Java 8 features:

- forEach() method in Iterable interface
  - It takes java.util.function.Consumer object as an argument.
- Default and static methods in interfaces
  - Explained below
- Functional interfaces
  - An interface with exactly one abstract method becomes Functional Interface. We don't need to use @FunctionalInterface annotation to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces.
  - Since functional interfaces have only one method, lambda expressions can easily provide the method implementation.
  - A new package java.util.function has been added with bunch of functional interfaces to provide target types for lambda expressions and method references.
- Java Streams API
  - A new java.util.stream has been added in Java 8 to perform filter/map/reduce like operations with the collection. Stream API will allow sequential as well as parallel execution.
  - Collection interface has been extended with *stream()* and *parallelStream()* default methods to get the Stream for sequential and parallel execution.
- Java Time API:
  - It has always been hard to work with Date, Time and Time Zones in java. There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.time package that will streamline the process of working with time in java.
  - It has some sub-packages
    - java.time.format provides classes to print and parse dates and times and
    - java.time.zone provides support for time-zones and their rules.
  - The new Time API prefers enums over integer constants for months and days of the week. One of the useful class is DateTimeFormatter for converting datetime objects to strings.
- Concurrency API improvements:
  - New methods added to ConcurrentHashMap
    - compute(), forEach(), merge(), reduce(), search()
  - CompletableFuture - that may be explicitly completed
  - Executors
    - newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.
- Java IO improvements:

Before Java 8, interfaces could have only public abstract methods. It was not possible to add new functionality to the existing interface without forcing all implementing classes to create an implementation of the new methods, nor it was possible to create interface methods with an implementation.

Default methods, Why?

- allow the interfaces to have methods with implementation without affecting the classes that implement the interface.
- provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods** or **virtual extension methods**.
- In case both the implemented interfaces contain deafult methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

Default methods are declared using the new default keyword. These are accessible through the instance of the implementing class and can be overridden.
Let's add a default method to our Vehicle interface, which will also make a call to the staticmethod of this interface:

```
default String getOverview() {
   return "ATV made by " + producer();
}
```

Assume that this interface is implemented by the class VehicleImpl. For executing the default method an instance of this class should be created:

```
Vehicle vehicle = new VehicleImpl();
String overview = vehicle.getOverview();
```

Static methods, why?

- since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.
- the scope of the static method definition is within the interface only.

Static methods:

```
static String producer() {
   return "N&F Vehicles";
}
```

The static producer() method is available only through and inside of an interface. It can't be overridden by an implementing class.

- Files.list(Path dir) - that returns a lazily populated Stream, the elements of which are the entries in the directory.
- Files.lines(Path path) - that reads all lines from a file as a Stream.
- Files.find() - that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
- BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

- Misc:
  - ThreadLocal static method withInitial(Supplier supplier) to create instance easily.
  - Comparator interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
  - min(), max() and sum() methods in Integer, Long and Double wrapper classes.
  - logicalAnd(), logicalOr() and logicalXor() methods in Boolean class.
  - ZipFile.stream() method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
  - Several utility methods in Math class.
  - jjs command is added to invoke Nashorn Engine.
  - jdeps command is added to analyze class files
  - JDBC-ODBC Bridge has been removed.
  - PermGen memory space has been removed

To call it outside the interface the standard approach for static method call should be used:
String producer = Vehicle.producer();

Optional:
Before Java 8 developers had to carefully validate values they referred to, because of a possibility of throwing the NullPointerException (NPE). All these checks demanded a pretty annoying and error-prone boilerplate code.

Java 8 Optional<T> class can help to handle situations where there is a possibility of getting the NPE. It works as a container for the object of type T. It can return a value of this object if this value is not a null. When the value inside this container is null it allows doing some predefined actions instead of throwing NPE.

```
Optional<String> optional = Optional.empty();

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

Optional<String> optional = Optional.ofNullable(getString());

List<String> list = getList();
List<String> listOpt = list != null ? list : new ArrayList<>();

List<String> listOpt = getList().orElseGet(() -> new
ArrayList<>());
```

```
String value = null;
String result = "";
try {
   result = value.toUpperCase();
} catch (NullPointerException exception) {
   throw new CustomException();
}

String value = null;
Optional<String> valueOpt = Optional.ofNullable(value);
String result =
valueOpt.orElseThrow(CustomException::new).toUpperCase
();
```

```
User user = getUser();
if (user != null) {
   Address address = user.getAddress();
   if (address != null) {
      String street = address.getStreet();
      if (street != null) {
         return street;
      }
   }
}
return "not specified";
```

Lambdas in Java 8:
A Lambda Expression ( or just a lambda for brevity) is a representation of an anonymous function which can be passed around as a parameter thus achieving behavior parameterization. A lambda consists of a list of parameters, a body, a return type and a list of exceptions which can be thrown. I.e. it is very much a function, just anonymous.

A lambda is a less verbose way of defining an instance of an interface provided the interface is functional.

Functional Interface:

```
@FunctionalInterface
public interface Foo {
   String method(String string);
}
```

A functional interface, introduced in Java 8, is an interface which has only a single abstract method. Conversely, if you have *any* interface which has only a single abstract method, then that will effectively be a functional interface. This interface can then be used anywhere where a functional interface is eligible to be used.

