Java 8 Features:

https://gbmishra.medium.com/a-comprehensive-guide-to-java-8-features-and-benefits-2cd8a2ee8dd3 -> Refer to this website for java 8 features

Latest Java Features include:

1) Lambda Expressions
2) Functional Interfaces
3) Stream APIs
4) Default and static methods
5) Optional Class
6) Method References
7) Collectors
8) New Date and Time API
9) Parallel Streams

1) Java 8 Interfaces (or) Functional Interfaces:

-> Java 8 introduces the use of "default" and "static" methods in the interface

Default Methods:

- Default methods allow you to add new methods to an interface without breaking the implementing classes.
- These methods have a body and can be overridden by the implementing class.
- Default methods can be accessed using the objectName created in main class
- // Syntax: objname.defaultMethodName();

Static Methods:

- Static methods in interfaces are similar to static methods in classes.
- They belong to the interface and are called using the interface name, not the implementing class.
- We declare methods with static keywords in interfaces and we can access them directly using the interface name.

Functional Interfaces:

- A functional interface is an interface with only one abstract method.
- Java 8 introduced the `@FunctionalInterface` annotation to indicate a functional interface.
- These interfaces can be used with **lambda expressions**. {These for loop ,Enhanced for loop is an external loop}

2) **ForEach() method**

-> **Traditional for loops** offer greater control, direct access to indices, and flexibility but can be more verbose and less readable.

-> Java 8 forEach simplifies iteration with a functional style, improves readability with lambda expressions, and integrates well with the Java Stream API for parallel processing.

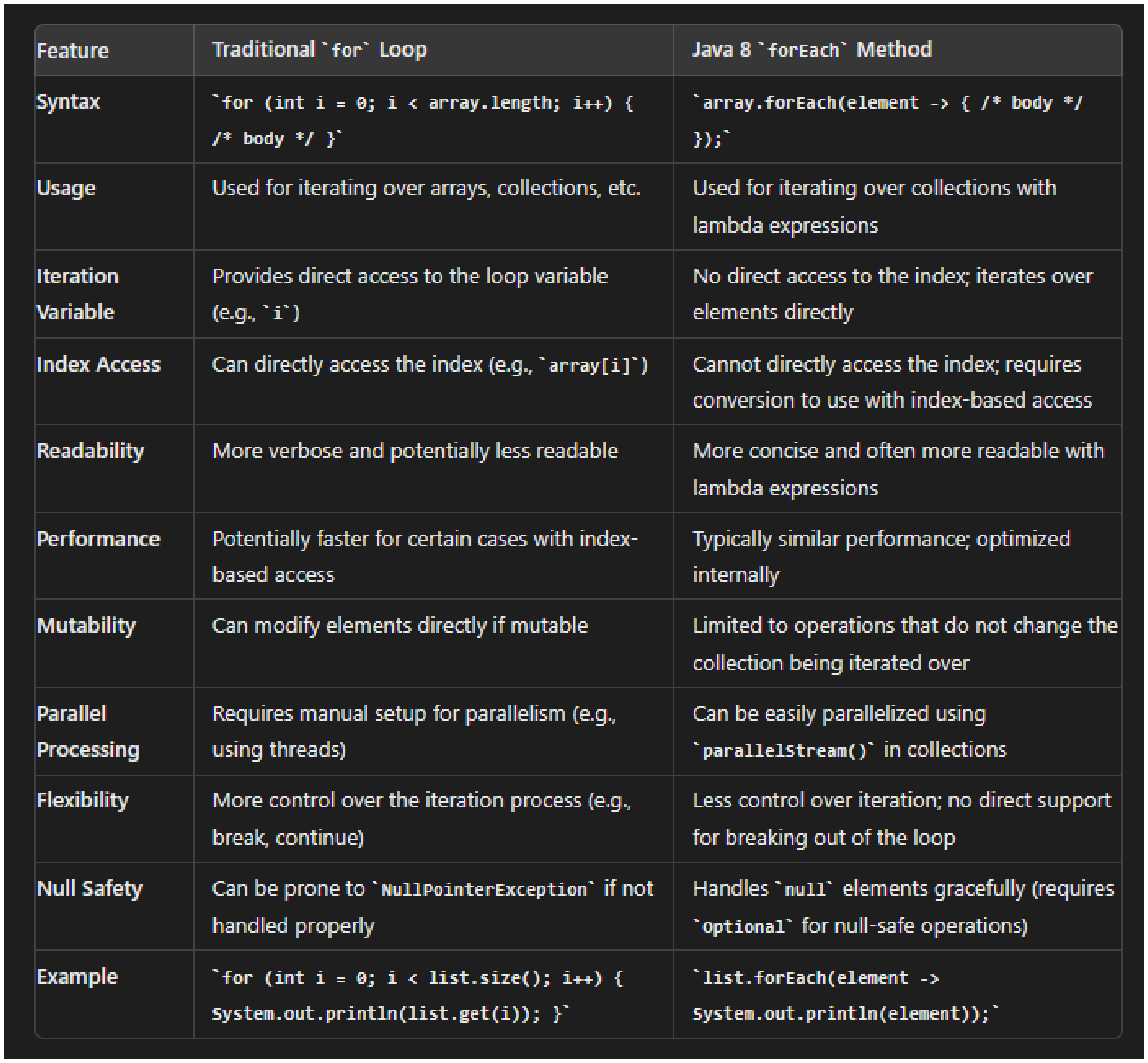
-> forEach() in Java 8 is an internal loop

