Major Java 8 Features Introduced

There are a few major Java 8 features mentioned below:

- **Lambda Expressions**: Concise functional code using `->`.
- **Functional Interfaces**: Single-method interfaces.
- **Introduced and Improved APIs:**
    1. **Stream API**: Efficient Data Manipulation.
    2. **Date/Time API**: Robust Date and Time Handling.
    3. **Collection API Improvements**: Enhanced Methods for Collections (e.g., `removeIf`, `replaceAll`).
    4. **Concurrency API Improvements**: New classes for parallel processing (e.g., `CompletableFuture`).
- **Optional Class**: Handle null values safely.
- **`forEach()` Method in Iterable Interface**: Executes an action for each element in a Collection.
- **Default Methods**: Evolve interfaces without breaking compatibility.
- **Static Methods**: Allows adding methods with default implementations to interfaces.
- **Method References**: Refer to methods easily.

Lambda Expressions:
Lambda expressions in Java, introduced in Java SE 8, represent instances of functional interfaces (interfaces with a single abstract method). They

provide a concise way to express instances of single-method interfaces using a block of code.

Lambda Expressions are anonymous functions. These functions do not need a name or a class to be used. Lambda expressions are added in Java 8. Lambda expressions express instances of functional interfaces An interface with a single abstract method is called a functional interface. One example is java.lang.Runnable.

Lambda expressions implement only one abstract function and therefore implement functional interfaces.

- **Functional Interfaces**: Lambda expressions implement single abstract methods of functional interfaces.
- **Code as Data**: Treat functionality as a method argument.
- **Class Independence**: Create functions without defining a class.
- **Pass and Execute**: Pass lambda expressions as objects and execute on demand.

Lambda Expression Syntax

```
lambda operator -> body
```

Lambda Expression Parameters

There are three Lambda Expression Parameters are mentioned below:

1. Zero Parameter
2. Single Parameter
3. Multiple Parameters

1. Lambda Expression with Zero parameter

```
() -> System.out.println("Zero parameter lambda");
```

2. Lambda Expression with Single parameter

```
(p) -> System.out.println("One parameter: " + p);
```

It is not mandatory to use parentheses if the type of that variable can be

Lambda Expression with Multiple parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 +
", " + p2);
```

Functional Interface:

Functional interface can have only one abstract method. It can contain multiple default and static methods. We use @FunctionalInterface Annotation is not mandatory but it is useful to ensure that the interface has only one abstract method.

Runnable, ***ActionListener***, *and* ***Comparable*** are some of the examples of functional interfaces.

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

- **Runnable –>** This interface only contains the run() method.

- **Comparable –>** This interface only contains the compareTo() method.
- **ActionListener –>** This interface only contains the actionPerformed() method.
- **Callable –>** This interface only contains the call() method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

1. Consumer

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer.

```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
}
```

Bi-Consumer – Bi-Consumer is the most exciting variant of the Consumer interface. The consumer interface takes only one argument, but on the other side, the Bi-Consumer interface takes two arguments. Both, Consumer and Bi-Consumer have no return value. It also returns nothing just like the Consumer interface. It is used in iterating through the entries of the map.

Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate.

biConsumer

```
public interface Predicate<T> {

    boolean test(T t);

}
```

Function

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. There are many versions of Function interfaces because a primitive type can't imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long. The different sequences of these primitive types are also used in the argument.

These versions are:

Bi-Function

The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

The prototype and syntax of Bi-Function is given below –

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{

   R apply(T t, U u);
    .......

}
```

In the above code of interface, T and U are the inputs, and there is only one output which is R.

Supplier

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream. generate method, which is implemented by the Supplier functional Interface.