**Java8 Features:**

**---------------**

1. Functional Interfaces and lambda expressions.
2. for each() Method in Iterable interface.
3. Optional Class
4. Default And Static Methods in Interfaces.
5. Java Stream API For Bulk Data Operations on Collections.
6. Java Date Time API.
7. Collection API Improvements
8. **Functional Interfaces and lambda expressions.**

* A Functional Interface is an interface that has exactly one abstract method.
* The @FunctionalInterface annotation prevents abstract methods from being accidentally added to functional interfaces.
* It’s similar to a @Override annotation, and it’s recommended that you use it. java.lang. Runnable is a fantastic example of a functional interface since it has one abstract method, run ().
* One of the most appealing features of the functional interface is creating objects using lambda expressions.

@FunctionalInterface

public interface FunctionalInterface\_one

{

public void firstInt\_method();

@Override

public String toString(); //Overridden from Object class

@Override

public boolean equals(Object obj); //Overridden from Object class

}

* An anonymous function may be defined as a Lambda Expression (or function) (a function with no name and an identifier).
* Lambda Expressions are defined precisely where they are required, often as a parameter to another function.
* Lambda Expressions, on the other hand, express instances of Functional Interfaces from a different viewpoint.
* Lambda Expressions implement functional interfaces by implementing the single abstract function provided in the functional interface.
* A basic example of the Lambda Expression is: (x,y) -> x+y

interface MyInterface

{

void abstract\_func(int x, int y);

default void default\_Fun()

{

System.out.println("This is default method");

}

}

class Main

{

public static void main(String args[])

{

//lambda expression

MyInterface fobj = (int x, int y)->System.out.println(x + y);

System.out.print("The result = ");

fobj.abstract\_func(5,5);

fobj.default\_Fun();

}

}

1. **foreach() method**

* In Java 8, the Java.lang interface now supports a “forEach” function.
* Iterable that can iterate over the collection’s items.
* The Iterable interface has a default method called “forEach.”
* Collection classes use it to iterate items, which extends the Iterable interface.
* You may send Lambda Expression as an argument to the “forEach” method,
* which accepts the Functional Interface as a single parameter.

Import java.util.ArrayList;

Import java.util.List;

public class Main {

public static void main(String[] args) {

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

subList.add("Carrot");

subList.add("Potato");

subList.add("Cauliflower");

subList.add("Ladyfinger");

subList.add("Tomato");

System.out.println("------------Vegetable List--------------");

subList.forEach(sub -> System.out.println(sub));

}

}

1. **Optional Class**

* The public final class “Optional” is used to handle NullPointerException in a Java program.
* Thus, optional reduces the number of null checks required to avoid a nullPointerException.
* Use-> You may use the Optional class to prevent the application from crashing and terminating unexpectedly.
* The Optional class has methods for checking the existence of a value for a given variable.

import java.util.Optional;

public class OptionalDemo {

public static void main(String[] args) {

String[] str = new String[10];

Optional<String> checkNull =Optional.ofNullable(str[5]);

if(checkNull.isPresent()) {

String word = str[5].toLowerCase();

System.out.println(str);

}

else {

System.out.println("string is null");

}

}

}

Output:

String is null

To verify whether the string is null in this application,

we utilize the Optional class’s “ofNullable” attribute. If it is, the user receives the relevant message.

1. **Default and static methods:**

* In Java 8, you may add non-abstract methods to interfaces, allowing you to create interfaces with method implementation.
* To construct interfaces with method implementation, use the Default and Static keywords.
* Lambda Expression functionality is mostly enabled through default approaches.
* You may extend the functionality of your libraries’ interfaces by using default methods.
* This ensures that the code created for previous versions is compatible with the newer interfaces (binary compatibility).

import java.util.Optional;

interface interface\_default {

default void default\_method(){

System.out.println("We are default method of interface");

}

}

class derived\_class implements interface\_default{

}

class Main{

public static void main(String[] args){

derived\_class obj1 = new derived\_class();

obj1.default\_method();

}

}

output: We are the default method of interface

* We have an interface called “interface default” with a default implementation of the function default method().
* Next, we create a class called “derived class” that implements the “interface default” interface.
* In this class, we haven’t implemented any interface functions.
* Then, in the main function, we construct a “derived class” object and invoke the interface’s “default method” without specifying it in the class.
* The usage of default and static methods in the interface is an example of this. If a class wishes to alter the default method, it may override it and give its own implementation.

