---

1. **Lambda Expressions** (Foundation)

- **Importance**: Lambda expressions are the core feature of Java 8 and allow you to write cleaner, more readable, and concise code, especially for functional-style operations.
- **Learn**:
  - Syntax of lambda expressions.
  - Using lambdas with functional interfaces (e.g., `Runnable`, `Comparator`, `Predicate`).
  - Replacing anonymous classes with lambda expressions.
  - **Practical Exercises**: Replace anonymous inner classes with lambda expressions in code.

```
List<String> names = Arrays.asList("Tejas", "Java", "Spring");
names.forEach(name -> System.out.println(name));
```

---

2. **Functional Interfaces** (Closely tied to lambdas)

- **Importance**: A functional interface has a single abstract method and is crucial for understanding lambda expressions.
- **Learn**:
  - Built-in functional interfaces like `Predicate`, `Function`, `Consumer`, `Supplier`.
  - Create custom functional interfaces.
  - **Practical Exercises**: Implement built-in functional interfaces and create custom ones.

```
Predicate<Integer> isEven = number -> number % 2 == 0;
System.out.println(isEven.test(4));  // true
```

---

3. **Streams API** (Essential for functional programming)

- **Importance**: The Streams API simplifies working with collections, allowing for powerful data manipulation, filtering, and transformation.
- The **Streams API** in Java 8 is one of the most important additions, as it allows for **functional-style operations** on collections and other data sources. Streams provide a clear, concise, and efficient way to process data using a series of transformations, filters, and operations. Here's a detailed breakdown of the Streams API:
- **Streams** represent a sequence of elements on which one or more operations can be performed.
- Streams do not store data; instead, they work on the data source (such as a collection, array, or I/O channel).
- Operations on streams are **lazily evaluated**, meaning that they are only executed when a terminal operation is called.
- Streams can be either **sequential** or **parallel** (for parallel processing).

Key Features of Streams API:

1. **Declarative**: You describe **what** you want to do, not **how**.
2. **Functional-style operations**: These include map, filter, reduce, etc., enabling operations like transforming, filtering, and collecting data.
3. **Lazy evaluation**: Intermediate operations like `filter` or `map` are lazy, which means they are not executed until a terminal operation (like `collect`, `forEach`) is invoked.
4. **Parallel Processing**: You can easily parallelize stream operations using **parallel streams**.

How to Create Streams:

1. **Collections**:

```java
List<String> list = Arrays.asList("Tejas", "Java", "Stream");
Stream<String> stream = list.stream();
```

2. **Arrays**:

```java
String[] array = {"Tejas", "Java", "Stream"};
Stream<String> stream = Arrays.stream(array);
```

3. **Stream.of()**:

```java
Stream<String> stream = Stream.of("Tejas", "Java", "Stream");
```

4. **Infinite Streams** using `Stream.generate()` or `Stream.iterate()`:

```
Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);
```

Stream Operations:

1. Intermediate Operations (Lazy operations):

Intermediate operations return a **stream**, meaning they can be chained together. They are not executed until a terminal operation is called.

- `filter()`: Filters elements based on a condition.

  ```
  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
  List<Integer> evenNumbers = numbers.stream()
      .filter(n -> n % 2 == 0)
      .collect(Collectors.toList());
  ```

- `map()`: Transforms each element of the stream using a function.

  ```
  List<String> names = Arrays.asList("Tejas", "Java", "Stream");
  List<Integer> nameLengths = names.stream()
      .map(String::length)
      .collect(Collectors.toList());
  ```

- `sorted()`: Sorts the stream elements (natural or custom order).

  ```
  List<String> names = Arrays.asList("Tejas", "Java", "Stream");
  List<String> sortedNames = names.stream()
      .sorted()
      .collect(Collectors.toList());
  ```

- `distinct()`: Removes duplicate elements from the stream.

  ```
  List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4);
  List<Integer> distinctNumbers = numbers.stream()
      .distinct()
      .collect(Collectors.toList());
  ```

- **`limit()` and `skip()`**: Limit the number of elements or skip a number of elements.

  ```
  List<Integer> limitedNumbers = Stream.iterate(1, n -> n + 1)
      .limit(5)
  ```

```
    .collect(Collectors.toList());  // [1, 2, 3, 4, 5]
```

2. Terminal Operations (Execute the stream):

Terminal operations cause the processing of the stream to be performed and return a result, such as a collection, value, or void.

