Java 8 Tips

* One of the major benefits of the functional interface is the possibility to use lambda expressions to instantiate them. We can instantiate an interface with an anonymous class but the code looks bulky.
* Java interface default methods are also referred to as **Defender Methods or Virtual extension** methods. Default method are implicitly public.
* Since static methods are allowed from Java 8, we can write a main method inside an interface and execute it as well
* java.util.function has been added with a bunch of functional interfaces to provide target types for lambda expressions and method references.
* We can use default and static keywords to create interfaces with method implementation.
* A ***default method*** cannot override a method from ***java.lang.Object***. The reasoning is very simple, it’s because Object is the base class for all the Java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class methods will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
* Java interface ***static method*** is similar to the default method except that we can’t override them in the implementation classes
* Java interface static method is visible to interface methods only
* Statistics collectors are introduced to calculate all statistics when stream processing is being done. Ex: **IntSummaryStatistics.**
* Either lambda expression or method reference does nothing but just another way call to an existing method. With method reference, it gains better readability.
* After having default and static methods inside the interface, we think about the need for an abstract class in Java. An interface and an abstract class are almost similar except that you can create a constructor in the abstract class whereas you cannot do this in the interface. Also in the interface, we cannot define a method without a default keyword.
* Do not use *ordinal()* to obtain the numeric representation of DayOfWeek and for Month. Use *getValue()* instead.
* ***Collectors*** is a final class that extends the Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.
* Local variable cannot be modified in lambda expression. Local variables referenced from a lambda expression must be final. Variables created outside a lambda expression cannot be modified inside it because lambda take a snapshot of the local variable. If the lambda doesn’t use the local variable we can change the value of it anywhere in our code, but if it uses we cant change the value anywhere.

**Lambda Expressions**

* Lambda Expression expresses an instance of the functional interface, in other words, you can say it provides a clear and concise way to represent a method of the functional interface using an expression.
* Lambda expression is an instance of a class that implements a functional interface.
* Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bound to an identifier. It can even be used as a parameter in a function.
* Syntax :(argument-list) -> {body}

1. **Argument-list**: It can be empty or non-empty as well.
2. **Arrow-token:** It is used to link the arguments-list and body of expression.
3. **Body:** It contains expressions and statements for lambda expression.

* **Rules of Lambda**
* A lambda expression can have zero, one, or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context.
* Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.
* The body of the lambda expressions can contain zero, one, or more statements.
* If the body of the lambda expression has 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 is more than one statement in the body then these must be enclosed in curly brackets.
* *Default methods of a functional interface cannot be accessed from within lambda expressions.*
* *If the lambda expression uses a parameter name that is the same as a variable name of the enclosing context, a compile error is generated.*
* If the body has one statement, curly brackets are not required, and the value of the expression (if any) is returned.
* If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must return with a return statement.
* The type of the parameters can be declared explicitly, or it can be inferred from the context.
* Lambda expressions don't contain information about which functional interfaces are implemented.
* **Use of Lambda**
  + Lambda Expressions enable you to encapsulate a single unit of behavior and pass it to other code. You can use lambda expressions if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
  + Less Coding
* **Summary - key points about lambda expressions**
* **Syntax Simplicity:** The syntax of a lambda expression is much more concise than using anonymous inner classes.
* **Functional Interfaces:** They are used primarily to define the implementation of an abstract method defined in a functional interface (an interface with exactly one abstract method).
* **Parameters and Body:** The structure of a lambda consists of a set of parameters, an arrow ->, and the body of the lambda.
* **Type Inference:** Java's type inference mechanism can often determine the type of the parameters, allowing you to skip declaring them.
* **Return Type:** If the body of the lambda consists of a single expression, the return type will be inferred, and the return keyword is not needed.
* **No Access Modifiers:** Lambda expressions do not have access modifiers or a throws clause.
* **Local Variable Access:** Lambdas can access final or effectively final local variables of the enclosing scope.
* **Parallel Execution Support:** They can be used to facilitate parallel processing, like in streams.
* **Enhanced Iteration:** They can be used with new iteration methods provided by the Iterable interface to make iterations over collections more concise.
* **Functional Programming Paradigm:** Lambda expressions bring elements of functional programming into Java and enable functional programming techniques and styles.
* **Immutable:** The variables used inside lambda expressions must be final or effectively final, making them immutable.
* **Interoperability with SAM Interfaces:** Lambda expressions can be used wherever a Single Abstract Method (SAM) interface is expected.
* **Target Typing:** The target type of a lambda expression is determined by the context in which the lambda is used.
* **Performance:** Lambda expressions are often more performant than anonymous classes since they are not compiled into separate classes.
* **Functional Method References:** They can be used in combination with method references to further simplify code where a method already exists to perform an operation.
* **No, this Reference:** Inside a lambda expression, this keyword refers to the enclosing instance, not the lambda itself.