1. **StreamAPI methods:**

* Java 8 introduced a powerful Stream API for processing sequences of elements in a functional style.
* filter(Predicate): Returns a stream consisting of elements that match the given predicate.
* The filter() method in the Java Stream API is used to select elements from a stream based on a specified predicate.
* It evaluates each element in the stream against the predicate, and if the predicate returns true for an element, that element is included in the resulting stream.
* If the predicate returns false, the element is excluded from the resulting stream.

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

Where:

* **T** is the type of the elements in the stream.
* **Predicate** is a functional interface representing a predicate (a boolean-valued function) that accepts an element of type T and returns a boolean.
* **predicate**: The predicate used to filter elements. Only elements for which this predicate returns true are included in the resulting stream.
* How filter() Works
* Predicate Evaluation: The filter() method iterates over each element in the stream.
* Predicate Application: For each element, it applies the provided predicate.
* Element Inclusion: If the predicate returns true for an element, that element is included in the resulting stream.
* Element Exclusion: If the predicate returns false, that element is excluded from the resulting stream.

public class StreamApi {

public static void main(String[] args) {

List<String> names = Arrays.asList("alice","bob","ane");

List<String> filteredName = names.stream()

.filter(name-> name.startsWith("a"))

.collect(Collectors.toList());

System.out.println(filteredName);

}

}

1. **map(Function):**

* Returns a stream consisting of the results of applying the given function to the elements.
* The map function in Java's Stream API is a powerful method that transforms each element of a stream into another object as per the mapping function provided.

**Function<? super T, ? extends R>**

Where this:

* Represents a function that takes an element of type T from the stream and returns an element of type R.
* Returns a new stream (Stream<R>) consisting of elements of type R, which are the result of applying the given mapper function to the elements of the original stream (Stream<T>).

|  |  |
| --- | --- |
| import java.util.Arrays;  import java.util.List;  import java.util.stream.Collectors;  class Person {  private String name;  private int age;  public Person(String name, int age) {  this.name = name;  this.age = age;  }  public String getName() {  return name;  }  public int getAge() {  return age;  }  } | public class Main {  public static void main(String[] args) {  List<Person> persons = Arrays.asList(  new Person("Alice", 30),  new Person("Bob", 25),  new Person("Charlie", 35) );  // Example: Extracting names using map function  List<String> names = persons.stream()  .map(Person ::getName)  .collect(Collectors.toList());  System.out.println(names);  // Output: [Alice, Bob, Charlie]  }  } |

* Stream Creation: persons.stream() creates a stream of Person objects from the List<Person>.
* Mapping with map: .map(Person :: getName) applies the getName() method reference on each Person object in the stream.
* Here, Person :: getName is a shorthand for person -> person.getName(). It extracts the name property of each Person object.
* Collecting Results: .collect(Collectors.toList()) collects the mapped names into a new List<String>.
* Common Use Cases:
* Data Transformation: Transforming objects from one form to another (e.g., extracting fields, converting types).
* Stream Element Modification: Applying functions to modify elements in a stream (e.g., calculating derived properties).

1. **flat map**

* The flatMap() method in Java Stream API is used to flatten a stream of collections or arrays into a single stream of elements.
* It's particularly useful when you have a stream of collections (e.g., lists) or arrays, and you want to process each element of these nested collections individually rather than as a whole collection.

