implementation using anonymous class
7. Drawable d=**new** Drawable(){
8. **public** **void** draw(){System.out.println("Drawing "+width);}
9. };
10. d.draw();
11. }
12. }

**With Lambda expression**

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{

**public** **void** draw();  }

1. **public** **class** LambdaExpressionExample2 {
2. **public** **static** **void** main(String[] args) {
3. **int** width=10;
4. //with lambda
5. **Drawable d2=()->{**
6. **System.out.println("Drawing "+width);**
7. **};**
8. d2.draw();
9. }  }
10. **interface** Sayable{
11. **public** String say();  }
12. **public** **class** LambdaExpressionExample3{
13. **public** **static** **void** main(String[] args) {
14. **Sayable s=()->{**
15. **return "I have nothing to say.";**
16. **};**
17. System.out.println(s.say());
18. }  }
19. **interface** Addable{
20. **int** add(**int** a,**int** b);  }
21. **public** **class** LambdaExpressionExample5{
22. **public** **static** **void** main(String[] args) {
23. // Multiple parameters in lambda expression
24. **Addable ad1=(a,b)->(a+b);**
25. System.out.println(ad1.add(10,20));
26. // Multiple parameters with data type in lambda expression
27. **Addable ad2=(int a,int b)->(a+b);**
28. System.out.println(ad2.add(100,200));
29. }
30. }
31. **interface** Addable{
32. **int** add(**int** a,**int** b);  }
33. **public** **class** LambdaExpressionExample6 {
34. **public** **static** **void** main(String[] args) {
35. // Lambda expression without return keyword.
36. **Addable ad1=(a,b)->(a+b);**
37. System.out.println(ad1.add(10,20));
38. // Lambda expression with return keyword.
39. **Addable ad2=(int a,int b)->{  return (a+b);};**
40. System.out.println(ad2.add(100,200));
41. }
42. }

**Java Lambda Expression Example: Foreach Loop**

1. list.forEach(
2. (n)->System.out.println(n)
3. );

**Multiple Statements**

**Creating Thread**

1. //Thread Example with lambda
2. Runnable r2=()->{
3. System.out.println("Thread2 is running...");
4. };

**Comparator**

1. // implementing lambda expression
2. Collections.sort(list,(p1,p2)->{
3. **return** p1.name.compareTo(p2.name);
4. });

**Filter Collection Data**

1. // using lambda to filter data
2. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
3. // using lambda to iterate through collection
4. filtered\_data.forEach(
5. product -> System.out.println(product.name+": "+product.price)
6. );

**Event Listener**

  b.addActionListener(e-> { tf.setText("hello swing");});

**Important points:**

> The body of a lambda expression can contain zero, one or more statements.

> When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.

> When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

**Java 8 functional interface example**

Functional interfaces are those interfaces which have only one abstract method, it can have default methods, static methods and it can also override java.lang.Object class method.  
There are many functional interfaces already present.  
For example: **Runnable** , **Comparable**.  
You can implement functional interfaces using [lambda expressions](https://www.java2blog.com/2014/06/lambda-expressions-in-java-8.html" \t "https://java2blog.com/java-8-functional-interface-example/_blank).

*// Using lambda expression*

Thread t1=**new** Thread(

()->System.out.println("In Run method")

);

Example of functional interface,@FunctionalInterface

public **interface** Decorable {

*// one abstract method*

**void** decorateWithCurtains();

*// default method*

**default** **void** decorateWithPaints(){

  System.out.println("Decorating using paints");

}

*// Overriding method of java.lang.Object*

@Override

public **int** hashCode();

}

Java can itself identify Functional Interface but you can also denote interface as Functional Interface by annotating it with @FunctionalInterface. If you annotate @FunctionalInterface, you should have only one abstract method otherwise you will get compilation error.

public **class** DecorableMain {

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

*// Using lambada expression*

Decorable dec=()->{System.out.println("Decorating with curtains");};

dec.decorateWithCurtains();

}

}

**How lambda expression and functional interfaces are related?**

Lambda expressions can only be applied to abstract method of functional interface.

Runnable has only one abstract method called run, so it can be used as below:

*// Using lambda expression*

Thread t1=**new** Thread(

()->System.out.println("In Run method")

);

Here we are using Thread constructor which takes Runnable as parameter. As you can see we did not specify any function name here, as Runnable has only one abstract method, java will implicitly create anonymous Runnable and execute run method.  
It will be as good as below code.

Thread t1=**new** Thread(**new** Runnable() {

   @Override

   public **void** run() {

    System.out.println("In Run method");

   }

  });

**Can you create your own functional interface?**

Yes, you can create your own functional interface. Java can implicitly identify functional interface but you can also annotate it with **@FunctionalInterface**.

**Java Method References**

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

**Types of Method References**

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

**Example : Reference to a constructor**

1. InstanceMethodReference methodReference = **new** InstanceMethodReference(); // Creating object
2. // Referring non-static method using reference
3. Sayable sayable = methodReference::saySomething;
4. // Calling interface method
5. sayable.say();
6. // Referring non-static method using anonymous object
7. Sayable sayable2 = **new** InstanceMethodReference()::saySomething; // You can use anonymous object also
8. // Calling interface method
9. sayable2.say();

**Reference to a constructor.**

1. Messageable hello = Message::**new**;
2. hello.getMessage("Hello");

**What is Optional? Why and how can you use it?**

Java 8 has introduced new class Called Optional. This class is basically introduced to avoid NullPointerException in java.  
Optional class encapsulates optional value which is either present or not.  
It is a wrapper around object and can be use to avoid NullPointerExceptions.

In Java 8, we have a newly introduced Optional class in java.util package. This class is introduced to avoid NullPointerException that we frequently encounters if we do not perform null checks in our code. Using this class we can easily check whether a variable has null value or not and by doing this we can avoid the NullPointerException. In this guide, we will see how to work with Optional class and the usage of various methods of this class.

public class Example {

public static void main(String[] args) {

  String[] str = new String[10];

Optional<String> isNull = Optional.ofNullable(str[9]);

  if(isNull.isPresent()){

 //Getting the substring

 String str2 = str[9].substring(2, 5);

 //Displaying substring

System.out.print("Substring is: "+ str2);

  }

  else{

  System.out.println("Cannot get the substring from an empty string");

  }

str[9] = "AgraIsCool";

Optional<String> isNull2 = Optional.ofNullable(str[9]);

  if(isNull2.isPresent()){

 //Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

}  }

public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

System.out.println(GOT.map(String::toLowerCase));

System.out.println(nothing.map(String::toLowerCase));

Optional<Optional<String>> anotherOptional = Optional.of(Optional.of("BreakingBad"));

System.out.println("Value of Optional object"+anotherOptional);

System.out.println("Optional.map: "

+anotherOptional.map(gender -> gender.map(String::toUpperCase)));

//Optional<Optional<String>>    -> flatMap -> Optional<String>

System.out.println("Optional.flatMap: "

 +anotherOptional.flatMap(gender -> gender.map(String::toUpperCase)));

}}

public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

  /\* Filter returns an empty Optional instance if the output doesn't

\* contain any value, else it returns the Optional object of the

\* given value.