Map vs flatMap:
Both are aggregate functions.

map() -
method works well with Optional – if the function returns the exact type we need

e.g.:

```
Optional<String> s = Optional.of("test");
assertEquals(Optional.of("TEST"),
s.map(String::toUpperCase));
```

flatMap() -
However, in more complex cases we might be given a function that returns an Optional too. In such cases using map() would lead to a nested structure, as the map() implementation does an additional wrapping internally.

map() example:

```
assertEquals(Optional.of(Optional.of("STRING")),
Optional.of("string").map(s -> Optional.of("STRING")));
```

flatMap() example:

```
assertEquals(Optional.of("STRING"),
Optional.of("string").flatMap(s -> Optional.of("STRING")));
```

Method References:
Static method: anyMatch(User::isRealUser)
Instance method: anyMatch(user::isLegalName)
Instance method of an object of particular type:
filter(String::isEmpty)
Constructor: User::new

Streams:
Streams represent a sequence of objects

Optionals: are classes that represent a value that can be present or absent

Predicate:
predicate is a new functional interface defined in java.util.function package which can be used in all the contexts where an object needs to be evaluated for a given test condition and a boolean value needs to be returned based on whether the condition was successfully met or not. Since Predicate is a functional interface, hence it can be used as the assignment target for a lambda expression or a method reference.

Wherever an object needs to be evaluated and a boolean value needs to be returned( or a boolean-valued Predicate exists – in mathematical terms) the Predicate functional interface can be used. The user need not define his/her own predicate-type functional interface.

The primary purpose served by Functional Interfaces:
One of the most important uses of Functional Interfaces is that implementations of their abstract method can be passed around as lambda expressions. By virtue of their ability to pass around functionality(i.e. behavior), functional interfaces primarily enable behavior parameterization.

Tips:

1. Don't overuse default methods in functional interfaces.
2. Instantiate Functional Interfaces with Lambda Expressions.
   Foo foo = parameter -> parameter + " from Foo";
3. Avoid Overloading Methods with Functional Interfaces as Parameters.
4. Don't Treat Lambda Expressions as Inner Classes
5. Keep Lambda Expressions Short And Self-explanatory
   a. Avoid Blocks of Code in Lambda's Body
   b. Avoid Specifying Parameter Types
   c. Avoid Parentheses Around a Single Parameter
   d. Avoid Return Statement and Braces
   e. Use Method References
   f. Use "Effectively Final" Variables
   g. According to the "effectively final" concept, a compiler treats every variable as final, as long as it is assigned only once.
   h. Protect Object Variables from Mutation

4 fundamental and most commonly used functional interfaces.

| Functional Interface | Purpose |
|---|---|
| Consumer<T> | Represents an operation that accepts a single input argument and returns no result. |
| Function <T, R> | Represents a function that accepts one argument and produces a result. |
| Predicate<T> | Represents a predicate (boolean-valued function) of one argument. |
| Supplier<T> | Represents a supplier of results. |

Supplier:
Supplier<T> is an in-built functional interface introduced in Java 8 in the java.util.function package. Supplier can be used in all contexts where there is no input but an output is expected.

Function Descriptor of Supplier<T>: Supplier's Function Descriptor is () -> T . This means that there is no input in the lambda definition and the return output is an object of type T.

Advantage of predefined java.util.function.Supplier: In all scenarios where there is no input to an operation and it is expected to return an

Default methods in predicate:
and(), or() and negate()

Static methods in predicate:
isEqual()

Consumer:
Consumer<T> is an in-built functional interface introduced in Java 8 in the java.util.function package. Consumer can be used in all contexts where an object needs to be consumed,i.e. taken as input, and some operation is to be performed on the object without returning any result. Common example of such an operation is printing where an object is taken as input to the printing function and the value of the object is printed( we will expand upon the printing example in more detail below when understanding how to use Consumer interface).

Function Descriptor of Consumer<T>:Consumer's Function Descriptor is T -> (). This means an object of type T is input to the lambda with no return value.

Salient Points regarding Consumer<T>'s source code :

- Consumer has been defined with the generic type T which is the same type which its accept() & andThen() methods take as input.
- accept() method is the primary abstract method of the Consumer functional interface. Its function descriptor being T -> (). I.e. accept() method takes as input the type T and returns no value.
- All lambda definitions for Consumer must be written in accordance with accept method's signature, and conversely all lambdas with the same signature as that of accept() are candidates for assignment to an instance of Consumer interface.
- andThen() is a default method
- in Consumer interface. Method andThen(), when applied on a Consumer interface, takes as input another instance of Consumer interface and returns as a result a new consumer interface which represents aggregation of both of the operations defined in the two Consumer interfaces.

Reactor Core:
Reactor Core is a Java 8 library which implements the reactive programming model. It's built on top of the Reactive Streams Specification, a standard for building reactive applications.

Reactive Streams Specifications:

| | |
|---|---|
| output the in-built functional interface Supplier<T> can be used without the need to define a new functional interface every time.<br><br>Salient Points regarding Supplier<T>'s source code :<br>● Supplier has been defined with the generic type T which is the same type which its get() methods return as output.<br>● get() method is the primary abstract method of the Supplier functional interface. Its function descriptor being () -> T . I.e. get() method takes no input and returns an output of type T.<br>● All lambda definitions for Supplier must be written in accordance with get() method's signature, and conversely all lambdas with the same signature as that of get() are candidates for assignment to an instance of Supplier interface. | specification for asynchronous stream processing. r*eactive Streams* is a standard for asynchronous stream processing with non-blocking back pressure. |

Note: Information gathered in this document has been collected from various sources on Internet.
Sources:
https://www.baeldung.com/java-8-new-features
https://www.baeldung.com/java-8-functional-interfaces
https://www.baeldung.com/java-8-lambda-expressions-tips
https://www.javabrahman.com/java-8/