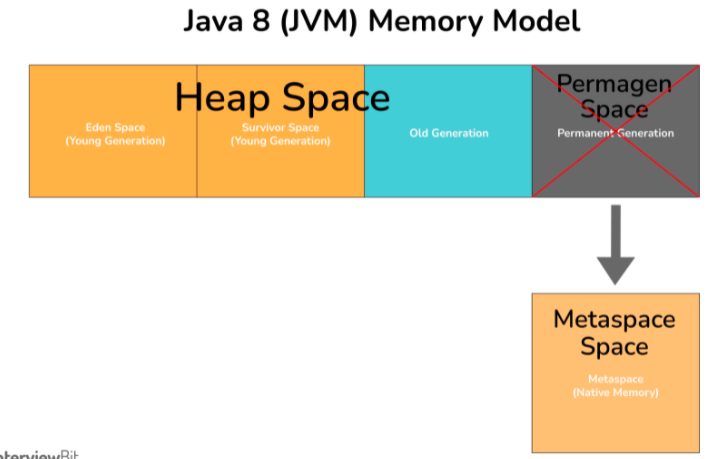
# What is MetaSpace? How does it differ from PermGen?

JVM

**PremGen:** MetaData information of classes was stored in PremGen (Permanent-Generation) memory type before Java 8. PremGen is fixed in size and cannot be dynamically resized. It was a contiguous Java Heap Memory.

**MetaSpace:** Java 8 stores the MetaData of classes in native memory called 'MetaSpace'. It is not a contiguous Heap Memory and hence can be grown dynamically which helps to overcome the size constraints. This improves the garbage collection, auto-tuning, and de-allocation of metadata.

# What are functional or SAM interfaces?

Functional Interfaces are an interface with only one abstract method. Due to which it is also known as the Single Abstract Method (SAM) interface. It is known as a functional interface because it wraps a function as an interface or in other words a function is represented by a single abstract method of the interface.

Functional interfaces can have any number of default, static, and overridden methods. For declaring Functional Interfaces @FunctionalInterface annotation is optional to use. If this annotation is used for interfaces with more than one abstract method, it will generate a compiler error.

@FunctionalInterface // Annotation is optional

**public** interface **Foo**() {

// Default Method - Optional can be 0 or more

**public** **default** String **HelloWorld**() {

**return** "Hello World";

}

// Static Method - Optional can be 0 or more

**public** **static** String **CustomMessage**(String msg) {

**return** msg;

}

// Single Abstract Method

**public** **void** **bar**();

}

**public** **class** **FooImplementation** **implements** **Foo** {

// Default Method - Optional to Override

@Override

**public** **default** String **HelloWorld**() {

**return** "Hello Java 8";

}

// Method Override

@Override

**public** **void** **bar**() {

System.out.println(“Hello World”);

}

}

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

FooImplementation fi = **new** FooImplementation();

System.out.println(fi.HelloWorld());

System.out.println(fi.CustomMessage(“Hi”));

fi.bar();

}

# Can a functional interface extend/inherit another interface?

A functional interface cannot extend another interface with abstract methods as it will void the rule of

one abstract method per functional interface.

It can extend other interfaces which do not have any abstract method and only have the default, static,

another class is overridden, and normal methods.

# **What is the default method, and why is it required?**

A method in the interface that has a predefined body is known as the default method. It uses the keyword default. default methods were introduced in Java 8 to have 'Backward Compatibility in case JDK modifies any interfaces. In case a new abstract method is added to the interface, all classes implementing the interface will break and will have to implement the new method. With default methods, there will not be any impact on the interface implementing classes. default methods can be overridden if needed in the implementation. Also, it does not qualify as synchronized or final.

