Java 8 features

1. Functional Interfaces
2. Lambda Expression
3. Method Expression
4. Stream API
5. Default and Static Method
6. forEach() Method
7. Date Time API
8. Optional Class
9. Collectors Class
10. Parallel Array Sorting

1. forEach() method

- The forEach() method is defined as a default method in Iterable Interface.
- It is used to iterate through elements
- Collection Interface extends Iterable Interface thus, classes implementing Collection Framework like Arrays, Lists, Stacks, etc. also extend Iterable Interface.
- The forEach() method facilitates functional programming in Java, i.e., we can use it to transverse elements/objects in a functional style or while working with stream API.
- It takes only a single Consumer object as an argument like lambda expression, method reference, etc.
- forEach() method takes action to be performed on each element of a Collection as the parameter.
- e.g.,

```
      HashMap<String,Integer> months = new HashMap<>();
      //Declaring a hash map and adding key-value pairs
      months.put("January",31);
      months.put("April",30);
      months.put("September",30);
      months.put("July",31);
      months.put("February",28);
      months.put("December",31);
      System.out.println("Months having 31 days from our list are:");

    months.forEach((month,days)->
    {
      if(days == 31) // if month has 31 days, printing month
        System.out.println(month);
    });
```

2. Date Time API

- Existing Date and Time APIs in Java were not thread-safe, there was a lack of consistency as java.util and java.sql both packages define Date class, and it didn't have support for timezone, so developers had to write an additional logic for timezone and thread-safety.
- Java 8 introduced the java.time package.
- The new Date and Time APIs are immutable and thus thread-safe, follow consistent domain models for the date, time, duration, and periods as well as support Local and Zonal Date/Time APIs.
- New Date and Time API have two important classes among them:
    1. **Local** - This class provides simple Date and Time operations in the Local Time Zone.
    2. **Zoned** - It contains Time zone-specific Date/Time and its operations.

- **LocalDate** datatype stores date, LocalDate.now() returns System's Date.

```
LocalDate CurrentDate = LocalDate.now();
```

- Similarly, **LocalTime** datatype stores time in hours, minutes, and seconds, LocalTime.now() returns the current time.

```
LocalTime todaysTime = LocalTime.now();
```

- **todaysTime.getHour()** and **todaysTime.getMinute()** returns value of hour and value of minute in LocalTime todaysTime respectively.

```
todaysTime.getHour()

todaysTime.getMinute()
```

- **parse()** method converts String to LocalDateTime. Its Format is yyyy-mm-ddThh:mm.

```
LocalDateTime date_Time = LocalDateTime.parse(LocalDate.now().plusDays(1)+"T09:45:00");

// LocalDateTime stores date(here, after adding 1) and time
```

- **LocalDateTime** stores date and time both in the same variable date_Time. LocalDate.now().plusDays(1) adds one day to the date i.e.28th January becomes 29th January in our example. We can add more than one day, month, etc to the date. In 22-01-29T09:45:00, 22-01-29 represents date in format yyyy-mm-dd and 09:45:00 represents time in format hh-mm-ss. T in between date and time separates both and denotes the time value in the LocalDateTime variable.

- **date_Time.getDayOfWeek()** returns the day on a particular date. In our example it is SATURDAY.

```
date_Time.getDayOfWeek()
```

- We can also use different Zones and ZonedDateTime.
- **ZoneId** stores the Zone we want to work upon, like Europe/London, Europe/Paris, Asia/Tokyo, etc. ZonedDateTime takes date and time and ZoneId as arguments.

```
ZoneId londonZone = ZoneId.of("Europe/London");

ZonedDateTime zonedDateTime = ZonedDateTime.of(date_Time,londonZone);
```

3. Optional class

- Optional class is present in the Java.util package.
- It is a public final class used to avoid NullPointerExceptions in Java Applications.
- The optional class contains methods that provide easy and concise ways to check if some value is null and perform an action if it is null.

- **Optional.ofNullable()** -
  - if a given object is null it returns an empty Optional, else it returns Non-empty Optional.
  - As in our example message is null, it will return empty Optional.
- **StringOptional.isPresent()** -
  - It returns true if Optional is Non-empty, else returns false. Thus, it prints Message is Empty.
- **StringOptional.orElse() -**
  - it assigns a value only if the given Optional is Empty.

4. Collectors Class

- The Collectors class extends the Object class and is located in the java.util.stream package.
- It is a final class.

- It provides various methods for reduction operations like accumulating elements into collections, summarizing them according to some criteria, etc.
- We can collect data using toList().
- we can also use toSet, toMap, toCollection, etc. as per the requirement.

5. Parallel Array Sorting

- The **parallelSort()** method is introduced in the Array class of java.util package.
- It uses the concept of multithreading in order to sort the array faster.
- It first goes on dividing the array into subarrays, these subarrays are sorted individually by multiple threads and then merged together.

```
import java.util.Arrays;
public class ParallelSorting {
    public static void main(String args[]){
        int[] nums = new int[]{673,982,82,749,102,4873,6241,9572,511};

        //Sorting array
        Arrays.parallelSort(nums);

        //using stream to print values
        Arrays.stream(nums).forEach(n->System.out.print(n+" "));
    }
}
```