- `forEach()`: Performs an action for each element of the stream.

  ```
  List<String> names = Arrays.asList("Tejas", "Java", "Stream");
  names.stream().forEach(System.out::println);
  ```

- `collect()`: Collects the stream into a collection, such as a `List`, `Set`, or `Map`.

  ```
  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
  List<Integer> doubled = numbers.stream()
      .map(n -> n * 2)
      .collect(Collectors.toList());
  ```

- `reduce()`: Performs a reduction on the elements of the stream, such as summing numbers or finding the max/min.

  ```
  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
  int sum = numbers.stream()
      .reduce(0, Integer::sum);  // sum = 15
  ```

- `count()`: Returns the count of elements in the stream.

  ```
  long count = Stream.of("Tejas", "Java", "Stream")
      .count();  // count = 3
  ```

- `findFirst()` / `findAny()`: Returns the first element or any element from the stream (findFirst is deterministic; findAny is non-deterministic, useful in parallel streams).

  ```
  Optional<String> first = Stream.of("Tejas", "Java", "Stream")
      .findFirst();  // Optional[Tejas]
  ```

Common Use Cases of Streams API:

1. Filtering and Collecting:

Filter and collect only specific elements.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());  // [2, 4]
```

2. Transformation (Map):

Transform each element into something else (e.g., converting strings to uppercase).

```
List<String> names = Arrays.asList("Tejas", "Java", "Stream");
List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());  // [TEJAS, JAVA, STREAM]
```

3. Sorting:

Sorting elements in natural or custom order.

```
List<String> names = Arrays.asList("Tejas", "Java", "Stream");
List<String> sortedNames = names.stream()
    .sorted()
    .collect(Collectors.toList());  // [Java, Stream, Tejas]
```

4. Group by using `Collectors.groupingBy()`:

Group elements based on a condition (e.g., grouping by string length).

```
List<String> names = Arrays.asList("Tejas", "Java", "Stream");
Map<Integer, List<String>> groupedByLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
```

5. Parallel Streams:

Process elements in parallel to boost performance on large datasets.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().forEach(System.out::println);
```

Best Practices for Streams API:

1. **Avoid Modifying the Source Collection**: Streams are designed to work on immutable data.
2. **Use Parallel Streams with Caution**: Only use parallel streams when you have a significant amount of data to process. For small datasets, it may introduce unnecessary overhead.
3. **Favor Method References Over Lambdas**: When possible, use method references (e.g., `System.out::println`) for cleaner and more readable code.
4. **Minimize Side Effects**: Avoid using streams for operations with side effects, like modifying external variables inside stream operations (except in terminal operations like `forEach()`).

---

4. Method References (Shorter and more readable code)

- **Importance**: Method references simplify the syntax further, especially in combination with lambdas and the Streams API.

- **Learn**:

  - Types of method references: static methods, instance methods, and constructors.
  - **Practical Exercises**: Refactor lambda expressions to use method references where possible.

  ```
  ClassName::methodName
  ```

- Method references can be used to refer to:

  - Static methods.

  ```
  (args) -> ClassName.staticMethod(args)
  // Becomes
  ClassName::staticMethod
  ```

  - Instance methods of a particular object.

  ```
  (args) -> instance.method(args)
  // Becomes
  instance::method
  ```

  - Instance methods of an arbitrary object of a particular type.

  ```
  (args) -> instance.method(args)
  // Becomes
  instance::method
  ```

  - Constructors.

```
() -> new ClassName(args)
// Becomes
ClassName::new
```

---

5. **Default and Static Methods in Interfaces** (Enhances interface capabilities)

- **Importance**: Java 8 introduced default and static methods in interfaces to provide method implementations in interfaces without breaking existing implementations.
- **Learn**:
  - Adding default methods in interfaces.
  - Defining static methods in interfaces.
  - **Practical Exercises**: Add a default method to an existing interface without breaking the implementation.

```
interface MyInterface {
    default void sayHello() {
        System.out.println("Hello from MyInterface");
    }
}
```

---

6. **Optional Class** (Handling null values gracefully)

- **Importance**: `Optional` helps to avoid `NullPointerException` by providing methods for explicitly handling null values.
- **Learn**:
  - Creating `Optional` objects.
  - Methods: `isPresent()`, `orElse()`, `orElseThrow()`, `ifPresent()`.
  - **Practical Exercises**: Use `Optional` to avoid null checks in code.

```
Optional<String> name = Optional.ofNullable(null);
System.out.println(name.orElse("Default Name"));  // Prints "Default Name"
```

---

7. **Collectors** (Data collection and aggregation)