\*/

System.out.println(GOT.filter(s -> s.equals("GAME OF THRONES")));

System.out.println(GOT.filter(s -> s.equalsIgnoreCase("GAME OF THRONES")));

}}

import java.util.Optional;

  public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

//orElse() method

System.out.println(GOT.orElse("Default Value"));

System.out.println(nothing.orElse("Default Value"));

//orElseGet() method

System.out.println(GOT.orElseGet(() -> "Default Value"));

System.out.println(nothing.orElseGet(() -> "Default Value"));

}}

import java.util.Optional;

  public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

/\* isPresent() method: Checks whether the given Optional

\* Object is empty or not.

\*/

if (GOT.isPresent()) {

  System.out.println("Watching Game of Thrones");

}

else {

System.out.println("I am getting Bored");

  }

/\* ifPresent() method: It executes only if the given Optional

\* object is non-empty.

\*/

//This will print as the GOT is non-empty

GOT.ifPresent(s -> System.out.println("Watching GOT is fun!"));

//This will not print as the nothing is empty

nothing.ifPresent(s -> System.out.println("I prefer getting bored"));

}}

public class Example {

public static void main(String[] args) {

  String[] str = new String[10];

Optional<String> isNull = Optional.ofNullable(str[9]);

  if(isNull.isPresent()){

 //Getting the substring

 String str2 = str[9].substring(2, 5);

 //Displaying substring

System.out.print("Substring is: "+ str2);

  }

  else{

  System.out.println("Cannot get the substring from an empty string");

  }

str[9] = "AgraIsCool";

Optional<String> isNull2 = Optional.ofNullable(str[9]);

  if(isNull2.isPresent()){

 //Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

}  }

**What are defaults methods?**

**Default method** are those methods in interface which have body and use default keywords.

Default method are introduced in Java 8 mainly because of backward compatibility.

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

**interface** TestInterface {

    // abstract method

**public** **void** square(**int** a);

    // default method

**default** **void** show() {

      System.out.println("Default Method Executed");

    }

}

**class** TestClass **implements** TestInterface {

    // implementation of square abstract method

**public** **void** square(**int** a)     {

        System.out.println(a\*a);

    }

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

        TestClass d = **new** TestClass();

        d.square(4);

        // default method executed

        d.show();    } }

The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as defender methods or virtual extension methods.

**Static Methods:**The interfaces can have static methods as well which is similar to static method of classes.

**interface** TestInterface {

    // abstract method

**public** **void** square (**int** a);

    // static method

**static** **void** show()     {

        System.out.println("Static Method Executed");

    }

}

**Default Methods and Multiple Inheritance**In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

**interface** TestInterface1 {

    // default method

**default** **void** show() {

        System.out.println("Default TestInterface1");

    }

}

**interface** TestInterface2 {

    // Default method

**default** **void** show() {

        System.out.println("Default TestInterface2");

    }

}

// Implementation class code

**class** TestClass **implements** TestInterface1, TestInterface2 {

    // Overriding default show method

**public** **void** show() {

        // use super keyword to call the show

        // method of TestInterface1 interface

        TestInterface1.**super**.show();

        // use super keyword to call the show

        // method of TestInterface2 interface

        TestInterface2.**super**.show();

    }

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

        TestClass d = **new** TestClass();

        d.show();

    }

}

**Important Points:**

1. Interfaces can have default methods with implementation in Java 8 on later.
2. Interfaces can have static methods as well, similar to static methods in classes.
3. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

**Predicate**

**Predicate in Collection**

// Java program to demonstrate working of predicates

// on collection. The program finds all admins in an

// arrayList of users.

**import** java.util.function.Predicate;

**import** java.util.\*;

**class** User {

    String name, role;

    User(String a, String b) {

        name = a;

        role = b;

    }

    String getRole() { **return** role; }

    String getName() { **return** name; }

**public** String toString() {

**return** "User Name : " + name + ", Role :" + role;

    }

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

        List<User> users = **new** ArrayList<User>();

        users.add(**new** User("John", "admin"));

        users.add(**new** User("Peter", "member"));

        List admins = process(users, (User u) -> u.getRole().equals("admin"));

        System.out.println(admins);

    }

**public** **static** List<User> process(List<User> users,

                            Predicate<User> predicat) {

        List<User> result = **new** ArrayList<User>();

**for** (User user: users)

**if** (predicat.test(user))

                result.add(user);

**return** result;

    }

}

**Are you aware of Date and Time API introduced in Java 8? What the issues with Old Date and time API?**

**Issues with old Date and TIme API:**

**Thread Safety:** You might be already aware that java.util.Date is mutable and not thread safe. Even java.text.SimpleDateFormat is also not Thread-Safe. New Java 8 date and time APIs are thread safe.

**Performance:** Java 8 ‘s new APIs are better in performance than old Java APIs.

**More Readable:** Old APIs such Calendar and Date are poorly designed and hard to understand. Java 8 Date and Time APIs are easy to understand and comply with ISO standards.

**Can you provide some APIs of Java 8 Date and Time?**

## **LocalDate**, **LocalTime**, and **LocalDateTime**are the Core API classes for Java 8. As the name suggests, these classes are local to context of observer. It denotes current date and time in context of Observer.

**New Date-Time API in Java 8**

**New date-time API is i**ntroduced in Java 8 to overcome the following drawbacks of old date-time API :

**Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is immutable and doesn’t have setter methods.

**Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

**Java 8 under the package java.time introduced a new date-time API, most important classes among them are :**

**Local :** Simplified date-time API with no complexity of timezone handling.

**Zoned :** Specialized date-time API to deal with various timezones.

// the current date LocalDate date = LocalDate.now();

// the current time

    LocalTime time = LocalTime.now();

// will give us the current time and date

    LocalDateTime current = LocalDateTime.now();

// to print in a particular format

    DateTimeFormatter format =

      DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

   String formatedDateTime = current.format(format);

// printing months days and seconds

    Month month = current.getMonth();

**int** day = current.getDayOfMonth();

**int** seconds = current.getSecond();

 // printing some specified date

    LocalDate date2 = LocalDate.of(1950,1,26);

 // printing date with current time.

    LocalDateTime specificDate =

        current.withDayOfMonth(24).withYear(2016);

// to get the current zone

    ZonedDateTime currentZone = ZonedDateTime.now();

**Period**and **Duration** classes :  
Period : It deals with date based amount of time.  
Duration : It deals with time based amount of time.

**ChronoUnits Enum :** java.time.temporal.ChronoUnit enum is added in Java 8 to replace integer values used in old API to represent day, month etc.