**Functional Interface**

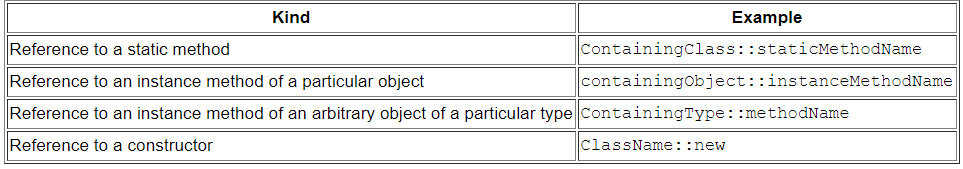
* Java provides an annotation **@FunctionalInterface**, which is used to declare an interface as a functional interface. The primary use of this annotation is for compiler-level errors when the interface violates the contracts of precisely one abstract method. Using the annotation ***@FunctionalInterface*** is optional.
* An interface that has ***only one abstract method*** is called a functional interface.
* It can have any number of ***default, static methods*** but can contain only one abstract method. It can contain any number of public methods inherited from the Object class.
* If a functional interface overrides one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
* Functional Interface is also known as ***Single Abstract Method Interfaces*** *or* ***SAM*** Interfaces.
* Runnable, Callable, Comparator, FileFilter, PathMatcher, EventHandler interfaces are some example of inbuild function interface.
* A functional interface can extend another interface only when it does not have any abstract method.
* An empty interface is not considered a functional interface
* **Function** is a functional interface; it takes an argument (object of type T) and returns an object (object of type R). The argument and output can be a different type. The function interface contains exactly one abstract method ***apply(T t)***. Note that it also contains default, static methods. Default methods are ***compose*** and ***andThen***. Static method is ***identity***.
* **compose**. Returns a composed function that first applies the before function to its input, and then applies this function to the result.
* **andThen** - Returns a composed function that first applies this function to its input, and then applies the after function to the result.
* *a.andThen(b) is equivalent to b.compose(a)*
* The difference between *compose and andThen* is the order they execute the functions. While the compose function executes the caller last and the parameter first, the andThen executes the caller first and the parameter last.
* **Function.identity()** returns a Function that always returns it’s input argument.
* **BiFunction** is a functional interface; it takes two arguments and returns an object. BiFunction interface contains exactly one abstract method ***apply(T arg0, U arg1)***. It has one default method ***andThen***.
* **Consumer** is a functional interface; it takes an argument and returns nothing. The consumer interface contains exactly one abstract method ***accept(T arg0)***. It has one default method ***andThen***.
* **BiConsumer:** It represents an operation that accepts two input arguments and returns no result. BiConsumer interface contains exactly one abstract method ***accept(T arg0, U arg1).***It has one default method ***andThen***.
* **Supplier** is a functional interface; it takes no arguments and returns a result. The supplier interface contains exactly one abstract method ***get(T t).*** It doesn’t have default and static methods.
* **Predicate** is a functional interface, which accepts an argument and returns a boolean. Usually, it is used to apply a filter for a collection of objects. Predicate interface contains exactly one abstract method ***test(T t).*** Note that it also contains default, static methods. Default methods are ***and, negate,*** and ***or***. Static method is ***isEqual***.
* **BiPredicate :** This interface represents a predicate that takes two arguments. Predicate interface contains one abstract method ***test(T t, U u)***.
* **UnaryOperator** - It represents a function that takes in one argument and operates on it. However what distinguishes it from a normal Function is that both its argument and return type are the same. If the input and return type are same we can use UnaryOperator instead of Function interface. It is a child of Function<T, T>
* **BinaryOperator** - It represents a binary operator which takes two operands and operates on them to produce a result. However, what distinguishes it from a normal BiFunciton is that both of its arguments and its return type are same.
* **DoubleToIntFunction** -It represents a function which takes in a double-valued argument and gives an int-valued result. The lambda expression assigned to an object of DoubleToIntFunction type is used to define its ***applyAsInt()*** which eventually applies the given operation on its only argument. It is similar to using an object of type ***Function<Double, Integer>.***