@FunctionalInterface // Annotation is optional

**public** interface **Foo**() {

// Default Method - Optional can be 0 or more

**public** **default** String **HelloWorld**() {

**return** "Hello World";

}

// Single Abstract Method

**public** **void** **bar**();

}

# What are static methods in Interfaces?

Static methods, which contains method implementation is owned by the interface and is invoked using the name of the interface, it is suitable for defining the utility methods and cannot be overridden

# What are some standard Java pre-defined functional interfaces?

Some of the famous pre-defined functional interfaces from previous Java versions are Runnable, Callable, Comparator, and Comparable. While Java 8 introduces functional interfaces like Supplier, Consumer, Predicate, etc. Please refer to the java.util.function doc for other predefined functional interfaces and its description introduced in Java 8.

**Runnable:** use to execute the instances of a class over another thread with no arguments and no return value.

**Callable:** use to execute the instances of a class over another thread with no arguments and it either returns a value or throws an exception.

**Comparator:** use to sort different objects in a user-defined order

**Comparable:** use to sort objects in the natural sort order

# What are the various categories of pre-defined function interfaces?

**Function:** To transform arguments in returnable value.

**Predicate:** To perform a test and return a Boolean value.

**Consumer:** Accept arguments but do not return any values.

**Supplier:** Do not accept any arguments but return a value.

**Operator:** Perform a reduction type operation that accepts the same input types.

# Some of the Functional Interfaces in the Standard Library

There are a lot of functional interfaces in the *java.util.function* package. The more common ones include

* Function – it takes one argument and returns a result
* Consumer – it takes one argument and returns no result (represents a side effect)
* Supplier – it takes no arguments and returns a result
* Predicate – it takes one argument and returns a boolean
* BiFunction – it takes two arguments and returns a result
* BinaryOperator – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types.
* UnaryOperator – it is similar to a Function, taking a single argument and returning a result of the same type

# What is the lambda expression in Java and How does a lambda expression relate to a functional interface?

Lambda expression is a type of function without a name. It may or may not have results and parameters. It is known as an anonymous function as it does not have type information by itself. It is executed on-demand. It is beneficial in iterating, filtering, and extracting data from a collection.

As lambda expressions are similar to anonymous functions, they can only be applied to the single abstract method of Functional Interface. It will infer the return type, type, and several arguments from the signature of the abstract method of functional interface.

# What is the basic structure/syntax of a lambda expression?

FunctionalInterface fi = (String name) -> {

System.out.println("Hello "+name);

**return** "Hello "+name;

}

Lambda expression can be divided into three distinct parts as below:

1. List of Arguments/Params:

(String name)

A list of params is passed in () round brackets. It can have zero or more params. Declaring the type of parameter is optional and can be inferred for the context.

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

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

a->{System.out.println(a)}; // Will print value of number a (here is a single parameter, if its type is inferred, it is not mandatory to use parentheses)

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

(a,b)->{a+b};//two argument, will return sum of these two numbers(not declare type of arguments as it can be inferred from context.)

(int a,b)->{a+b};//Compilation error : can not declare one argument’s type and do not declare type for other argument.

2. Arrow Token:

->   
Arrow token is known as the lambda arrow operator. It is used to separate the parameters from the body, or it points the list of arguments to the body.

3. Expression/Body:

{

System.out.println("Hello "+name);

**return** "Hello "+name;

}

A body can have expressions or statements. {} curly braces are only required when there is more than one line. In one statement, the return type is the same as the return type of the statement. In other cases, the return type is either inferred by the return keyword or void if nothing is returned.

# What are the types and common ways to use lambda expressions?

A lambda expression does not have any specific type by itself. A lambda expression receives type once it is assigned to a functional interface. That same lambda expression can be assigned to different functional interface types and can have a different type.