**TemporalAdjuster :**It is used to perform various date related operations.

## **Do we have PermGen in Java 8? Are you aware of MetaSpace?**

Until Java 7, JVM used an area called PermGen to store classes. It got removed in Java 8 and replaced by MetaSpace.  
Major advantage of MetaSpace over permgen:  
PermGen was fixed in term of mazimum size and can not grow dynamically but Metaspace can grow dynamically and do not have any size constraint.

**Java Stream interface method :**

**boolean allMatch(Predicate<? super T> predicate)**  : It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.

**boolean anyMatch(Predicate<? super T> predicate) :** It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.

**static <T> Stream.Builder<T> builder() :** It returns a builder for a Stream.

**<R,A> R collect(Collector<? super T,A,R> collector) :** It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

**<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) :** It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.

**static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) :** It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.

**long count() :**It returns the count of elements in this stream. This is a special case of a reduction.

**Stream<T> distinct() :** It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.

**static <T> Stream<T> empty() :** It returns an empty sequential Stream.

**Stream<T> filter(Predicate<? super T> predicate) :** It returns a stream consisting of the elements of this stream that match the given predicate.

**Optional<T> findAny() :** It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.

**Optional<T> findFirst() :**It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.

**<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) :** It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

**DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper):**  It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)

**IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) :** It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

**LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) :** It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

**void forEach(Consumer<? super T> action) :** It performs an action for each element of this stream.

**void forEachOrdered(Consumer<? super T> action) :**It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.

**static <T> Stream<T> generate(Supplier<T> s) :** It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

**static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) :**It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.

**Stream<T> limit(long maxSize):** It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.

**<R> Stream<R> map(Function<? super T,? extends R> mapper) :**It returns a stream consisting of the results of applying the given function to the elements of this stream.

**DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) :** It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.

**IntStream mapToInt(ToIntFunction<? super T> mapper) :**It returns an IntStream consisting of the results of applying the given function to the elements of this stream.

**LongStream mapToLong(ToLongFunction<? super T> mapper) :** It returns a LongStream consisting of the results of applying the given function to the elements of this stream.

**Optional<T> max(Comparator<? super T> comparator) :**It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.

**Optional<T> min(Comparator<? super T> comparator):** It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.

**boolean noneMatch(Predicate<? super T> predicate):** It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.

**@SafeVarargs static <T> Stream<T> of(T... values) :**It returns a sequential ordered stream whose elements are the specified values.

**static <T> Stream<T> of(T t) :**It returns a sequential Stream containing a single element.

**Stream<T> peek(Consumer<? super T> action) :** It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

**Optional<T> reduce(BinaryOperator<T> accumulator):** It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.

**T reduce(T identity, BinaryOperator<T> accumulator):** It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

**<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner):** It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.

**Stream<T> skip(long n) :** It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.

**Stream<T> sorted() :**It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.

**Stream<T> sorted(Comparator<? super T> comparator):** It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.

**Object[] toArray():** It returns an array containing the elements of this stream.

**<A> A[] toArray(IntFunction<A[]> generator) :**It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

1. List<Float> productPriceList2 =productsList.stream()
2. .filter(p -> p.price > 30000)// filtering data
3. .map(p->p.price)        // fetching price
4. .collect(Collectors.toList()); // collecting as list
5. Stream.iterate(1, element->element+1)
6. .filter(element->element%5==0)
7. .limit(5)
8. .forEach(System.out::println);
9. // This is more compact approach for filtering data
10. productsList.stream()
11. .filter(product -> product.price == 30000)
12. .forEach(product -> System.out.println(product.name));
13. // This is more compact approach for filtering data
14. Float totalPrice = productsList.stream()
15. .map(product->product.price)
16. .reduce(0.0f,(sum, price)->sum+price);   // accumulating price
17. System.out.println(totalPrice);
18. // More precise code
19. **float** totalPrice2 = productsList.stream()
20. .map(product->product.price)
21. .reduce(0.0f,Float::sum);   // accumulating price, by referring method of Float class
22. System.out.println(totalPrice2);
23. // Using Collectors's method to sum the prices.
24. **double** totalPrice3 = productsList.stream()
25. .collect(Collectors.summingDouble(product->product.price));
26. // max() method to get max Product price
27. Product productA = productsList.stream()
28. .max((product1, product2)->
29. product1.price > product2.price ? 1: -1).get();
31. System.out.println(productA.price);
32. // min() method to get min Product price
33. Product productB = productsList.stream()
34. .max((product1, product2)->
35. product1.price < product2.price ? 1: -1).get();
36. System.out.println(productB.price);
37. // count number of products based on the filter
38. **long** count = productsList.stream()
39. .filter(product->product.price<30000)
40. .count();
41. System.out.println(count);
42. // Converting product List into Set
43. Set<Float> productPriceList =
44. productsList.stream()
45. .filter(product->product.price < 30000)   // filter product on the base of price
46. .map(product->product.price)
47. .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
48. System.out.println(productPriceList);
49. // Converting Product List into a Map
50. Map<Integer,String> productPriceMap =
51. productsList.stream()
52. .collect(Collectors.toMap(p->p.id, p->p.name));
53. System.out.println(productPriceMap);
54. List<Float> productPriceList =
55. productsList.stream()
56. .filter(p -> p.price > 30000) // filtering data
57. .map(Product::getPrice)         // fetching price by referring getPrice method
58. .collect(Collectors.toList());  // collecting as list
59. System.out.println(productPriceList);

Java 8 Streams from creation to parallel execution.

To understand this material, readers need to have a basic knowledge of Java 8 (lambda expressions, Optional, method references) and of the Stream API.

**Stream Creation**

There are many ways to create a stream instance of different sources. Once created, the instance will not modify its source, therefore allowing the creation of multiple instances from a single source.

**2.1. Empty Stream**

The empty() method should be used in case of a creation of an empty stream:

Stream<String> streamEmpty = Stream.empty();

Its often the case that the empty() method is used upon creation to avoid returning null for streams with no element:

public Stream<String> streamOf(List<String> list) {

return list == null || list.isEmpty() ? Stream.empty() : list.stream();

}

**2.2. Stream of Collection**

Stream can also be created of any type of Collection (Collection, List, Set):

Collection<String> collection = Arrays.asList("a", "b", "c");

Stream<String> streamOfCollection = collection.stream();

**2.3. Stream of Array**

Array can also be a source of a Stream:

Stream<String> streamOfArray = Stream.of("a", "b", "c");

They can also be created out of an existing array or of a part of an array:

String[] arr = new String[]{"a", "b", "c"};

Stream<String> streamOfArrayFull = Arrays.stream(arr);

Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);

**2.4. Stream.builder()**

When builder is used the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream<Object>:

Stream<String> streamBuilder =

Stream.<String>builder().add("a").add("b").add("c").build();

**2.5. Stream.generate()**

The generate() method accepts a Supplier<T> for element generation. As the resulting stream is infinite, developer should specify the desired size or the generate() method will work until it reaches the memory limit:

Stream<String> streamGenerated =

Stream.generate(() -> "element").limit(10);

The code above creates a sequence of ten strings with the value – “element”.