**Default & Static Methods**

* Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods. To provide backward compatibility this method is introduced.
* You can also define static methods inside the interface. Static methods are used to define utility methods.
* Just like regular interface methods, default methods are also implicitly ***public***.
* The ***stream()*** method defined in the Collection interface is a default method.
* We can override the default method also to provide a more specific implementation for the method. We can’t override ***the static methods*** defined in interface.
* Java interface default methods are also referred to as ***Defender Methods*** or **Virtual extension** methods.
* With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. This ambiguity can be resolved by two ways.
* First solution is to create its own method that overrides the default implementation in the implementing class.
* Second solution is to call the default method of the specified interface using super keyword in the overridden default method in the implementing class. *(Interface.super.defaultmethod)*
* The idea behind static interface methods is to provide a simple mechanism that allows us to increase the degree of cohesion of a design by putting together related methods in one single place without having to create an object.
* Furthermore, static methods in interfaces make it possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.
* Since static methods don't belong to a particular object, they are not available to the classes implementing the interface, and they have to be called by using the interface name preceding the method name.
* Since interface static methods by default not available to the implementation class, overriding concept is not applicable. Based on our requirement we can define exactly same method in the implementation class, it’s valid but not overriding.

**Method References**

* Method references help to point to methods by their names. A method reference is described using the **"::"** symbol.
* The arguments for the method type are inferred by JRE at the runtime through the context it is defined.
* A method reference can be used to point to the following types of methods
* Static Methods - ClassName::staticMethodName
* Instance Methods - Object::instanceMethodName
* Constructor using new keyword - ClassName::new



* ***Method reference is used to refer method of functional interface.*** It is a compact and easy form of lambda expression. Each time when you are using a lambda expression to just referring a method, you can replace your lambda expression with a method reference.
* ***The compiler turns the method references into lambdas in the background***.
* Before the release of Java 8, no methods were able to be passed as parameters. In such cases, if a method has to be passed, an object is required to be passed that contains the method. But, through the upgrade of the language, the methods are now being able to be passed as parameters.

**Optional Class**

* It is a public final class and used to deal with *NullPointerException* in Java application. You must import *java.util* package to use this class. It provides methods, which are used to check the presence of value for a particular variable.
* Advantages of Java 8 Optional
* Null checks are not required
* No more NullPointerException at run-time
* We can create Optional objects with the static factory methods *Optional.empty*, *Optional.of*, and *Optional.ofNullable*
* The *Optional.ofNullable()* method returns a Non-empty Optional if a value present in the given object. Otherwise, returns empty Optional. *Optional.empty()* method is useful to create an empty Optional object.
* The Optional class supports many methods such as ***map, flatMap,*** and ***filter***, which are conceptually similar to the methods of a stream
* ***isPresent()*** - Returns true if a value is present; otherwise, returns false.
* An ***empty()*** static method returns an empty Optional instance. No value is present for this Optional.
* ***ifPresent()*** If a value is present, invoke the specified consumer with the value, otherwise, do nothing.
* The ***ifPresentOrElse(Consumer, Runnable)*** method helps us to perform the specified Consumer action at the value of this Optional object. If a value is not present in this Optional, then this method performs the given empty-based Runnable emptyAction, passed as the second parameter.
* ***orElse()*** Return the value if present otherwise returns another.
* ***orElseGet()*** - Returns the value if present; otherwise, returns the one provided by the given Supplier.
* ***orElseThrow()*** Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.
* ***get()*** If a value is present in this Optional, returns the value, otherwise throws *NoSuchElementException*.