For eg consider expression s -> s.isEmpty() :

Predicate<String> stringPredicate = s -> s.isEmpty();   
Predicate<List> listPredicate = s -> s.isEmpty();  
Function<String, Boolean> func = s -> s.isEmpty();  
Consumer<String> stringConsumer = s -> s.isEmpty();

**Common ways to use the expression**

Assignment to a functional Interface —> Predicate<String> stringPredicate = s -> s.isEmpty();  
Can be passed as a parameter that has a functional type —> stream.filter(s -> s.isEmpty())  
Returning it from a function —> return s -> s.isEmpty()  
Casting it to a functional type —> (Predicate<String>) s -> s.isEmpty()

# what is Method Reference?

Method reference is a compact way of referring to a method of functional interface. It is used to refer to a method without invoking it. :: (double colon) is used for describing the method reference. The syntax is class::methodName

For e.g.:

Integer::parseInt(str) \\ method reference

str -> Integer.ParseInt(str); \\ equivalent lambda

A method reference can be identified by a double colon separating a class or object name and the name of the method. It has different variations

String::**new**; \\ constructor ref  
String::valueOf; \\ Static method reference:  
str::toString; \\ Bound instance method reference:  
String::toString; \\ Unbound instance method reference:

# What is an Optional class?

Optional is a container type which may or may not contain value i.e. zero(null) or one(not-null) value. It is part of java.util package. There are pre-defined methods like isPresent(), which returns true if the value is present or else false and the method get(), which will return the value if it is present.

It encapsulates optional values, i.e., null or not-null values, which helps in avoiding null checks, which results in better, readable, and robust code It acts as a wrapper around the object and returns an object instead of a value, which can be used to avoid run-time NullPointerExceptions.

**static** Optional<String> **changeCase**(String word) {

**if** (name != **null** && word.startsWith("A")) {

**return** Optional.of(word.toUpperCase());

}

**else** {

**return** Optional.ofNullable(word); // someString can be null

}

}

create an *Optional* object with the static method *of()*:

the argument passed to the *of()* method can't be *null.* Otherwise, we'll get a *NullPointerException*:

String name = "test";

Optional<String> opt = Optional.of(name);

Optional<String> empty = Optional.empty(); // create empty object

in case we expect some *null* values, we can use the *ofNullable()* method:

String name = "baeldung";

Optional<String> opt = Optional.ofNullable(name);

opt.isPresent()

if we pass in a null reference, it doesn't throw an exception but rather returns an empty Optional object:

String name = **null**;

Optional<String> opt = Optional.ofNullable(name);

opt.isPresent()

isPresent() : check there is a value in it or not

orElse(): method returns the wrapped value if it's present, and its argument otherwise:

String nullName = **null**;

String name = Optional.ofNullable(nullName).orElse("john");

orElseThrow() method : Instead of returning a default value when the wrapped value is not present, it throws an exception:

String nullName = **null**;

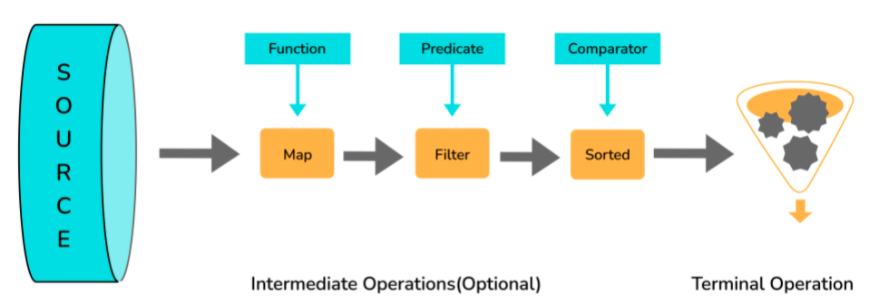
String name = Optional.ofNullable(nullName)

.orElseThrow( IllegalArgumentException::**new**);

# What are the main components of a Stream?

Components of the stream are:

* A data source
* Set of Intermediate Operations to process the data source
* Single Terminal Operation that produces the result



# **What Is Stream Pipelining in Java 8**

Stream pipelining is the concept of chaining operations together. We do this by splitting the operations that can happen on a stream into two categories: intermediate operations and terminal operations.

Each intermediate operation returns an instance of Stream itself when it runs. Therefore, we can set up an arbitrary number of intermediate operations to process data, forming a processing pipeline.

