**Java 8 Features:**

Default Methods:

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as **Extension Methods**. Considering the application of lamdas in java many interfaces must modify which will break the existing implementation.e.g forEach in Iterable interface. Due to the problem described above a new concept was introduced. **Virtual extension methods**, or, as they are often called, **defender methods**, can now be added to interfaces providing a default implementation of the declared behavior.

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

- When you extend an interface that contains a default method, you can do the following:

* Not mention the default method at all, which lets your extended interface inherit the default method.
* Redeclare the default method, which makes it abstract.
* Redefine the default method, which overrides it.

Simply speaking, interfaces in Java can now implement methods. The benefit that default methods bring is that now it’s possible to add a new default method to the interface and it doesn’t break the implementations

After introducing Default Method, it seems that interfaces and abstract classes are same. However, they are still different concept in Java 8.Abstract class can define constructor. They are more structured and can have a state associated with them. While in contrast, default method can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state. Hence, both use for different purposes and choosing between two really depends on the scenario context.

interface Formula {

double calculate(int a);

default double sqrt(int a) {

return Math.sqrt(a);

}

}

Note: Interface can also contain static methods. - you can also define [static methods](https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html)in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class

Multiple Inheritance:

If a class implements multiple interface having same default method then the class needs to override and specify the method.If you want to use default method written in one the implemented interface them we need to use <<InterfaceName>>.super.<<MethodName>>();

Lambda expressions:

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax. Lambdas are applicable to only functional interfaces.

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);

});

Collections.sort(names, (String a, String b) -> b.compareTo(a));

Collections.sort(names, (a, b) -> b.compareTo(a));

Functional Interface:

How does lambda expressions fit into Javas type system? Each lambda corresponds to a given type, specified by an interface. A so called functional interface must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.

We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method. To ensure that your interface meet the requirements, you should add the @FunctionalInterface annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

Example:

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

Keep in mind that the code is also valid if the @FunctionalInterface annotation would be ommited.

Method and Constructor Reference:

Java 8 enables you to pass references of methods or constructors via the :: keyword.

Method:

Convertor convertor = I -> Integer.valueOf(i);

Can be written as

Convertor convertor = Integer::valueOf;

Constructor:

PersonFactory pf = Person::new;

Lambda scopes:

You can access final variables from the local outer scope as well as instance fields and static variables. In constrast to local variables we have both read and write access to instance fields and static variables from within lambda expressions. Default methods **cannot** be accessed from within lambda expressions. Vaiables used in lambda expression needs to final otherwise the compiler generates the error "local variables referenced from a lambda expression must be final or effectively final“However, like local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final.

Build in Functional Interfaces:

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like Comparator or Runnable. Those existing interfaces are extended to enable Lambda support via the @FunctionalInterface annotation.

Perdicate: Represents a predicate (boolean-valued function) of one argument.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).

Predicate<String> predicate = s -> s.length() == 3;

System.***out***.println(predicate.and(s -> s.startsWith("f")).test("foo"));

Function: Represents a function that accepts one argument and produces a result.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-).

Function<String, Integer> func = Integer::*valueOf*;

String apply = func.andThen(i -> i.toString()).apply("5");

Supplier: Represents a supplier of results.There is no requirement that a new or distinct result be returned each time the supplier is invoked.

Supplier<Person> personSupplier = Person::new;

personSupplier.get(); // new Person

Consumer:Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));

BiFunction<T,U,R>: Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html" \o "interface in java.util.function).This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html#apply-T-U-).

BinaryOperator: Represents an operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html" \o "interface in java.util.function) for the case where the operands and the result are all of the same type.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [BiFunction.apply(Object, Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html" \l "apply-T-U-).

public interface **BinaryOperator<T>**

extends [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<T,T,T>

Optionals are not functional interfaces, instead it's a nifty utility to prevent NullPointerException. It's an important concept for the next section, so let's have a quick look at how Optionals work.