**Stream API**

* A stream in Java is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
* We can use stream by importing ***java.util.stream*** package in our programs. This package consists of classes, interfaces, and an enum to allow functional-style operations on the elements.
* Stream operations are divided into ***intermediate*** and ***terminal*** operations.
* The terminal operations of the Java Stream interface typically return a single value. Terminal operations cannot be chained together. Intermediate operations return another stream as a result, they can be chained together to form a pipeline of operations.
* The JDK contains many terminal operations (such as average, sum, min, max, and count) that return one value by combining the contents of a stream. These operations are called ***reduction*** operations.
* The Stream provides the 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 do not modify its 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.
* We can use Stream to ***filter, collect, print***, and convert from one data structure to other etc.
* Stream supports aggregate operations like ***filter, map, limit, reduce, find, match***, and so on.
* Most of the stream operations return the stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. ***collect()*** method is a terminal operation that is normally present at the end of the pipelining operation to mark the end of the stream
* Collection interface has been extended with **stream()** and **parallelStream()** default methods to get the Stream for sequential and parallel execution.
* With Java 8, Parallel processing values are not in order, so parallel processing will be very helpful while working with huge collections.
* We can also use ***collectors*** to compute a sum of numeric values.
* Java Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it is not possible to reuse the same stream multiple times.
* Ways of Creating Stream
* **empty()** method should be used in case of the creation of an empty stream: ***Stream<String> stream = Stream.empty();***
* **Stream.of()** to create a stream from a similar type of data.
* Collection interface has two methods to generate a Stream.
* **stream()** − Returns a sequential stream considering collection as its source.
* **parallelStream()** − Returns a parallel Stream considering collection as its source.
* **Stream.generate() -** accepts a Supplier<T> for element generation. As the resulting stream is infinite, the developer should specify the desired size, or the generate() method will work until it reaches the memory limit.
* **Stream.iterate()** methods to create Stream
* *Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);*
* The first element of the resulting stream is the first parameter of the iterate() method. When creating every following element, the specified function is applied to the previous element. In the example above the second element will be 42.
* Using **Arrays.stream()** and **String.chars()** methods.
* Methods in Stream
* **filter()** to test stream elements for a condition and generate the filtered list. The filter method takes a Predicate that is called for each element in the stream.
* **map()** to apply functions to a stream. The map method is used to return a stream consisting of the results of applying the given function to the elements of this stream.
* *List number = Arrays.asList(2,3,4,5);*
* *List square = number.stream().map(x->x\*x).collect(Collectors.toList());*
* **sorted()** to sort the stream elements by passing the Comparator argument.
* *List names = Arrays.asList("Reflection","Collection","Stream");*
* *List result = names.stream().sorted().collect(Collectors.toList());*
* **flatMap()** to create a stream from the stream of the list. As the name suggests, it is the combination of a map and a flat operation. This means you first apply the map function and then flatten the result.
* **reduce()** to perform a reduction on the elements of the stream, using an associative accumulation function, and return an Optional.The reduce method takes a ***BinaryOperator*** as a parameter.
* **count()** to count the number of items in the stream.
* **forEach()** for iterating over the stream.
* **match()** for matching methods in Stream API.
* **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
* **allMatch()** checks if the predicate is true for all the elements in the stream
* **anyMatch()** checks if the predicate is true for any one element in the stream.
* **noneMatch()** checks if there are no elements matching the predicate.
* **findFirst()** to find the first element on condition, it returns an Optional.
* **peek()** performs the specified operation on each element of the stream and returns a new stream which can be used further. It is an intermediate operation.
* **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
* **limit()** to limit the number of elements in a stream to a number given to the limit() method as a parameter.
* **toArray()** to get an array out of the stream
* **collect()** to get stuff out of the stream once we are done with all the processing
* **Collectors.joining()** will insert the delimiter between the two String elements of the stream. It internally uses a java.util.StringJoiner to perform the joining operation.
* **Collectors.toCollection()** to extract the elements into any other collection by passing in a Supplier<Collection> - *Collectors.toCollection(Vector::new)*
* Java Stream API operations that return a new Stream are called ***intermediate operations***. Most of the time, these operations are lazy in nature, so they start producing new stream elements and send them to the next operation. Intermediate operations are never the result-producing operations. Commonly used intermediate operations are ***filter, map, flatMap, distinct, sorted, peek, limit, skip***.
* An intermediate operation is called ***short circuiting***, if it may produce a finite stream for an infinite stream. For example ***limit()*** *and* ***skip()*** are two short-circuiting intermediate operations.
* Java 8 Stream API operations that return a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can’t use the stream.
* ***Terminal operations*** are eager in nature ie they process all the elements in the stream before returning the result. Commonly used terminal methods are ***forEach, toArray, min, max, findAny, findFirst, anyMatch, allMatch, noneMatch, collect, count, reduce.***
* *A terminal operation is called short circuiting*, if it may terminate in finite time for infinite stream. For example ***anyMatch, allMatch, noneMatch, findFirst*** *and* ***findAny*** are short-circuiting terminal operations.