There must then be a terminal operation which returns a final value and terminates the pipeline.

# Difference between Intermediate and terminal operations in Stream?

Intermediate operations are lazy in nature and do not get executed immediately. Terminal operations are not lazy, they are executed as soon as encountered. Intermediate operation is memorized and is called when terminal operation is called.

All Intermediate operations return stream as it just transforms stream into another and terminal operation don’t.

# What are the most commonly used Intermediate operations?

**Filter(Predicate<T>)** - Allows selective processing of Stream elements. It returns elements that are satisfying the supplied condition by the predicate.

**map(Funtion<T, R>)** - Returns a new Stream, transforming each of the elements by applying the supplied mapper function.= sorted() - Sorts the input elements and then passes them to the next stage.

**distinct()** - Only pass on elements to the next stage, not passed yet.

**limit(long maxsize)** - Limit the stream size to maxsize.

Stream.of(0,1,2,3,4,5,6,7,8)

.limit(6)

/\*limit is set to 6, hence it will print the

numbers starting from 0 to 5

\*/

**skip(long start)** - Skip the initial elements till the start.

Stream.of(0,1,2,3,4,5,6,7,8)

.skip(6)

/\*

It will skip till 6th index. Hence 7th, 8th and 9th

index elements will be printed

\*/

**peek(Consumer)** - Apply a consumer without modification to the stream.

**flatMap(mapper)** - Transform each element to a stream of its constituent elements and flatten all the streams into a single stream.

# What is the stateful intermediate operation? Give some examples of stateful intermediate operations.

To complete some of the intermediate operations, some state is to be maintained, and such intermediate operations are called stateful intermediate operations. Parallel execution of these types of operations is complex.

For Eg: sorted() , distinct() , limit() , skip() etc.

Sending data elements to further steps in the pipeline stops till all the data is sorted for sorted() and stream data elements are stored in temporary data structures.

# What is the most common type of Terminal operations?

* collect() - Collects single result from all elements of the stream sequence.
* reduce() - Produces a single result from all elements of the stream sequence
  + count() - Returns the number of elements on the stream.
  + min() - Returns the min element from the stream.
  + max() - Returns the max element from the stream.
* Search/Query operations
  + anyMatch() , noneMatch() , allMatch() , ... - Short-circuiting operations.
  + Takes a Predicate as input for the match condition.
  + Stream processing will be stopped, as and when the result can be determined.
* Iterative operations
  + forEach() - Useful to do something with each of the Stream elements. It accepts a consumer.
  + forEachOrdered() - It is helpful to maintain order in parallel streams.

# **What is the use of the peek() method in Java 8?**

The peek() method is a part of the stream class in Java 8, which is used to see actions performed through a stream pipeline. Peeking can be done at every step to print messages about the code being executed onto the console.

Peeking has a wide amount of usage when efficiency is a requirement, when debugging code with the lambda expression, or when performing stream processing

# How are Collections different from Stream?

Collections are the source for the Stream. Java 8 collection API is enhanced with the default methods returning Stream<T> from the collections.

| **Collections** | **Streams** |
| --- | --- |
| Data structure holds all the data elements | No data is stored. Have the capacity to process an infinite number of elements on demand |
| External Iteration | Internal Iteration |
| Can be processed any number of times | Traversed only once |
| Elements are easy to access | No direct way of accessing specific elements |
| Is a data store | Is an API to process the data |

# What is the feature of the new Date and Time API in Java 8?

* Immutable classes and Thread-safe
* Timezone support
* Fluent methods for object creation and arithmetic
* Addresses I18N issue for earlier APIs
* Influenced by popular joda-time package
* All packages are based on the ISO-8601 calendar system

# Define Nashorn in Java 8

Nashorn is a JavaScript processing engine that is bundled with Java 8. It provides better compliance with ECMA (European Computer Manufacturers Association) normalized JavaScript specifications and better performance at run-time than older versions.