**2.6. Stream.iterate()**

Another way of creating an infinite stream is by using the iterate() method:

Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);

The first element of the resulting stream is a first parameter of the iterate() method. For creating every following element the specified function is applied to the previous element. In the example above the second element will be 42.

**2.7. Stream of Primitives**

Java 8 offers a possibility to create streams out of three primitive types: int, long and double. As Stream<T> is a generic interface and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: IntStream, LongStream, DoubleStream.

Using the new interfaces alleviates unnecessary auto-boxing allows increased productivity:

IntStream intStream = IntStream.range(1, 3);

LongStream longStream = LongStream.rangeClosed(1, 3);

The range(int startInclusive, int endExclusive) method creates an ordered stream from the first parameter to the second parameter. It increments the value of subsequent elements with the step equal to 1. The result doesn't include the last parameter, it is just an upper bound of the sequence.

The rangeClosed(int startInclusive, int endInclusive) method does the same with only one difference – the second element is included. These two methods can be used to generate any of the three types of streams of primitives.

Since Java 8 the Random class provides a wide range of methods for generation streams of primitives. For example, the following code creates a DoubleStream, which has three elements:

Random random = new Random();

DoubleStream doubleStream = random.doubles(3);

**2.8. Stream of String**

String can also be used as a source for creating a stream.

With the help of the chars() method of the String class. Since there is no interface CharStream in JDK, the IntStream is used to represent a stream of chars instead.

IntStream streamOfChars = "abc".chars();

The following example breaks a String into sub-strings according to specified RegEx:

Stream<String> streamOfString =

Pattern.compile(", ").splitAsStream("a, b, c");

**2.9. Stream of File**

Java NIO class Files allows to generate a Stream<String> of a text file through the lines() method. Every line of the text becomes an element of the stream:

Path path = Paths.get("C:\\file.txt");

Stream<String> streamOfStrings = Files.lines(path);

Stream<String> streamWithCharset =

Files.lines(path, Charset.forName("UTF-8"));

The Charset can be specified as an argument of the lines() method.

**3. Referencing a Stream**

It is possible to instantiate a stream and to have an accessible reference to it as long as only intermediate operations were called. Executing a terminal operation makes a stream inaccessible.

To demonstrate this we will forget for a while that the best practice is to chain sequence of operation. Besides its unnecessary verbosity, technically the following code is valid:

Stream<String> stream =

Stream.of("a", "b", "c").filter(element -> element.contains("b"));

Optional<String> anyElement = stream.findAny();

But an attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException:

Optional<String> firstElement = stream.findFirst();

As the IllegalStateException is a RuntimeException, a compiler will not signalize about a problem. So, it is very important to remember that Java 8 streams can't be reused.

This kind of behavior is logical because streams were designed to provide an ability to apply a finite sequence of operations to the source of elements in a functional style, but not to store elements.

So, to make previous code work properly some changes should be done:

List<String> elements =

Stream.of("a", "b", "c").filter(element -> element.contains("b"))

.collect(Collectors.toList());

Optional<String> anyElement = elements.stream().findAny();

Optional<String> firstElement = elements.stream().findFirst();

**4. Stream Pipeline**

To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – the source, intermediate operation(s) and a terminal operation.

Intermediate operations return a new modified stream. For example, to create a new stream of the existing one without few elements the skip() method should be used:

Stream<String> onceModifiedStream =

Stream.of("abcd", "bbcd", "cbcd").skip(1);

If more than one modification is needed, intermediate operations can be chained. Assume that we also need to substitute every element of current Stream<String> with a sub-string of first few chars. This will be done by chaining the skip() and the map() methods:

Stream<String> twiceModifiedStream =

stream.skip(1).map(element -> element.substring(0, 3));

As you can see, the map() method takes a lambda expression as a parameter. If you want to learn more about lambdas take a look at our tutorial Lambda Expressions and Functional Interfaces: Tips and Best Practices.

A stream by itself is worthless, the real thing a user is interested in is a result of the terminal operation, which can be a value of some type or an action applied to every element of the stream. Only one terminal operation can be used per stream.

The right and most convenient way to use streams are by a stream pipeline, which is a chain of stream source, intermediate operations, and a terminal operation. For example:

List<String> list = Arrays.asList("abc1", "abc2", "abc3");

long size = list.stream().skip(1)

.map(element -> element.substring(0, 3)).sorted().count();

**5. Lazy Invocation**

Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.

To demonstrate this, imagine that we have method wasCalled(), which increments an inner counter every time it was called:

private long counter;

private void wasCalled() {

counter++;

}

Let's call method wasCalled() from operation filter():

List<String> list = Arrays.asList(“abc1”, “abc2”, “abc3”);

counter = 0;

Stream<String> stream = list.stream().filter(element -> {

wasCalled();

return element.contains("2");

});

As we have a source of three elements we can assume that method filter() will be called three times and the value of the counter variable will be 3. But running this code doesn't change counter at all, it is still zero, so, the filter() method wasn't called even once. The reason why – is missing of the terminal operation.

Let's rewrite this code a little bit by adding a map() operation and a terminal operation – findFirst(). We will also add an ability to track an order of method calls with a help of logging:

Optional<String> stream = list.stream().filter(element -> {

log.info("filter() was called");

return element.contains("2");

}).map(element -> {

log.info("map() was called");

return element.toUpperCase();

}).findFirst();

Resulting log shows that the filter() method was called twice and the map() method just once. It is so because the pipeline executes vertically. In our example the first element of the stream didn't satisfy filter's predicate, then the filter() method was invoked for the second element, which passed the filter. Without calling the filter() for third element we went down through pipeline to the map() method.

The findFirst() operation satisfies by just one element. So, in this particular example the lazy invocation allowed to avoid two method calls – one for the filter() and one for the map().

**6. Order of Execution**

From the performance point of view, the right order is one of the most important aspects of chaining operations in the stream pipeline:

long size = list.stream().map(element -> {

wasCalled();

return element.substring(0, 3);

}).skip(2).count();

Execution of this code will increase the value of the counter by three. This means that the map() method of the stream was called three times. But the value of the size is one. So, resulting stream has just one element and we executed the expensive map() operations for no reason twice out of three times.

If we change the order of the skip() and the map() methods, the counter will increase only by one. So, the method map() will be called just once:

long size = list.stream().skip(2).map(element -> {

wasCalled();

return element.substring(0, 3);

}).count();

This brings us up to the rule: intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element. So, keep such methods as skip(), filter(), distinct() at the top of your stream pipeline.