- **Importance**: The `Collectors` utility class provides powerful ways to collect and transform data from streams into collections, strings, maps, etc.
- **Learn**:
  - Using `Collectors.toList()`, `Collectors.toMap()`, `Collectors.joining()`.
  - Grouping and partitioning data with `Collectors.groupingBy()` and `Collectors.partitioningBy()`.
  - **Practical Exercises**: Use collectors to group, partition, and transform stream results.

```
List<String> names = Arrays.asList("Tejas", "Java", "Spring");
String joinedNames = names.stream().collect(Collectors.joining(", "));
System.out.println(joinedNames);  // "Tejas, Java, Spring"
```

---

8. Stream Parallelism (Parallel Streams) (For performance optimization)

- **Importance**: Parallel streams allow for parallel processing of data, potentially speeding up tasks that can be parallelized.
- **Learn**:
  - Creating parallel streams.
  - Understanding when and when not to use parallel streams (performance considerations).
  - **Practical Exercises**: Use parallel streams to process large datasets in parallel.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().forEach(System.out::println);
```

---

9. New Date and Time API (java.time package) (Modern date and time handling)

- **Importance**: Java 8 introduced a much-improved `java.time` package, replacing the old `java.util.Date` and `java.util.Calendar` APIs.
- **Learn**:
  - Classes: `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Period`, `Duration`.
  - Formatting and parsing dates with `DateTimeFormatter`.
  - **Practical Exercises**: Use the new date and time API to perform date manipulations and format dates.

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.of(12, 30);
System.out.println(date);  // 2024-10-31
```

---

10. CompletableFuture (Asynchronous Programming) (Advanced)

- **Importance**: `CompletableFuture` is a powerful tool for writing asynchronous code in Java.
- **Learn**:
  - Creating and running asynchronous tasks.
  - Combining multiple `CompletableFuture` tasks.
  - Error handling with `CompletableFuture`.
  - **Practical Exercises**: Implement asynchronous tasks using `CompletableFuture`.

```
CompletableFuture.supplyAsync(() -> "Hello")
    .thenAccept(result -> System.out.println(result));
```

---

11. Nashorn JavaScript Engine (Advanced)

- **Importance**: The Nashorn engine allows running JavaScript code within Java applications.
- **Learn**:
  - Running JavaScript code from Java using Nashorn.
  - Embedding JavaScript in Java applications.
  - **Practical Exercises**: Execute JavaScript code using the Nashorn engine.

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval("print('Hello from Nashorn')");
```

---

12. New Arrays Methods (Arrays.parallelSort) (Parallel array sorting)

- **Importance**: Java 8 added methods for parallel sorting of arrays, which can improve performance in some cases.
- **Learn**:
  - Sorting arrays using `Arrays.parallelSort()`.
  - Performance considerations of parallel sorting.
  - **Practical Exercises**: Sort large arrays using `parallelSort` and compare performance with regular sorting.

```
int[] numbers = {5, 3, 8, 1, 9};
Arrays.parallelSort(numbers);
System.out.println(Arrays.toString(numbers));  // [1, 3, 5, 8, 9]
```

---

13. Improvements in Collections (Advanced)

- **Importance**: Java 8 introduced small but useful improvements in the `Collections` framework.
- **Learn**:
  - `Map.computeIfAbsent()` and `Map.computeIfPresent()`.
  - `forEach()` method in `Map` and other collection classes.
  - **Practical Exercises**: Use the new collection methods to simplify map handling.

```
Map<String, Integer> map = new HashMap<>();
map.computeIfAbsent("Java", key -> 1);
```

COLLECTORS CLASS METHOD USE CASES

- The `Collectors` class in Java is part of the Stream API, and it provides various methods to create collectors, which can be used to perform reduction operations on streams like grouping, counting, summarizing, etc.
- These methods are often used with `Stream.collect()` to process and accumulate elements of a stream into various forms such as lists, sets, maps, or even custom results.

USE CASES LIST:

- **`Collectors.groupingBy()`**: Group elements by a key.
- **`Collectors.counting()`**: Count elements.
- **`Collectors.toList()`**: Collect elements into a list.
- **`Collectors.toSet()`**: Collect elements into a set.
- **`Collectors.joining()`**: Concatenate elements into a string.
- **`Collectors.summingInt()`**: Sum integer values.
- **`Collectors.averagingInt()`**: Calculate the average of integer values.
- **`Collectors.maxBy()`**: Find the maximum element.
- **`Collectors.mapping()`**: Apply a mapping function and collect.
- **`Collectors.partitioningBy()`**: Partition elements into two groups.

Common `Collectors` Methods and Their Use Cases:

1. **`Collectors.groupingBy()`**: Groups elements of the stream by a classifier function, similar to SQL's `GROUP BY`.

   - **Use Case**: Group elements by a key and return a `Map` with the key and the grouped elements.

   Example:

   ```
   List<String> items = Arrays.asList("apple", "banana", "orange", "apple",
   "orange", "banana", "banana");