Syntax for forEach() method:

Collections.forEach(element -> print_statements)

EX: objectName.forEach(element -> System.out.println(element)); //for Collections

Ex: ObjectName.stream().forEach(element -> System.out.println(element)); // for streams



Feature	Traditional `for` Loop	Java 8 `forEach` Method
Syntax	`for (int i = 0; i < array.length; i++) { /* body */ }`	`array.forEach(element -> { /* body */ });`
Usage	Used for iterating over arrays, collections, etc.	Used for iterating over collections with lambda expressions
Iteration Variable	Provides direct access to the loop variable (e.g., `i`)	No direct access to the index; iterates over elements directly
Index Access	Can directly access the index (e.g., `array[i]`)	Cannot directly access the index; requires conversion to use with index-based access
Readability	More verbose and potentially less readable	More concise and often more readable with lambda expressions
Performance	Potentially faster for certain cases with index-based access	Typically similar performance; optimized internally
Mutability	Can modify elements directly if mutable	Limited to operations that do not change the collection being iterated over
Parallel Processing	Requires manual setup for parallelism (e.g., using threads)	Can be easily parallelized using `parallelStream()` in collections
Flexibility	More control over the iteration process (e.g., break, continue)	Less control over iteration; no direct support for breaking out of the loop
Null Safety	Can be prone to `NullPointerException` if not handled properly	Handles `null` elements gracefully (requires `Optional` for null-safe operations)
Example	`for (int i = 0; i < list.size(); i++) { System.out.println(list.get(i)); }`	`list.forEach(element -> System.out.println(element));`

3) Lambda Expression:

-> A lambda can only represent (or) implement a functional interface (i.e., one method interface).

-> A lambda expression in Java is a way to provide clear and concise syntax for writing anonymous methods. Lambda expressions are used primarily to define the behavior of a functional interface in a more readable and compact form.

Syntax : (parameters)-> expression **or** (parameters)->{ statements }

-> Parameters: A comma-separated list of parameters (like method parameters).

-> Arrow Operator (`->`): Separates the parameters from the body of the lambda expression.

-> Expression or Block: The body of the lambda expression. It can be a single expression or a block of statements.

=> We have to call the default method from the interface using interfaceObjectName.

Ex: InferfaceObjectName.methodName();

=> We have to call the static method from the interface using InterfaceClassName.