**7. Stream Reduction**

The API has many terminal operations which aggregate a stream to a type or to a primitive, for example, count(), max(), min(), sum(), but these operations work according to the predefined implementation.

There are two methods which allow to do this – the reduce() and the collect() methods.

**7.1. The reduce() Method**

There are three variations of this method, which differ by their signatures and returning types. They can have the following parameters:

**identity** – the initial value for an accumulator or a default value if a stream is empty and there is nothing to accumulate;

**accumulator** – a function which specifies a logic of aggregation of elements. As accumulator creates a new value for every step of reducing, the quantity of new values equals to the stream's size and only the last value is useful. This is not very good for the performance.

**combiner** – a function which aggregates results of the accumulator. Combiner is called only in a parallel mode to reduce results of accumulators from different threads.

So, let's look at these three methods in action:

OptionalInt reduced =

IntStream.range(1, 4).reduce((a, b) -> a + b);

reduced = 6 (1 + 2 + 3)

int reducedTwoParams =

IntStream.range(1, 4).reduce(10, (a, b) -> a + b);

reducedTwoParams = 16 (10 + 1 + 2 + 3)

int reducedParams = Stream.of(1, 2, 3)

.reduce(10, (a, b) -> a + b, (a, b) -> {

log.info("combiner was called");

return a + b;

});

The result will be the same as in the previous example (16) and there will be no login which means, that combiner wasn't called. To make a combiner work, a stream should be parallel:

int reducedParallel = Arrays.asList(1, 2, 3).parallelStream()

.reduce(10, (a, b) -> a + b, (a, b) -> {

log.info("combiner was called");

return a + b;

});

The result here is different (36) and the combiner was called twice. Here the reduction works by the following algorithm: accumulator ran three times by adding every element of the stream to identity to every element of the stream. These actions are being done in parallel. As a result, they have (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;). Now combiner can merge these three results. It needs two iterations for that (12 + 13 = 25; 25 + 11 = 36).

**7.2. The collect() Method**

Reduction of a stream can also be executed by another terminal operation – the collect() method. It accepts an argument of the type Collector, which specifies the mechanism of reduction. There are already created predefined collectors for most common operations. They can be accessed with the help of the Collectors type.

In this section we will use the following List as a source for all streams:

List<Product> productList = Arrays.asList(new Product(23, "potatoes"),

new Product(14, "orange"), new Product(13, "lemon"),

new Product(23, "bread"), new Product(13, "sugar"));

Converting a stream to the Collection (Collection, List or Set):

List<String> collectorCollection =

productList.stream().map(Product::getName).collect(Collectors.toList());

Reducing to String:

String listToString = productList.stream().map(Product::getName)

.collect(Collectors.joining(", ", "[", "]"));

The joiner() method can have from one to three parameters (delimiter, prefix, suffix). The handiest thing about using joiner() – developer doesn't need to check if the stream reaches its end to apply the suffix and not to apply a delimiter. Collector will take care of that.

Processing the average value of all numeric elements of the stream:

double averagePrice = productList.stream()

.collect(Collectors.averagingInt(Product::getPrice));

Processing the sum of all numeric elements of the stream:

int summingPrice = productList.stream()

.collect(Collectors.summingInt(Product::getPrice));

Methods averagingXX(), summingXX() and summarizingXX() can work as with primitives (int, long, double) as with their wrapper classes (Integer, Long, Double). One more powerful feature of these methods is providing the mapping. So, developer doesn't need to use an additional map() operation before the collect() method.

Collecting statistical information about stream’s elements:

IntSummaryStatistics statistics = productList.stream()

.collect(Collectors.summarizingInt(Product::getPrice));

By using the resulting instance of type IntSummaryStatistics developer can create a statistical report by applying toString() method. The result will be a String common to this one “IntSummaryStatistics{count=5, sum=86, min=13, average=17,200000, max=23}”.

It is also easy to extract from this object separate values for count, sum, min, average by applying methods getCount(), getSum(), getMin(), getAverage(), getMax(). All these values can be extracted from a single pipeline.

Grouping of stream’s elements according to the specified function:

Map<Integer, List<Product>> collectorMapOfLists = productList.stream()

.collect(Collectors.groupingBy(Product::getPrice));

In the example above the stream was reduced to the Map which groups all products by their price.

Dividing stream’s elements into groups according to some predicate:

Map<Boolean, List<Product>> mapPartioned = productList.stream()

.collect(Collectors.partitioningBy(element -> element.getPrice() > 15));

Pushing the collector to perform additional transformation:

Set<Product> unmodifiableSet = productList.stream()

.collect(Collectors.collectingAndThen(Collectors.toSet(),

Collections::unmodifiableSet));

In this particular case, the collector has converted a stream to a Set and then created the unmodifiable Set out of it.

Custom collector:

If for some reason, a custom collector should be created, the most easier and the less verbose way of doing so – is to use the method of() of the type Collector.

Collector<Product, ?, LinkedList<Product>> toLinkedList =

Collector.of(LinkedList::new, LinkedList::add,

(first, second) -> {

first.addAll(second);

return first;

});

LinkedList<Product> linkedListOfPersons =

productList.stream().collect(toLinkedList);

**Parallel Streams**

Before Java 8, parallelization was complex. Emerging of the ExecutorService and the ForkJoin simplified developer’s life a little bit, but they still should keep in mind how to create a specific executor, how to run it and so on. Java 8 introduced a way of accomplishing parallelism in a functional style.

The API allows creating parallel streams, which perform operations in a parallel mode. When the source of a stream is a Collection or an array it can be achieved with the help of the parallelStream() method:

Stream<Product> streamOfCollection = productList.parallelStream();

boolean isParallel = streamOfCollection.isParallel();

boolean bigPrice = streamOfCollection

.map(product -> product.getPrice() \* 12)

.anyMatch(price -> price > 200);

If the source of stream is something different than a Collection or an array, the parallel() method should be used:

IntStream intStreamParallel = IntStream.range(1, 150).parallel();

boolean isParallel = intStreamParallel.isParallel();

Under the hood, Stream API automatically uses the ForkJoin framework to execute operations in parallel. By default, the common thread pool will be used and there is no way (at least for now) to assign some custom thread pool to it. This can be overcome by using a custom set of parallel collectors.

When using streams in parallel mode, avoid blocking operations and use parallel mode when tasks need the similar amount of time to execute (if one task lasts much longer than the other, it can slow down the complete app’s workflow).

The stream in parallel mode can be converted back to the sequential mode by using the sequential() method:

IntStream intStreamSequential = intStreamParallel.sequential();

boolean isParallel = intStreamSequential.isParallel();

**Java Streams in Java 8**

Stream is one of the major features added to Java 8.Java 8 Streams should not be confused with Java I/O streams (ex: FileInputStream etc).

Streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.