   // Group items by their name
   Map<String, Long> groupedItems = items.stream()
       .collect(Collectors.groupingBy(Function.identity(),
   Collectors.counting()));

   System.out.println(groupedItems);
   ```

   Output:

   ```
   {orange=2, banana=3, apple=2}
   ```

   Variants:

   - `groupingBy(classifier)`: Groups elements based on a classifier function.

  - `groupingBy(classifier, downstream)`: Groups elements and applies a downstream collector to each group.
  - `groupingBy(classifier, supplier, downstream)`: Uses a custom `Map` implementation to store the results.

2. **`Collectors.counting()`**: Counts the number of elements in a stream. It's often used as a downstream collector for `groupingBy()` or other collectors.

   - **Use Case**: Count the total number of elements in a stream or the number of elements in each group.

   Example:

```java
List<String> items = Arrays.asList("apple", "banana", "orange", "apple",
"orange");

// Count the total number of elements
long count = items.stream()
    .collect(Collectors.counting());

System.out.println("Total count: " + count);
```

   Output:

```
Total count: 5
```

3. **`Collectors.toList()`**: Collects the elements of the stream into a `List`.

   - **Use Case**: Convert a stream back to a `List`.

   Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Collect the stream into a list
List<Integer> resultList = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());

System.out.println(resultList);
```

   Output:

```
[2, 4]
```

4. `Collectors.toSet()`: Collects the elements of the stream into a `Set`.

   - **Use Case**: Collect unique elements into a set.

   Example:

   ```
   List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 2, 3);

   // Collect the stream into a set
   Set<Integer> resultSet = numbers.stream()
       .collect(Collectors.toSet());

   System.out.println(resultSet);
   ```

   Output:

   ```
   [1, 2, 3, 4, 5]
   ```

5. `Collectors.joining()`: Concatenates the elements of a stream into a single `String`. Optionally, you can provide a delimiter, prefix, and suffix.

   - **Use Case**: Join elements into a string, e.g., CSV format or space-separated values.

   Example:

   ```
   List<String> words = Arrays.asList("apple", "banana", "orange");

   // Join words into a single string
   String result = words.stream()
       .collect(Collectors.joining(", "));

   System.out.println(result);
   ```

   Output:

   ```
   apple, banana, orange
   ```

6. `Collectors.summingInt()`, `summingDouble()`, `summingLong()`: Summing collectors that sum up the values of a specific property in the elements.

   - **Use Case**: Summing properties of objects in a stream.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Sum the numbers in the list
int sum = numbers.stream()
    .collect(Collectors.summingInt(Integer::intValue));

System.out.println(sum);
```

Output:

```
15
```

7. `Collectors.averagingInt()`, `averagingDouble()`, `averagingLong()`: Calculates the average of numeric properties of objects in a stream.

   - **Use Case**: Find the average of numeric values.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Find the average of numbers in the list
double average = numbers.stream()
    .collect(Collectors.averagingInt(Integer::intValue));

System.out.println(average);
```

   Output:

```
3.0
```

8. **`Collectors.maxBy()` and `Collectors.minBy()`**: Find the maximum or minimum element according to a given comparator.

   - **Use Case**: Get the largest or smallest element based on a property.

   Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Find the max element
Optional<Integer> max = numbers.stream()
    .collect(Collectors.maxBy(Comparator.naturalOrder()));
```

```
max.ifPresent(System.out::println);
```

Output:

```
5
```

9. `Collectors.mapping()`: Applies a mapping function before collecting the elements.

   - **Use Case**: Transform the elements before collecting them.

   Example:

```java
List<String> names = Arrays.asList("John", "Jane", "Jack");

// Collect to a list of uppercase names
List<String> upperCaseNames = names.stream()
    .collect(Collectors.mapping(String::toUpperCase, Collectors.toList()));

System.out.println(upperCaseNames);
```

   Output:

```
[JOHN, JANE, JACK]
```

10. `Collectors.partitioningBy()`: Partitions the elements of a stream into two groups based on a predicate. It returns a `Map<Boolean, List<T>>`.

    - **Use Case**: Split elements into two groups (true/false based on a condition).

    Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Partition numbers into even and odd
Map<Boolean, List<Integer>> partitioned = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println(partitioned);
```

    Output:

```
{false=[1, 3, 5], true=[2, 4]}
```