Java 8 – Complete and Detailed Notes

1. Introduction to Java 8

Java 8 was a major release in 2014 that brought powerful functional programming capabilities and improvements to existing APIs. Key features:

- **Lambda Expressions**
- **Functional Interfaces**
- **Stream API**
- **Method References**
- **Optional Class**
- **Default & Static Methods in Interfaces**
- **New Date/Time API**
- **Collectors**
- **Parallel Streams**

2. Lambda Expressions

Definition

- Lambda expressions let you write anonymous functions. You can pass behavior as a method parameter.

Syntax

(parameter1, parameter2) -> { body }

Example

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.forEach(name -> System.out.println(name));
```

Advantages

- Eliminates boilerplate code
- Enables functional programming

- Improves readability

---

3. Functional Interfaces

Definition

- An interface with **only one abstract method**. Used as the basis for lambda expressions.

Examples of Built-in Functional Interfaces

Interface	**Abstract Method**	**Description**
Runnable	run()	No input, no output
Callable<T>	call()	Returns a value
Function<T,R>	apply(T)	One input, one output
Predicate<T>	test(T)	Returns boolean
Consumer<T>	accept(T)	Takes input, no output
Supplier<T>	get()	Returns a value

Example

```
@FunctionalInterface
interface MyFunction {
   void run();
}
MyFunction f = () -> System.out.println("Running");
```

---

4. Stream API

What is Stream API?

- A **stream** is a sequence of data that supports functional-style operations like filter, map, reduce, etc.

Benefits

- **Declarative**: less boilerplate
- **Lazy Execution**: intermediate ops are executed only when terminal operation runs
- **Parallel Processing**
- **Pipeline Processing**

Common Stream Methods (with examples):

◈ filter(Predicate)

- Filters elements based on a condition.

```
List<String> names = List.of("Apple", "Banana", "Avocado");

names.stream().filter(name ->
name.startsWith("A")).forEach(System.out::println);
```

◈ map(Function)

- Transforms each element.

```
List<String> words = List.of("java", "spring");

words.stream().map(String::toUpperCase).forEach(System.out::println);
```

◈ sorted()

- Sorts the stream.

```
List<Integer> nums = List.of(3, 1, 4);

nums.stream().sorted().forEach(System.out::println);
```

❖ skip(n)

- Skips first n elements.

```
List<Integer> nums = List.of(1,2,3,4,5);
nums.stream().skip(2).forEach(System.out::println);
```

❖ distinct()

- Removes duplicates.

```
List<Integer> nums = List.of(1,2,2,3);
nums.stream().distinct().forEach(System.out::println);
```

❖ limit(n)

- Limits to first n elements.

```
List<String> names = List.of("A","B","C","D");
names.stream().limit(2).forEach(System.out::println);
```

❖ findFirst()

- Returns first element (Optional).

```
Optional<String> result = List.of("One", "Two").stream().findFirst();
System.out.println(result.orElse("Empty"));
```

❖ allMatch(Predicate)

- Checks if all match a condition.

```
boolean allEven = List.of(2, 4, 6).stream().allMatch(i -> i % 2 == 0);
```

❖ forEach(Consumer)

- Applies a function to each element.

```
List.of(1, 2, 3).forEach(System.out::println);
```

◈ collect(Collectors)

- Terminal operation to collect stream into Collection/Map/etc.

```
List<String> list = List.of("a", "b").stream().collect(Collectors.toList());
Set<String> set = List.of("a", "b").stream().collect(Collectors.toSet());
```

◈ 5. Streams on Map

◈ entrySet(), keySet(), values()

```
Map<Integer, String> map = Map.of(1, "One", 2, "Two");


map.entrySet().stream()
  .filter(e -> e.getKey() > 1)
  .forEach(e -> System.out.println(e.getKey() + ":" + e.getValue()));


map.keySet().stream().forEach(System.out::println);
map.values().stream().forEach(System.out::println);
```

◈ Collectors.toMap()

```
List<String> names = List.of("Ram", "Sham");
Map<Integer, String> map = names.stream()
    .collect(Collectors.toMap(String::length, s -> s));
```

◈ mapToInt, sum, count, average, max, min

```
int sum = List.of(1, 2, 3).stream().mapToInt(i -> i).sum();

long count = List.of(1, 2, 3).stream().count();

OptionalDouble avg = List.of(1, 2, 3).stream().mapToInt(i -> i).average();
```

◈ flatMap(Function)

- Used to flatten nested lists.

```
List<List<String>> nested = List.of(List.of("A"), List.of("B", "C"));
nested.stream().flatMap(List::stream).forEach(System.out::println);
```

◈ parallelStream()

- Used to process in multiple threads.

```
List.of(1,2,3,4).parallelStream().forEach(System.out::println);
```

---

6. Default and Static Methods in Interfaces

Default Method

```
interface Vehicle {

    default void start() {

        System.out.println("Vehicle started");

    }

}

class Car implements Vehicle {}
```

Static Method

```
interface Helper {
   static void print() {
      System.out.println("Static in interface");
   }
}
Helper.print();
```

---

7. Optional Class

```
Optional<String> name = Optional.ofNullable("Java");
name.ifPresent(System.out::println);


Optional<String> empty = Optional.empty();
System.out.println(empty.orElse("Default")); // Default
```

---

8. Method References

```
List<String> list = List.of("a", "b");
list.forEach(System.out::println); // instead of name ->
System.out.println(name)


Function<String, Integer> parser = Integer::parseInt;
Supplier<List<String>> supplier = ArrayList::new;
```

---

9. New Date and Time API (java.time)

```
// LocalDate
LocalDate today = LocalDate.now();
LocalDate birthDate = LocalDate.of(1996, 7, 10);


// Period
Period age = Period.between(birthDate, today);
System.out.println("Years: " + age.getYears());


// LocalDateTime
LocalDateTime dt = LocalDateTime.now();


// ZonedDateTime
ZonedDateTime zdt = ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));


// Formatting
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
System.out.println(LocalDate.now().format(formatter));
```

Summary

Java 8 revolutionized Java with functional programming, better data handling, and cleaner code. It's essential for modern development and interviews.