This functionality – java.util.stream – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

#### Java Stream Creation

First obtain a stream from an existing array:

private static Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

**Stream stream** **= Stream.of(arrayOfEmps);**

**We can also obtain a stream from an existing list:**

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();

**Note** that Java 8 added a new stream() method to the Collection interface.

And we can create a stream from individual objects using Stream.of():

Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);

// Creating an Stream having single element only

Stream stream = Stream.of("Geeks");

// Displaying the Stream having single element

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

// Creating an Stream

        Stream stream = Stream.of("Geeks", "for", "Geeks");

// Displaying the sequential ordered stream

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

**simply using Stream.builder():**

Stream.Builder<Employee> empStreamBuilder = Stream.builder();

empStreamBuilder.accept(arrayOfEmps[0]);

empStreamBuilder.accept(arrayOfEmps[1]);

empStreamBuilder.accept(arrayOfEmps[2]);

Stream<Employee> empStream = empStreamBuilder.build();

**Java Stream Operations :**

some common usages and operations we can perform on and with the help of the stream support in the language.

#### **forEach**

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream().forEach(e -> e.salaryIncrement(10.0));

**forEach()** is a terminal operation, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used.

#### **map**

**map()** produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

Integer[] empIds = { 1, 2, 3 };

List<Employee> employees = Stream.of(empIds)

.map(employeeRepository::findById)

.collect(Collectors.toList());

Here, we obtain an Integer stream of employee ids from an array. Each Integer is passed to the function employeeRepository::findById() – which returns the corresponding Employee object; this effectively forms an Employee stream.

#### **collect**

its one of the common ways to get stuff out of the stream once we are done with all the processing.

collect() performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the Stream instance.

List<Employee> employees = empList.stream().collect(Collectors.toList());

The strategy for this operation is provided via the Collector interface implementation. In the example above, we used the toList collector to collect all Stream elements into a List instance.

#### **filter**

filter() : this produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

Integer[] empIds = { 1, 2, 3, 4 };

List<Employee> employees = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 200000)

.collect(Collectors.toList());

we first filter out null references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

#### **findFirst**

findFirst() returns an Optional for the first entry in the stream; the Optional can be empty

Integer[] empIds = { 1, 2, 3, 4 };

Employee employee = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 100000)

.findFirst()

.orElse(null);

the first employee with the salary greater than 100000 is returned. If no such employee exists, then null is returned.

#### **toArray**

If we need to get an array out of the stream, we can simply use toArray():

Employee[] employees = empList.stream().toArray(Employee[]::new);

The syntax Employee[]::new creates an empty array of Employee – which is then filled with elements from the stream.

#### **flatMap**

#### A stream can hold complex data structures like Stream<List<String>>. flatMap() helps us to flatten the data structure to simplify further operations:

List<List<String>> namesNested = Arrays.asList(

Arrays.asList("Jeff", "Bezos"),

Arrays.asList("Bill", "Gates"),

Arrays.asList("Mark", "Zuckerberg"));

List<String> namesFlatStream = namesNested.stream()

.flatMap(Collection::stream)

.collect(Collectors.toList());

we were able to convert the Stream<List<String>> to a simpler Stream<String> – using the flatMap() API.

### Method Types and Pipelines

Java stream operations are divided into intermediate and terminal operations.**Intermediate** operations such as filter() return a new stream on which further processing can be done. **Terminal** operations, such as forEach(), mark the stream as consumed.

A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Here’s a sample stream pipeline, where empList is the source, filter() is the intermediate operation and count is the terminal operation:

Long empCount = empList.stream()

.filter(e -> e.getSalary() > 200000)

.count();

Some operations are deemed short-circuiting operations. Short-circuiting operations allow computations on infinite streams to complete in finite time:

Stream<Integer> infiniteStream = Stream.iterate(2, i -> i \* 2);

List<Integer> collect = infiniteStream

.skip(3)

.limit(5)

.collect(Collectors.toList());

Here, we use short-circuiting operations skip() to skip first 3 elements, and limit() to limit to 5 elements from the infinite stream generated using iterate().

### **Lazy Evaluation**

One of the most important characteristics of Java streams is that they allow for significant optimizations through lazy evaluations.

Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

All intermediate operations are lazy, so they’re not executed until a result of a processing is actually needed.

For example, consider the findFirst() example we saw earlier. How many times is the map() operation performed here? 4 times, since the input array contains 4 elements?

Integer[] empIds = { 1, 2, 3, 4 };

Employee employee = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 100000)

.findFirst()

.orElse(null);

Stream performs the map and two filter operations, one element at a time.

It first performs all the operations on id 1. Since the salary of id 1 is not greater than 100000, the processing moves on to the next element.

Id 2 satisfies both of the filter predicates and hence the stream evaluates the terminal operation findFirst() and returns the result.

No operations are performed on id 3 and 4.

Processing streams lazily allows avoiding examining all the data when that’s not necessary. This behavior becomes even more important when the input stream is infinite and not just very large.

### **Comparison Based Stream Operations**

#### **Sorted :**this sorts the stream elements based on the comparator passed we pass into it.

List<Employee> employees = empList.stream()

.sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))

.collect(Collectors.toList());

#### **min and max :** As the name suggests, min() and max() return the minimum and maximum element in the stream respectively, based on a comparator. They return an Optional since a result may or may not exist

Employee firstEmp = empList.stream()

.min((e1, e2) -> e1.getId() - e2.getId())

.orElseThrow(NoSuchElementException::new);

We can also avoid defining the comparison logic by using **Comparator.comparing():**

Employee maxSalEmp = empList.stream()

.max(Comparator.comparing(Employee::getSalary))

.orElseThrow(NoSuchElementException::new);

**Distinct :** distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not:

List<Integer> intList = Arrays.asList(2, 5, 3, 2, 4, 3);

List<Integer> distinctIntList = intList.stream().distinct().collect(Collectors.toList());

**allMatch, anyMatch, and noneMatch**

These operations all take a predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined.

List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);

boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);

boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);

**allMatch()** checks if the predicate is true for all the elements in the stream. Here, it returns false as soon as it encounters 5, which is not divisible by 2.

**anyMatch()** checks if the predicate is true for any one element in the stream. Here, again short-circuiting is applied and true is returned immediately after the first element.

**noneMatch()** checks if there are no elements matching the predicate. Here, it simply returns false as soon as it encounters 6, which is divisible by 3.

# **Difference between Streams and Collections in Java 8**

Java Collections framework is used for storing and manipulating group of data. It is an in-memory data structure and every element in the collection should be computed before it can be added in the collections.