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

* mapper: A function that takes an element of type T from the original stream and returns a stream of elements of type R.
* How flatMap() Works
* Function Application: The mapper function is applied to each element of the original stream.
* Flattening: Each resulting stream produced by the mapper function is flattened into a single stream.
* This means that elements from the nested streams are pulled out into a single flat stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class FlatMap {

public static void main(String[] args) {

List<List<String>> nestedlist = Arrays.asList( Arrays.asList("abc","efg","xyz"),

Arrays.asList("sankey","mickey","Aditya"),

Arrays.asList("Anu") );

List<String> flattendList = nestedlist.stream()

.flatMap(List :: stream)

.collect(Collectors.toList());

System.out.println(nestedlist);

System.out.println(flattendList);

}

}

* nestedList is a list containing three nested lists of strings.
* flatMap(List::stream) is applied to nestedList. Here, List::stream is a method reference that converts each List<String> in nestedList into a Stream<String>.
* flatMap(List::stream) then flattens these streams into a single Stream<String> containing all the elements from all nested lists.
* collect(Collectors.toList()) is used to collect all elements from the flattened stream into a new List<String>.
* Additional Notes
* Nested Streams: flatMap() is particularly useful when dealing with nested structures such as lists of lists or arrays of arrays.
* Mapping Function: The mapper function provided to flatMap() should return a stream for each element of the original stream.
* This allows for flexibility in how elements are transformed and flattened.
* Use Cases: Common use cases include flattening lists of lists, processing and flattening arrays, and dealing with optional values in streams.
* In summary, flatMap() in Java Stream API is a powerful tool for flattening nested collections or arrays into a single stream of elements, enabling concise and efficient data manipulation and transformation operations.
* How filter() and map() Work Together
* Filtering: The filter() method is applied first to filter elements based on a predicate.
* It evaluates each element and includes only those elements for which the predicate returns true.
* Mapping: The map() method is applied after filtering to transform each element in the stream into another object using a mapper function.
* The transformed elements are then collected into a new stream.

**stream() Method Overview**

* The stream() method in Java is a part of the Collection interface introduced in Java 8 as part of the Stream API.
* It converts a collection into a sequential Stream, which allows for functional-style operations on the elements of the collection.
* Streams provide a powerful and concise way to work with collections in Java, enabling operations such as filtering, mapping, reduction, and more.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("apple", "banana", "cherry", "date");

// Convert list to a sequential stream

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

// Perform operations on the stream (e.g., print each element)

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

}

}

**Arrays.stream():**

* The Arrays.stream() method in Java is used to create a stream from an array.
* This allows you to apply the powerful stream operations provided by the Stream API to the elements of the array.
* Here’s a detailed explanation of how Arrays.stream() works, along with an example:

import java.util.Arrays;

import java.util.stream.IntStream;

public class ArraysStreamExample {

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

// Convert array to IntStream

IntStream stream = Arrays.stream(array);

// Perform operations on the stream (e.g., print each element)

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

}

}

**Stream.generate()**

* The Stream.generate() method in the Java Stream API creates an infinite sequential unordered stream where each element is generated by the provided Supplier.
* This method is useful when you need to generate a stream of elements dynamically based on some logic provided by the Supplier.

**static <T> Stream<T> generate(Supplier<T> s)**

* T is the type of elements in the stream.
* s is a Supplier that provides elements for the stream.

import java.util.stream.Stream;

public class StreamGenerateExample {

public static void main(String[] args) {

// Generate a stream of 10 elements, each element is "element"

Stream.generate(() -> "element")

.limit(10)

.forEach(System.out::println);

}

}

**Stream.iterate()**

* The Stream.iterate() method in the Java Stream API creates an infinite sequential ordered stream produced by iterative application of a function to an initial element.
* It generates a stream where each element is computed based on the previous element,starting from an initial seed value.

**static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)**

* **T** is the type of elements in the stream.
* seed is the initial element.
* f is a function (UnaryOperator) to apply to the previous element to produce the next element.

import java.util.stream.Stream;

public class StreamIterateExample {

public static void main(String[] args) {

// Generate a stream of 10 even numbers starting from 0

Stream.iterate(0, n -> n + 2)

.limit(10)

.forEach(System.out::println);

}

}

**Key differences:**

Generation Logic:

* Stream.generate() uses a Supplier to generate elements independently of the previous element,
* while Stream.iterate() uses a function (UnaryOperator) that generates each element based on the previous element.

Order:

* Both methods produce an ordered stream, but Stream.generate() is unordered by nature of its independent generation,
* whereas Stream.iterate() produces an ordered stream based on the iterative application of the function.