Optional is a simple container for a value which may be null or non-null. Think of a method which may return a non-null result but sometimes return nothing. Instead of returning nullyou return an Optional in Java 8.

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true

optional.get(); // "bam"

optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

Streams: A java.util.Stream represents a sequence of elements on which one or more operations can be performed. Stream operations are either intermediate or terminal. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Sorted method does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of stringCollection is untouched:

Stream : A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline.A stream represents a sequence of elements and supports different kind of operations to perform computations upon those element. Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. A naive approach would be to execute all the operations in the streams horizontally one after another on all elements of the stream. But instead each element moves along the chain vertically. This behavior can reduce the actual number of operations performed on each element.Operation attached to the streams are not called until it is attached to terminal operation.The actual of number CPU execution depends on the order in which operations are called in streams. streams cannot be reused. As soon as you call any terminal operation the stream is closed. To overcome this limitation we have to create a new stream chain for every terminal operation we want to execute(we can use Supplier functional interface)

 Intermediate operations: Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. An important characteristic of intermediate operations is laziness. intermediate operations will only be executed when a terminal operation is present.

Terminal operations: Terminal operations are either void or return a non-stream result.

Pipeline: A *pipeline* is a sequence of aggregate operations. The following example prints the male members contained in the collection roster with a pipeline that consists of the aggregate operations filter and forEach:

roster

.stream()

.filter(e -> e.getGender() == Person.Sex.MALE)

.forEach(e -> System.out.println(e.getName()));

A pipeline contains the following components:A source,intermediate(stream,filters),a terminal operation.

Sequential:Non parallel.Executes in one thread.

Different types of creating streams:

Streams can be created from various data sources, especially collections. Lists and Sets support new methods stream() and parallelStream() to either create a sequential or a parallel stream.e.g

Arrays.asList(“a1’,”a2’,”a3”);

Stream.of(“a1’,”a2’,”a3”);

Besides regular object streams Java 8 ships with special kinds of streams for working with the primitive data types int, long and double. As you might have guessed it's IntStream,LongStream and DoubleStream.

IntStreams can replace the regular for-loop utilizing IntStream.range() : IntStream.range(1,4).forEach(System.out::println);

Sometimes it's useful to transform a regular object stream to a primitive stream or vice versa. For that purpose object streams support the special mapping operations mapToInt(),mapToLong() and mapToDouble

 Primitive streams can be transformed to object streams via mapToObj()

 Streams operations like sort doesn’t happen on backed collection it only creates sorted view of streams.

### Differences Between Aggregate Operations and Iterators

Aggregate operations, like forEach, appear to be like iterators. However, they have several fundamental differences:

* **They use internal iteration**: Aggregate operations do not contain a method like next to instruct them to process the next element of the collection. With *internal delegation*, your application determines *what* collection it iterates, but the JDK determines *how* to iterate the collection. With *external iteration*, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. See the section [Parallelism](https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html) for more information.
* **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called *stream operations*.
* **They support behavior as parameters**: You can specify [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

Parallel Streams : A combiner is required in terminal operation if you are executing it in parallel mode otherwise it will execute in sequential mode.

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common ForkJoinPool available via the staticForkJoinPool.commonPool() method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores. sort is executed sequentially on the main thread only. Actually, sort on a parallel stream uses the new Java 8 method Arrays.parallelSort() under the hood. As stated in [Javadoc](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#parallelSort-T:A-) this method decides on the length of the array if sorting will be performed sequentially or in parallel:If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.For reduce operations combiner function is only called in parallel but not in sequential streams.

some parallel stream operations like reduce and collect need additional computations (combine operations) which isn't needed when executed sequentially.

Furthermore we've learned that all parallel stream operations share the same JVM-wide common ForkJoinPool. So you probably want to avoid implementing slow blocking stream operations since that could potentially slow down other parts of your application which rely heavily on parallel streams

Map: In java8 map has method like putIfAbsent and forEach() which takes BiConsumer has arguments.