Stream API is only used for processing group of data. It does not modify the actual collection, they only provide the result as per the pipelined methods.

|  |  |  |
| --- | --- | --- |
| Key | Collections | Streams |
| Basic | It is used for storing and manipulating group of data. | Stream API is only used for processing group of data. |
| Package | All the classes and interfaces of this API is in the Java.util package | All the classes and interfaces of this API is in the java.util.stream  package |
| Eager/Lazy | All the elements in the collections are computed in the beginning. | In streams, intermediate operations are lazy. |
| Data Modification | In collections, we can remove or add elements. | We can’t modify streams. |
| External /Internal iterator | Collections perform iteration over the collection. | Stream perform iteration   internally. |

Collection is used for storing data in different data structures while Stream API is used for computation of data on a large set of Objects.

Collection API we can store a finite number of elements in a data structure. With Stream API, we can handle streams of data that can contain infinite number of elements.

Eager vs. Lazy: Collection API constructs objects in an eager manner. Stream API creates objects in a lazy manner.

Multiple consumption: Most of the Collection APIs support iteration and consumption of elements multiple times. With Stream API we can consume or iterate elements only once.

# Java provides a new additional package in Java 8 called **java.util.stream**. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

**Stream provides following features:**

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

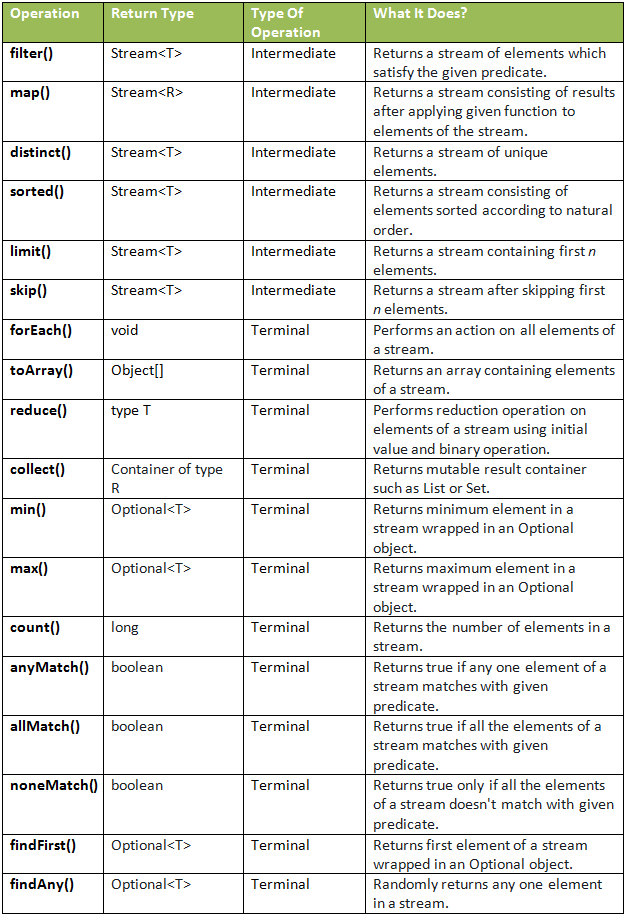
You can use stream to filter, collect, print, and convert from one data structure to other etc.

### **Java 8 Stream Intermediate And Terminal Operations :**

Some operations produce another stream as a result and some operations produce non-stream values as a result. The operations which return another stream as a result are called **intermediate** operations and the operations which return non-stream values like primitive or object or collection or return nothing are called **terminal** operations.

|  |  |
| --- | --- |
| **Intermediate** | **Terminal** |
| They return stream. Intermediate operations return a stream as a result | They return non-stream values. terminal operations return non-stream values like primitive or object or collection or may not return anything. |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory | Terminal operations are eagerly loaded.and executed when the terminal operation is called on the stream. |
| They don’t produce end result.  intermediate operations doesn’t give end result. They just transform one stream to another stream. | terminal operations give end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

**Below is the list of intermediate and terminal operations :**



Using Java 8 Streams, you can write most complex data processing queries without much difficulties.

### What Are Streams? :- Streams can be defined as a sequences of elements from a source which support data processing operations. You can treat streams as operations on data.

### **Why Streams? :** Almost every Java application use Collections API to store and process the data. the most used Java API, it is not easy to write the code for even some common data processing operations like filtering, finding, matching, sorting, mapping etc using Collections API . So, there needed Next-Gen API to process the data. So Java API designers have come with Java 8 Streams API to write more complex data processing operations with much of ease.

**3) Characteristics Of Java 8 Streams**

**3.1) Streams are not the data structures :** Streams doesn’t store the data. You can’t add or remove elements from streams. Hence, they are not the data structures. They are the just operations on data.

**3.2) Stream Consumes a data source :**Stream consumes a source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. Remember, stream doesn’t modify the source.

**3.3) Intermediate And Terminal Operations :**Most of the stream operations return another new stream and they can be chained together to form a pipeline of operations.

The operations which return stream themselves are called intermediate operations. For example – filter(), distinct(), sorted() etc.

The operations which return other than stream are called terminal operations. count(). min(), max() are some terminal operations.

**3.4) Pipeline Of Operations :** A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. In the below example, int array is the source, filter() and distinct() are intermediate operations and forEach() is a terminal operation.

IntStream.of(new int[] {4, 7, 1, 8, 3, 9, 7}).filter((int i) -> i >5).distinct().forEach(System.out::println);

**3.5) Internal Iteration :** Collections need to be iterated explicitly. i.e you have to write the code to iterate over collections. But, all stream operations do the iteration internally behind the scene for you. You need not to worry about iteration at all while writing the code using Java 8 Streams API.

**3.6) Parallel Execution :**To gain the performance while processing the large amount of data, you have to process it in parallel and use multi core architectures. Java 8 Streams can be processed in parallel without writing any multi threaded code. For example, to process the collections in parallel, you just use parallelStream() method instead of stream() method.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

**//Normal Execution**

names.stream().filter((String name) -> name.length() > 5).skip(2).forEach(System.out::println);

**//Parallel Execution**

names.parallelStream().filter((String name) -> name.length() > 5).skip(2).forEach(System.out::println);

**3.7) Streams are lazily populated :**All elements of a stream are not populated at a time. They are lazily populated as per demand because intermediate operations are not evaluated until terminal operation is invoked.

**3.8) Streams are traversable only once :**You can’t traverse the streams more than once just like iterators. If you traverse the stream first time, it is said to be consumed.

List<String> nameList = Arrays.asList("Dinesh", "Ross", "Kagiso", "Steyn");

Stream<String> stream = nameList.stream();

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

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

//Error : stream has already been operated upon or closed

**3.9) Short Circuiting Operations :**Short circuiting operations are the operations which don’t need the whole stream to be processed to produce a result. For example – findFirst(), findAny(), limit() etc.

## **1 . Java 8 Stream Operations :**

## **Stream Creation Operations**

**empty() : Creates an empty stream**

**Method Signature :** public static<T> Stream<T> empty()

Type Of Method : Static Method

What It Does? : Returns an empty stream of type T.