# What is the use of JJS in Java 8?

As part of Java 8, JJS is a command-line tool that helps to execute the JavaScript code in the console. Below is the example of CLI commands:

JAVA>jjs  
jjs> print("Hello, Java 8 - I am the new JJS!")  
Hello, Java 8 - I am the new JJS!  
jjs> quit()  
>>

# Explain with example, LocalDate, LocalTime, and LocalDateTime APIs.

**LocalDate**

* Date with no time component
* Default format - yyyy-MM-dd (2020-02-20)
* LocalDate today = LocalDate.now(); // gives today’s date
* LocalDate aDate = LocalDate.of(2011, 12, 30); //(year, month, date)

**LocalTime**

* Time with no date with nanosecond precision
* Default format - hh:mm:ss:zzz (12:06:03.015) nanosecond is optional
* LocalTime now = LocalTime.now(); // gives time now
* LocalTime aTime2 = LocalTime.of(18, 20, 30); // (hours, min, sec)

**LocalDateTime**

* Holds both Date and Time
* Default format - yyyy-MM-dd-HH-mm-ss.zzz (2020-02-20T12:06:03.015)
* LocalDateTime timestamp = LocalDateTime.now(); // gives timestamp now
* //(year, month, date, hours, min, sec)
* LocalDateTime dt1 = LocalDateTime.of(2011, 12, 30, 18, 20, 30);

# What Is the Difference Between *Map* and *flatMap* Stream Operation

There is a difference in signature between map and flatMap. Generally speaking, a map operation wraps its return value inside its ordinal type, while flatMap does not

For example, in Optional, a map operation would return Optional<String> type, while flatMap would return String type.

- Both map and flatMap are intermediate stream operations that receive a function and apply this function to all the elements of a stream.

The difference is that for the map, this function returns a value,   
but for flatMap, this function returns a stream. The flatMap operation “flattens” the streams into one.

Map Stream operation gives one output value per input value

Map Stream operation is generally used for simple operation on Stream

List<String> Names = Arrays.asList("Saket", "Trevor", "Franklin", "Michael");

List<String> UpperCase = Names.stream().map(String::toUpperCase).collect(Collectors.toList());

// Changed the characters into upper case after converting it into Stream

whereas flatMap Stream operation gives zero or more output value per input value.

flatMap Stream operation is used for more complex Stream operation.

List<List<String>> Names = Arrays.asList(Arrays.asList("Saket", "Trevor"), Arrays.asList("John", "Michael"),

Arrays.asList("Shawn", "Franklin"), Arrays.asList("Johnty", "Sean"));

/\* Created a “List of List of type String” i.e. List<List<String>>

Stored names into the list

\*/

List<String> Start = Names.stream().flatMap(FirstName -> FirstName.stream()).filter(s -> s.startsWith("S"))

.collect(Collectors.toList());

/\* Converted it into Stream and filtered

out the names which start with 'S'

\*/

# Difference between Stream’s findFirst() and findAny()

findFirst will always return the first element from the stream whereas findAny is allowed to choose any element from the stream.  
findFirst has deterministic behavior whereas findAny is nondeterministic behavior.

| **findFirst()** | **findAny()** |
| --- | --- |
| Returns the first element in the Stream | Return any element from the Stream |
| Deterministic in nature | Non-deterministic in nature |

# What is the difference between Java 8 Internal and External Iteration?

**Answer:** **The difference between Internal and External Iteration is enlisted below.**

| **Internal Iteration** | **External Iteration** |
| --- | --- |
| It was introduced in Java 8 (JDK-8). | It was introduced and practiced in the previous version of Java (JDK-7, JDK-6 and so on). |
| It iterates internally on the aggregated objects such as Collection. | It iterates externally on the aggregated objects. |
| It supports the Functional programming style. | It supports the OOPS programming style. |
| Internal Iterator is passive. | External Iterator is active. |
| It is less erroneous and requires less coding. | It requires little more coding and it is more error-prone. |

# **Differentiate between Spliterator and a regular iterator in Java 8.**

|  |  |
| --- | --- |
| **Spliterator** | **Iterator** |
| Introduced with Java 8 | Present since Java 1.2 |
| Used in Stream API | Used in Collection API |
| Helps in the iteration of streams in a parallel and sequential order | Iterates collections in a sequential order only |
| Example: tryAdvance() | Examples: next() and hasNext() |

# **Explain StringJoiner Class in Java 8? How can we achieve joining multiple Strings using StringJoiner Class?**

StringJoiner can join multiple strings separated by delimiters along with providing prefix and suffix to them.