**Date**

* **java.util.Date** is not thread-safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
* ***java.time.temporal***.ChronoUnit enum is added in Java 8 to replace the integer values used in old API to represent day, month, etc
* The **Date-Time API** provides four classes that deal exclusively with date information, without respect to time or time zone.
* ***LocalDate*** represents a year-month-day in the ISO calendar and is useful for representing a date without a time. Some important methods in LocalDate class are below
* Get current date or today's using the method *LocalDate.now();*
* Get local date by passing year,month and dayOfMonth to the method *LocalDate.of(year, month, dayOfMonth);*
* Get next day of today's or specific date using the method *localDate.plusDays(1);*
* Get previous day by passing specific date to the method *localDate.minus(1, ChronoUnit.DAYS);*
* Get day of week using the method *DayOfWeek day = localDate.getDayOfWeek();*
* Get first day of the month *LocalDate.now().with(TemporalAdjusters.firstDayOfMonth());*
* Get start of the day *localDate.atStartOfDay();*
* ***YearMonth*** class represents the month of a specific year.
* *YearMonth.lengthOfMonth()* method to determine the number of days for several year and month combinations. *YearMonth.of(2010, Month.FEBRUARY)* return the month February of the year 2010.
* ***MonthDay*** class represents the day of a particular month
* ***Year*** class represents a year.
* Year.isLeap method to determine if the given year is a leap year. *Year.of(2012).isLeap();*
* The **Date-Time API** provides two classes that deal exclusively with time information.
* **LocalTime** class is similar to the other classes whose names are prefixed with Local, but deals in time only. This class is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.
* An instance of current LocalTime can be created from the system clock by *LocalTime now = LocalTime.now();*
* The below code samples explain how to create an instance using the factory “of” and “parse” methods

*LocalTime sixThirty = LocalTime.parse("06:30");*

*LocalTime sixThirty = LocalTime.of(6, 30);*

* **LocalDateTime** is used to represent a combination of date and time. The class offers a variety of APIs and some of the most commonly used ones are below.
* An instance of LocalDateTime can be obtained from the system clock similar to LocalDate and LocalTime *LocalDateTime.now();*
* The below code samples explain how to create an instance using the factory “of” and “parse” methods. The result would be a LocalDateTime instance representing 20 February 2015, 06:30 AM:

*LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);*

*LocalDateTime.parse("2015-02-20T06:30:00");*

* Adding days, subtract and getting month examples.

*localDateTime.plusDays(1);*

*localDateTime.minusHours(2);*

*localDateTime.getMonth()*

* The ***parse*** and the ***format*** methods throw an exception if a problem occurs during the conversion process. Therefore, your parse code should catch the ***DateTimeParseException*** error and your format code should catch the ***DateTimeException*** error.
* The **ZonedDateTime** class combines the *LocalDateTime* class with the *ZoneId* class.It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with a time zone (region/city, such as Europe/Paris).The java.time.ZonedDateTime class is an *immutable* class which represents a date-time with time-zone information such as ‘2017-06-16T21:25:37.258+05:30[Asia/Calcutta]’.
* A **Duration** is most suitable in situations that measure machine-based time, such as code that uses an Instant object. A Duration object is measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes. A Duration can have a negative value if it is created with an endpoint that occurs before the start point.
* **Instant** class represents the start of a nanosecond on the timeline. This class is used for generating a time stamp to represent machine time. Java Instant class is used to represent the specific moment on the timeline.
* The **Clock** class was added in Java 8 and provides access to an instant in time using the best available system clock, and to be used as a time provider which can be effectively stubbed for testing purposes. Instances of this class are used to find the current instant, which can be interpreted using the stored time-zone to find the current date and time. As such, a clock can be used instead of *System.currentTimeMillis()* and *TimeZone.getDefault().*
* The **Period** class represents a quantity of time in terms of years, months and days. A Period uses date-based values (years, months, days). This class is *immutable and thread-safe.*
* **DayOfWeek** is an enum representing the 7 days of the week - Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday. In addition to the textual enum name, each day-of-week has an int value. The int value follows the ISO-8601 standard, from 1 (Monday) to 7 (Sunday). It is recommended that applications use the enum rather than the int value to ensure code clarity.
* **Month** is an enum representing the 12 months of the year - January, February, March, April, May, June, July, August, September, October, November, and December. In addition to the textual enum name, each month-of-year has an int value. The int value follows normal usage and the ISO-8601 standard, from 1 (January) to 12 (December). It is recommended that applications use the enum rather than the int value to ensure code clarity.