Stream<Student> emptyStream = Stream.empty();

System.out.println(emptyStream.count()); //Output : 0

**of(T t) : Creates a stream of single element of type T**

**Method Signature :** public static<T> Stream<T> of(T t)

Type Of Method : Static Method

What It Does? : Returns a single element stream of type T.

Stream<Student> singleElementStream = Stream.of(new Student());

System.out.println(singleElementStream.count());//Output : 1

**of(T… values) : Creates a stream from values**

**Method Signature :** public static<T> Stream<T> of(T… values)

Type Of Method : Static Method

What It does? : Returns a stream consisting of supplied values as elements.

Stream<Integer> streamOfNumbers = Stream.of(7, 2, 6, 9, 4, 3, 1);

System.out.println(streamOfNumbers.count());//Output : 7

**Creating streams from collections**

From Java 8, every collection type will have a method called **stream()** which returns the stream of respective collection type.

Example : Creating a stream from List :

List<String> listOfStrings = new ArrayList<>();

listOfStrings.add("One"); listOfStrings.add("Two"); listOfStrings.add("Three");

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

// Output :

// One

// Two // Three

### **Selection Operations**

**filter() : Selecting with a predicate**

**Method Signature :** Stream<T> filter(Predicate<T> predicate)

Type Of Operation : Intermediate Operation

What it does? : Returns a stream of elements which satisfy the given predicate.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

//Selecting names containing more than 5 characters

names.stream().filter((String name) -> name.length() > 5).forEach(System.out::println);

// Output :

// Johnson

// Samontik

**distinct() : Selects only unique elements**

**Method Signature :** Stream<T> distinct()

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream of unique elements.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika"); //Selecting only unique names

names.stream().distinct().forEach(System.out::println);

**limit() : Selects first n elements**

**Method Signature :** Stream<T> limit(long maxSize)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream containing first n elements.

names.stream().limit(4).forEach(System.out::println);

**skip() : Skips first n elements**

**Method Signature :** Stream<T> skip(long n)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream after skipping first n elements.

### **Mapping Operations**

**map() : Applies a function**

**Method Signature :** Stream<R> map(Function<T, R> mapper);

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of results after applying given function to elements of the stream.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Returns length of each name

names.stream().**map(String::length)**.forEach(System.out::println);

Output : 5 7 9 7 4

Other versions of map() method : **mapToInt(), mapToLong() and mapToDouble()**.

### **Sorting Operations**

**sorted() : Sorting according to natural order**

**Method Signature :** Stream<T> sorted()

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of elements sorted according to natural order.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Sorting the names according to natural order

names.stream().sorted().forEach(System.out::println);

// Output :

// Brijesh // David // John// Johnson // Samontika

**sorted(Comparator) : Sorting according to supplied comparator**

**Method Signature :** Stream<T> sorted(Comparator<T> comparator)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of elements sorted according to supplied Comparator.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

//Sorting the names according to their length

names.stream().sorted((String name1, String name2) -> name1.length() - name2.length()).forEach(System.out::println);

// Output :

// John// David// Johnson// Brijesh// Samontika

### **Reducing Operations** Reducing operations are the operations which combine all the elements of a stream repeatedly to produce a single value. For example, counting number of elements, calculating average of elements, finding maximum or minimum of elements etc.

**reduce() : Produces a single value**

Method Signature : T reduce(T identity, BinaryOperator<T> accumulator);

Type Of Operation : Terminal Operation

What It Does? : This method performs reduction operation on elements of a stream using initial value and binary operation.

int sum = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).reduce(0, (a, b) -> a+b);

//Output : 32

There is another form of reduce() method which takes no initial value. But returns an Optional object.

OptionalInt sum = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).reduce((a, b) -> a+b);

//Output : OptionalInt[32]

**Methods min(), max(), count() and collect() are special cases of reduction operation.**

**min() : Finding the minimum**

Method Signature : Optional<T> min(Comparator<T> comparator)

Type Of Operation : Terminal Operation

What It Does? : It returns minimum element in a stream wrapped in an Optional object.

OptionalInt min = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).min();

//Output : OptionalInt[1]

//Here, min() of IntStream will be used as we are passing an array of ints

**max() : Finding the maximum**

Method Signature : Optional<T> max(Comparator<T> comparator)

Type Of Operation : Terminal Operation

What It Does? : It returns maximum element in a stream wrapped in an Optional object.

OptionalInt max = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).max();

//Output : OptionalInt[9]

//Here, max() of IntStream will be used as we are passing an array of ints

**count() : Counting the elements**

Method Signature : long count()

Type Of Operation : Terminal Operation

What It Does? : Returns the number of elements in a stream.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Counting the names with length > 5

long noOfBigNames = names.stream().filter((String name) -> name.length() > 5).count();

System.out.println(noOfBigNames);

// Output : 3

**collect() : Returns mutable container**

Method Signature : R collect(Collector<T> collector)

Type Of Operation : Terminal Operation

What It Does? : collect() method is a special case of reduction operation called mutable reduction operation because it returns mutable result container such as List or Set.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Storing first 3 names in a mutable container

List<String> first3Names = names.stream().limit(3).collect(Collectors.toList());

System.out.println(first3Names);

// Output : [David, Johnson, Samontika]

#### **anyMatch() : Any one element matches**

#### **allMatch() : All elements matches**

#### **noneMatch() : No element matches**

#### **findFirst() : Finding first element**

#### **findAny() : Finding any element**

**forEach() :**

Method Signature : void forEach(Consumer<T> action)

Type Of Operation : Terminal Operation

What It Does? : Performs an action on all elements of a stream.

Stream.of("First", "Second", "Second", "Third", "Fourth").limit(3).distinct().forEach(System.out::println);

// Output

// First

// Second

**toArray() : Stream to array**

Method Signature : Object[] toArray()

Type Of Operation : Terminal Operation

What It Does? : Returns an array containing elements of a stream.

List<String> names = new ArrayList<>();

//Storing first 3 names in an array

Object[] streamArray = names.stream().limit(3).toArray();

System.out.println(Arrays.toString(streamArray));

// Output

// [David, Johnson, Samontika]

**peek() :**

Method Signature : Stream<T> peek(Consumer<T> action)

Type Of Operation : Intermediate Operation

What It Does? : Performs an additional action on each element of a stream. This method is only to support debugging where you want to see the elements as you pass in a pipeline.

List<String> names = new ArrayList<>();

names.stream()

.filter(name -> name.length() > 5)

.peek(e -> System.out.println("Filtered Name :"+e))

.map(String::toUpperCase)

.peek(e -> System.out.println("Mapped Name :"+e))

.toArray();

//Output :

//Filtered Name :Johnson

//Mapped Name :JOHNSON

//Filtered Name :Samontika

//Mapped Name :SAMONTIKA

//Filtered Name :Brijesh

//Mapped Name :BRIJESH