StringJoiner stj = new StringJoiner(",");

// Separated the elements with a comma in between.

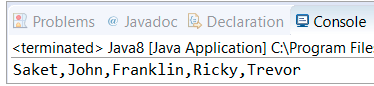
stj.add("Saket");

stj.add("John");

stj.add("Franklin");

stj.add("Ricky");

stj.add("Trevor");



# **Can you briefly explain the working of the random keyword in Java 8?**

The random keyword, as the name suggests, is used to generate random values for computations and operations in Java 8.

The following piece of code is used to print out 20 random numbers using the forEach loop:

Random random = new Random();

random.ints().limit(20).forEach(System.out::println);

# Exceptions in Java 8 Lambda Expressions

## **Overview**

In Java 8, Lambda Expressions started to facilitate functional programming by providing a concise way to express behavior. However, the Functional Interfaces provided by the JDK don't deal with exceptions very well – and the code becomes verbose and cumbersome when it comes to handling them.

## **Handling Unchecked Exceptions**

First, let's understand the problem with an example.

We have a List<Integer> and we want to divide a constant, say 50 with every element of this list and print the results:

List<Integer> integers = Arrays.asList(3, 9, 7, 6, 10, 20);

integers.forEach(i -> System.out.println(50 / i));

This expression works but there's one problem. If any of the elements in the list is 0, then we get an ArithmeticException: / by zero. Let's fix that by using a traditional try-catch block such that we log any such exception and continue execution for next elements:

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(i -> {

**try** {

System.out.println(50 / i);

} **catch** (ArithmeticException e) {

System.err.println(

"Arithmetic Exception occured : " + e.getMessage());

}

});

The use of try-catch solves the problem, but the conciseness of a Lambda Expression is lost and it's no longer a small function as it's supposed to be.

To deal with this problem, we can write **a lambda wrapper for the lambda function**. Let's look at the code to see how it works:

**static** Consumer<Integer> **lambdaWrapper**(Consumer<Integer> consumer) {

**return** i -> {

**try** {

consumer.accept(i);

} **catch** (ArithmeticException e) {

System.err.println(

"Arithmetic Exception occured : " + e.getMessage());

}

};

}

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(lambdaWrapper(i -> System.out.println(50 / i)));

At first, we wrote a wrapper method that will be responsible for handling the exception and then passed the lambda expression as a parameter to this method.

The wrapper method works as expected but, you may argue that it's basically removing the try-catch block from lambda expression and moving it to another method and it doesn't reduce the actual number of lines of code being written.

This is true in this case where the wrapper is specific to a particular use case but we can make use of generics to improve this method and use it for a variety of other scenarios:

## **Handling Checked Exceptions**

Let's modify the example from the previous section and instead of printing to the console, let's write to a file.

**static** **void** **writeToFile**(Integer integer) **throws** IOException {

// logic to write to file which throws IOException

}

Note that the above method may throw the IOException.

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(i -> writeToFile(i));

On compilation, we get the error:

java.lang.Error: Unresolved compilation problem: Unhandled exception type IOException

Because **IOException is a checked exception, we must handle it explicitly**. We have two options.

First, we may simply throw the exception outside of our method and take care of it somewhere else.

Alternatively, we can handle it inside the method that uses a lambda expression.

Let's explore both of the options.