**Base64**

* Java 8 now has inbuilt encoder and decoder for Base64 encoding. In Java 8, we can use three types of Base64 encoding.

1. **Simple** − Output is mapped to a set of characters lying in A-Za-z0-9+/. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
2. **URL** − Output is mapped to set of characters lying in A-Za-z0-9+\_. Output is URL and filename safe.
3. **MIME** − Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters each, and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. No line separator is present to the end of the encoded output.

**Important Questions**

1. **Based on the syntax rules just shown, which of the following is/are NOT valid lambda expressions?**
2. () -> {}
3. () -> “geeksforgeeks”
4. () -> { return “geeksforgeeks”;)
5. (Integer i) -> return “geeksforgeeks” + i;
6. (String s) -> {“geeksforgeeks”;}

4 and 5 are invalid lambdas, the rest are valid. Details:

1. This lambda has no parameters and returns void. It’s similar to a method with an empty body: public void run() { }.
2. This lambda has no parameters and returns a String as an expression.
3. This lambda has no parameters and returns a String (using an explicit return statement, within a block).
4. return is a control-flow statement. To make this lambda valid, curly braces are required as follows: (Integer i) -> { return “geeksforgeeks” + i; }.
5. “geeks for geeks” is an expression, not a statement. To make this lambda valid, you can remove the curly braces and semicolon as follows: (String s) -> “geeks for geeks”. Or if you prefer, you can use an explicit return statement as follows: (String s) -> { return “geeks for geeks”; }.
6. **Why the compose and identity method is omitted in Bifunction interface Java 8?**

***Function.identity()*** returns a Function that accepts a single argument and returns that argument. A BiFunction has two arguments, so which one of them would identity() return?

Similarly, ***compose()*** is used to compose two Functions by passing the result of the first Function as the single argument expected by the second Function. How would you implement something similar for a function that takes two arguments? The first function will result in a single value, but the second function required two arguments.

You can still compose a BiFunction with a Function by passing the result of the BiFunction as an argument for the Function. For that purpose you have ***andThen***.

1. **What is ‘reducing’ in the context of Streams?**

Reducing in the context of Java8 Streams refers to the process of combining all elements in the stream repeatedly to produce a single value which is returned as the result of the reduction operation.

*T reduce(T identity, BinaryOperator<T> accumulator)*

1. **Why cant we change the local variable value in lambda expressions?**

Lambda expressions can be reused many times. If the value is changed by one lambda expressions. The changed values will be available to other which is not correct. Due to that the local variable used in lambda expression is always final. But we can update the static or member(method) variables present inside the class. Because they stored on the heap, but local variables are on the stack. Because we're dealing with heap memory, the compiler can guarantee that the lambda will have access to the latest value always.

**Notes**:

* We cannot write the default methods as private, because that will violate the purpose of default methods. Compiler will throw an error it private is added.
* We can’t override the methods present inside the **Object class** as default methods inside an interface. The compiler will throw errors if we do so.
* We can’t write default methods inside a class. Even in the scenarios where we are overriding the default keyword should not be used inside the class methods
* From Java 8, just like we can write ***default methods*** inside interfaces, we can also write ***static methods*** inside them to define any utility functionality.
* Since ***static methods*** are allowed from Java 8, we can write a main method inside an interface and execute it as well
* ***peek()*** method in Stream API exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline“.
* ***Collections.reverse(list)*** does not use the comparator interface as underlying logic. It just uses the ***ListIterator*** to reverse the list.