Ex: InterfaceClassName.MethodName();

4) **Stream APIs:**

=>The **Stream API** in Java 8 is "used for processing collections of objects in a functional and declarative manner."

=>A Stream represents a sequence of elements that supports various operations to process data.Streams can be finite or infinite.

=> **Pipelining**: Stream operations can be chained together to form a processing pipeline.

=>**Parallelism**: Streams can be easily processed in parallel using parallelStream().

=> As streams are Declarative programming , Streams enable a more readable and expressive code structure, focusing on **what** you want to achieve rather than **how** to do it.

=> There are two types of operation are there in Streams.They are:

1) Intermediate Opearations
2) Terminal Operations

Intermediate Operations (returns another Stream):

- **`filter(Predicate)`**: Filters elements based on a condition.
- **`map(Function)`**: Transforms elements into another form.
- **`distinct()`**: Removes duplicates from the stream.
- **`sorted()`**: Sorts the elements.
- **`limit(long)`**: Limits the number of elements in the stream.
- **`skip(long)`**: Skips the first n elements of the stream.

Terminal Operations (ends the stream processing):

- **`forEach(Consumer)`**: Performs an action for each element.
- **`collect(Collector)`**: Collects the result into a collection like `List`, `Set`, or `Map`.
- **`reduce(BinaryOperator)`**: Combines all elements into a single result.
- **`count()`**: Returns the number of elements in the stream.
- **`anyMatch()`**, **`allMatch()`**, **`noneMatch()`**: Check if elements match a condition.

5) Java New Date and Time API

->**LocalDate**: Represents a date (year, month, day) without a time zone.
->**LocalTime**: Represents a time (hours, minutes, seconds, nanoseconds) without a date or time zone.
-> LocalDateTime: Represents both date and time, but without a time zone.
-> ZonedDateTime: Represents a date and time with a time zone.
-> Period: Represents a date-based amount of time, such as "2 years, 3 months, 4 days."
-> Duration: Represents a time-based amount of time, such as "34 minutes, 10 seconds."
-> Instant: Represents a timestamp, typically used for machine time (point in time in UTC).
-> DateTimeFormatter: Used to format or parse dates and times.

6) Method References in Java 8:

=> **Method references** in Java 8 provide a way to refer to methods directly, without invoking them.
=> They are a shorthand for lambda expressions where a method is used to perform a specific operation.

Syntax: ClassName::methodName

Types of Method References:

Java 8 provides four types of method references:

1. Reference to a static method
2. Reference to an instance method of a particular object
3. Reference to an instance method of an arbitrary object of a particular type
4. Reference to a constructor

=> Reference to a static method

Syntax : { ClassName::StaticMethodName }

=> Reference to an instance method of a particular object

You can refer to an instance method of a particular object using a method reference.

Syntax : { ClassObjectName::InstanceMethodName }

=> Reference to an Instance Method of an Arbitrary Object of a Particular Type

You can refer to an instance method of an arbitrary object of a particular type. This is commonly used when working with streams.

=> Here we will use the “datatype” to the example {like String,Integer,Double etc}

Syntax : { Datatype :: MethodName}

Ex: { String :: toUpperCase} //Converting data to uppercase

Ex: { System.out::println } // to print data

Basic Ex: data.forEach(System.out::println); // Will print the original data

=> Constructor reference.

You can refer to a constructor using the ClassName::new syntax. This is useful when you need to create an instance of a class within a stream or collection pipeline.

Syntax: ClassName::new

->**Constructor references** are shorthand for lambda expressions that call constructors.

->They are often used in scenarios where you want to pass the responsibility of object creation to a higher-order function (like in streams, collections, etc.).

=> They work with **functional interfaces** whose method signatures match the constructor’s signature.

In Constructor Reference

=> import java.util.function.BiFunction;   -> is used to take two values as input

=> import java.util.function.Function;   -> is used to take one value as input