### **Throwing Checked Exception from Lambda Expressions**

Let's see what happens when we declare the IOException on the main method:

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

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(i -> writeToFile(i));

}

Still, **we get the same error of unhandled IOException during the compilation**.

java.lang.Error: Unresolved compilation problem: Unhandled exception type IOException

This is because lambda expressions are similar to [Anonymous Inner Classes](https://www.baeldung.com/java-anonymous-classes).

In our case, **writeToFile method is the implementation of** [**Consumer<Integer>**](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/class-use/Consumer.html) **functional interface**

Let's take a look at the *Consumer*‘s definition:

@FunctionalInterface

**public** **interface** **Consumer**<**T**> {

**void** **accept**(T t);

}

As we can see *accept* method doesn't declare any checked exception. This is why *writeToFile* isn't allowed to throw the *IOException.*

The most straightforward way would be to use a *try-catch* block, wrap the checked exception into an unchecked exception and rethrow it:

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(i -> {

**try** {

writeToFile(i);

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

});

This gets the code to compile and run. However, this approach introduces the same issue we already discussed in the previous section – it's verbose and cumbersome.

We can get better than that.

Let's create a custom functional interface with a single *accept* method that throws an exception.

@FunctionalInterface

**public** **interface** **ThrowingConsumer**<**T**, **E** **extends** **Exception**> {

**void** **accept**(T t) **throws** E;

}

And now, let's implement a wrapper method that's able to rethrow the exception:

**static** <T> Consumer<T> **throwingConsumerWrapper**(

ThrowingConsumer<T, Exception> throwingConsumer) {

**return** i -> {

**try** {

throwingConsumer.accept(i);

} **catch** (Exception ex) {

**throw** **new** RuntimeException(ex);

}

};

}

Finally, we're able to simplify the way we use the *writeToFile* method:

### **Handling a Checked Exception in Lambda Expression**

In this final section, we'll modify the wrapper to handle checked exceptions.

Since our ThrowingConsumer interface uses generics, we can easily handle any specific exception.

**static** <T, E extends Exception> Consumer<T> **handlingConsumerWrapper**(

ThrowingConsumer<T, E> throwingConsumer, Class<E> exceptionClass) {

**return** i -> {

**try** {

throwingConsumer.accept(i);

} **catch** (Exception ex) {

**try** {

E exCast = exceptionClass.cast(ex);

System.err.println(

"Exception occured : " + exCast.getMessage());

} **catch** (ClassCastException ccEx) {

**throw** **new** RuntimeException(ex);

}

}

};

}

Let's see how to use it in practice:

List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);

integers.forEach(handlingConsumerWrapper(

i -> writeToFile(i), IOException.class));

Note, that the above code **handles only IOException, whereas any other kind of exception is rethrown as a RuntimeException**.

## **Conclusion**

In this article, we showed how to handle a specific exception in lambda expression without losing the conciseness with the help of wrapper methods. We also learned how to write throwing alternatives for the Functional Interfaces present in JDK to either throw or handle a checked exception.

# Lamda Programming question

Given a list of employees, you need to filter all the employee whose age is greater than 20 and print the employee names.

List<String> employeeFilteredList = employeeList.stream()

.filter(e->e.getAge()>20)

.map(Employee::getName)

.collect(Collectors.toList());

Given the list of employees, count number of employees with age 25?

List<Employee> employeeList = createEmployeeList();

long count = employeeList.stream()

.filter(e->e.getAge()>25)

.count();

System.out.println("Number of employees with age 25 are : "+count);

Given the list of employees, find the employee with name “Mary”

List<Employee> employeeList = createEmployeeList();

Optional<Employee> e1 = employeeList.stream()

.filter(e->e.getName().equalsIgnoreCase("Mary")).findAny();

if(e1.isPresent())

System.out.println(e1.get());

Given a list of employee, find maximum age of employee?

List<Employee> employeeList = createEmployeeList();

OptionalInt max = employeeList.stream().

mapToInt(Employee::getAge).max();

if(max.isPresent())

System.out.println("Maximum age of Employee: "+max.getAsInt());

Given a list of employees, sort all the employee on the basis of age?