**Common Use Cases:**

* **Stream.generate():** Use when you need a stream of elements that are independently generated based on a supplier, such as generating random numbers or constant values.
* **Stream.iterate():** Use when you need a stream of elements that are generated based on a function applied iteratively, such as generating a sequence of numbers or elements in a predictable order.

In summary, **Stream.generate() and Stream.iterate()** are useful methods in the Java Stream API for generating streams of elements dynamically.

They provide flexibility in generating both ordered and unordered streams based on different generation strategies.

**Sorted():**

* The sorted() method in the Java Stream API returns a stream with elements sorted according to their natural order.
* For elements that implement the Comparable interface (like String, Integer, etc.), natural ordering is used.
* For custom objects, you need to provide a comparator.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class SortedExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("banana", "apple", "pear", "orange");

// Sort the stream in natural order

List<String> sortedList = list.stream()

.sorted()

.collect(Collectors.toList());

System.out.println("Original List: " + list);

System.out.println("Sorted List: " + sortedList);

}

}

**distinct():**

* The distinct() method in the Java Stream API returns a stream with unique elements, according to the equals() method of the elements.
* It removes duplicate elements from the stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class DistinctExample {

public static void main(String[] args) {

List<String> list = Arrays.asList("apple", "banana", "apple", "orange");

// Get distinct elements from the stream

List<String> distinctList = list.stream()

.distinct()

.collect(Collectors.toList());

System.out.println("Original List: " + list);

System.out.println("Distinct List: " + distinctList);

}

}

**limit():**

* The limit() method in the Java Stream API truncates the stream to be no longer than the specified size.
* It returns a stream consisting of the first n elements of the original stream.

import java.util.stream.Stream;

public class LimitExample {

public static void main(String[] args) {

// Generate a stream of numbers and limit to 3 elements

Stream<Integer> stream = Stream.iterate(1, n -> n + 1);

stream.limit(3)

.forEach(System.out::println);

}

}

**skip()**

* The skip() method in the Java Stream API skips the first n elements of the stream and returns a new stream with the remaining elements.

import java.util.stream.Stream;

public class SkipExample {

public static void main(String[] args) {

// Generate a stream of numbers and skip the first 2 elements

Stream<Integer> stream = Stream.iterate(1, n -> n + 1);

stream.skip(2)

.forEach(System.out::println);

}

}

**Fail fast and fail safe:**

* The Java Collection supports two types of iterators-> Fail Fast and Fail Safe(useful in exception handling).
* The Fail fast iterator aborts the operation as soon it exposes failures and stops the entire operation.
* Comparatively, Fail Safe iterator doesn't abort the operation in case of a failure. Instead, it tries to avoid failures as much as possible.
* The Concurrent modification in Java is to modify an object concurrently while another task is running over it. In simple terms, concurrent modification is the process of modifying objects while another thread is running over them. It will change the structure of the data collection, either by removing, adding, or updating the value of the elements in the collection.
* Not all iterators support this behaviour; implementation of some iterator may throw ConcurrentModificationException.

**Fail Fast**

* The Fail Fast iterators immediately throw ConcurrentModificationException in case of structural modification of the collection.
* Structural modification means adding, removing, updating the value of an element in a data collection while another thread is iterating over that collection.
* Some examples of Fail Fast iterator are iterator on ArrayList, HashMap collection classes.

**Fail Safe Iterator**

* The Fail-Safe iterators are just opposite to Fail Fast iterators; unlike them,
* A fail-safe iterator does not throw any exceptions unless it can handle if the collection is modified during the iteration process.
* This can be done because they operate on the copy of the collection object instead of the original object.
* The structural changes performed on the original collection ignored by them and affect the copied collection, not the original collection. So, the original collection will be kept structurally unchanged.

**Diff between fail fast and fail safe:**

|  |  |
| --- | --- |
| **Fail fast iterator** | **Fail safe iterator** |
| * It throws a ConcurrentModificationException in modifying the object during the iteration process. * No clone object is created during the iteration process. * It requires low memory during the process. * It is fast. * HashMap, ArrayList, Vector, HashSet, etc | * It does not throw Exception. * A copy or clone object is created during the iteration process. * It requires more memory during the process. * It is slightly slower than Fail Fast. * CopyOnWriteArrayList, ConcurrentHashMap, etc. |