List<Employee> employeeList = createEmployeeList();

employeeList.sort((e1,e2)->e1.getAge()-e2.getAge());

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

Join the all employee names with “,”

List<Employee> employeeList = createEmployeeList();

List<String> employeeNames = employeeList

.stream()

.map(Employee::getName)

.collect(Collectors.toList());

String employeeNamesStr = String.join(",", employeeNames);

System.out.println("Employees are: "+employeeNamesStr);

Given the list of employee, group them by employee name?

List<Employee> employeeList = createEmployeeList();

Map<String, List<Employee>> map = employeeList.stream()

.collect(Collectors.groupingBy(Employee::getName));

map.forEach((name,employeeListTemp)->System.out.println("Name: "+name+" ==>"+employeeListTemp));

Given the list of numbers, remove the duplicate elements from the list.

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

List<Integer> listWithDuplicates = Arrays.asList(arr);

Set<Integer> setWithoutDups = listWithDuplicates.stream().collect(Collectors.toSet());

setWithoutDups.forEach((i)->System.out.print(" "+i));

or

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

List<Integer> listWithDuplicates = Arrays.asList(arr);

List<Integer> listWithoutDups = listWithDuplicates.stream().distinct().collect(Collectors.toList());

listWithoutDups.forEach((i)->System.out.print(" "+i));

Given a list of numbers, square them and filter the numbers which are greater 10000 and then find average of them

Integer[] arr=new Integer[]{100,24,13,44,114,200,40,112};

List<Integer> list = Arrays.asList(arr);

OptionalDouble average = list.stream()

.mapToInt(n->n\*n)

.filter(n->n>10000)

.average();

if(average.isPresent())

System.out.println(average.getAsDouble());

**Write a program to print 5 random numbers using forEach**

Random random = new Random();

random.ints().limit(5).forEach(System.out::println);

/\* limit is set to 5 which means only 5 numbers will be printed

with the help of terminal operation forEach

\*/

**Write a program to print 5 random numbers in sorted order using forEach**

Random random = new Random();

random.ints().limit(5).sorted().forEach(System.out::println);

/\* sorted() method is used to sort the output after

terminal operation forEach

\*/

**get the sum of all numbers present in a list**

list.stream().mapToInt(i -> i).sum();

**What is the easiest way to print the sum of all of the numbers present in a list using Java 8**

List<Integer> numbers = Arrays.asList(5, 4, 10, 12, 87, 33, 75);

IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();

System.out.println("Sum of all numbers : " + stats.getSum());

**program to find the lowest and highest number of a Stream?**

Integer highest = Stream.of(1, 2, 3, 77, 6, 5)

.max(Comparator.comparing(Integer::valueOf))

.get();

/\* We have used max() method with Comparator.comparing() method

to compare and find the highest number

\*/

Integer lowest = Stream.of(1, 2, 3, 77, 6, 5)

.min(Comparator.comparing(Integer::valueOf))

.get();

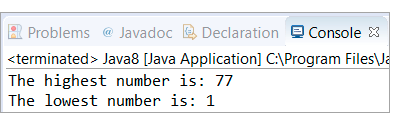
/\* We have used max() method with Comparator.comparing() method

to compare and find the highest number

\*/

System.out.println("The highest number is: " + highest);

System.out.println("The lowest number is: " + lowest);



**find the number of Strings in a list whose length is greater than 5**

long count = list.stream().filter(str -> str.length() > 5).count();

/\* Converted the list into Stream and filtering out

the Strings whose length more than 5

and counted the length

\*/

System.out.println("We have " + count + " strings with length greater than 5");

**program to concatenate two Streams?**

List<String> list1 = Arrays.asList("Java", "8");

List<String> list2 = Arrays.asList("explained", "through", "programs");

Stream<String> concatStream = Stream.concat(list1.stream(), list2.stream());

concatStream.forEach(str -> System.out.print(str + " "));