=======Java 8 Features By Madhu Vundavalli ** 9059111409 =====

1. Enhanced Productivity
Java 8 introduced numerous language enhancements that streamline coding and increase productivity. Features like lambda expressions and the Stream API allow for more concise and expressive code, reducing boilerplate and making your codebase cleaner and easier to maintain.

2. Improved Performance
The Stream API enables efficient data processing by leveraging internal iteration and parallelism. This can lead to performance improvements for data-intensive applications, as it allows for operations to be parallelised with minimal effort.

3. Modern Programming Practices
Java 8 features, such as default methods in interfaces, encourage modern programming practices. Default methods enable you to add new functionality to interfaces without breaking existing implementations, thus promoting better design and evolution of APIs.

4. Functional Programming Paradigms
Java 8 introduced functional programming concepts, such as lambda expressions and functional interfaces. These features support more declarative programming styles and facilitate working with higher-order functions, which can lead to more expressive and flexible code.

5. Better Collections Handling
The Stream API provides a powerful way to handle collections, offering operations like filtering, mapping, and reducing in a more intuitive and readable manner compared to traditional loops and conditionals.

6. Date and Time API
Java 8 introduced a new Date and Time API (`java.time` package) that addresses many issues with the old `java.util.Date` and `java.util.Calendar` classes. The new

API provides a more comprehensive and immutable way to handle dates and times, which improves both the clarity and reliability of date-time operations.

7. Concurrency Enhancements
Java 8 includes improvements to concurrency with the addition of CompletableFuture and new methods in the `java.util.concurrent` package. These features provide a more powerful and flexible way to handle asynchronous programming and parallel computation.

8. Compatibility with Modern Libraries and Frameworks
Many modern Java libraries and frameworks leverage Java 8 features. Having a good understanding of these features ensures compatibility with current tools and libraries, and helps you take advantage of their full potential.

9. Career Opportunities
Proficiency in Java 8 features is often a requirement or preference for many job roles in the Java ecosystem. Understanding these features can make you more competitive in the job market and help you stay relevant as a Java developer.

10. Community and Industry Standards
Java 8 features have become industry standards, and understanding them is important for working on collaborative projects and adhering to best practices in the Java community.

In summary, mastering Java 8 features enhances your coding efficiency, keeps your skills up-to-date, and aligns with modern programming practices, making you a more effective and competitive Java developer.

Here are the top 10 Java 8 features:

1. Lambda Expressions
2. Functional Interfaces
3. Stream API

4. Default Methods
5. Optional Class
6. New Date and Time API (java.time)
7. Method References
8. Collectors Class
9. ForEach Method
10. Nashorn JavaScript Engine

1. **Lambda expressions :**

Lambda expressions were introduced in Java 8 to provide a concise way to represent anonymous functions (functions with no name) and to simplify the use of functional interfaces. They allow you to write more readable and concise code, especially when working with functional programming constructs.

Syntax of Lambda Expressions

(parameters) -> expression

Or, if there are multiple statements:

```
(parameters) -> {
   // statements
}
```

Examples

1. Basic Example: Using a lambda expression to implement a functional interface.

```
  @FunctionalInterface
  interface Greeting {
     void greet(String name);
```

```
}

public class LambdaExample {
    public static void main(String[] args) {
        Greeting greeting = (name) -> System.out.println("Hello, " + name);
        greeting.greet("Alice");
    }
}
```

In this example, `Greeting` is a functional interface with a single abstract method `greet()`. The lambda expression `(name) -> System.out.println("Hello, " + name)` provides the implementation.

2. Using Lambda with Collections: Sorting a list using a lambda expression.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("John");
        names.add("Alice");
        names.add("Bob");

        // Sort the list using a lambda expression
        Collections.sort(names, (a, b) -> a.compareTo(b));

        System.out.println(names);
    }
}
```

Here, `Collections.sort()` takes a lambda expression `(a, b) -> a.compareTo(b)` as the comparator for sorting the list.

3. Using Lambda with Streams: Filtering and printing a list of integers.

```
import java.util.Arrays;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Filter and print even numbers using a lambda expression
        numbers.stream()
                .filter(n -> n % 2 == 0)
                .forEach(n -> System.out.println(n));
    }
}
```

In this example, `numbers.stream()` creates a stream from the list, `.filter(n -> n % 2 == 0)` filters the stream to include only even numbers, and `.forEach(n -> System.out.println(n))` prints each even number.

4. Using Lambda with Functional Interfaces: Implementing a custom functional interface with multiple methods.

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}
```

```
public class LambdaExample {
    public static void main(String[] args) {
        // Addition
        MathOperation addition = (a, b) -> a + b;
        System.out.println("Addition: " + addition.operate(5, 3));

        // Multiplication
        MathOperation multiplication = (a, b) -> a * b;
        System.out.println("Multiplication: " + multiplication.operate(5, 3));
    }
}
```

In this example, `MathOperation` is a functional interface with a single abstract method `operate()`. Two lambda expressions implement the `operate()` method for addition and multiplication.

Functional interface :

- A functional interface in Java is an interface that contains only one abstract method.
- It can have multiple default or static methods, but it must have only one abstract method.
- Functional interfaces are used primarily with lambda expressions, method references, and constructor references to facilitate functional programming in Java.

The @FunctionalInterface annotation is used to mark an interface as a functional interface. While this annotation is optional, it provides clarity and ensures that the interface remains functional (i.e., it will generate a compile-time error if more than one abstract method is added).

Example:

```
@FunctionalInterface
interface MyFunctionalInterface
{
 void execute();
}
```

Usage of Functional Interface

1. Lambda Expressions: Functional interfaces provide a target for lambda expressions. They simplify the implementation of single-method interfaces, enabling more concise and readable code.

Example:

```
MyFunctionalInterface myFunc = () -> System.out.println("Executing...");
myFunc.execute();
```

2. Method References: Functional interfaces can be used with method references to refer to a method or constructor without executing it.

Example:

```
MyFunctionalInterface myFunc = System.out::println;
myFunc.execute();
```

3. Streams API: Functional interfaces are heavily used in Java Streams (e.g., Predicate, Function, Consumer, etc.), allowing for functional-style operations on collections.

Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

Importance of Functional Interface

1. Enables Functional Programming: Functional interfaces bridge the gap between object-oriented and functional programming, enabling developers to write concise, expressive code.

2. Lambda Support: By serving as a target for lambda expressions, functional interfaces reduce boilerplate code, making Java code more readable and maintainable.

3. Improves Code Reusability: Since functional interfaces are general-purpose, they can be reused across different parts of an application.

4. Enhances Stream Operations: Functional interfaces are essential in enabling the use of the Streams API for data processing, making operations like filtering, mapping, and reducing more efficient and expressive.

Common Functional Interfaces in Java

- **Runnable:** Represents a task that can be run without any input or output.
- **Callable<T>:** Similar to Runnable, but can return a value and throw a checked exception.
- **Comparator<T>:** Used to compare two objects of type T.
- **Predicate<T>:** Represents a function that takes one argument and returns a boolean.
- **Function<T, R>:** Represents a function that takes one argument of type T and returns a result of type R.

- **Consumer<T>:** Represents a function that takes one argument and returns nothing.
- **Supplier<T>:** Represents a function that takes no arguments and returns a result of type T.

These interfaces are essential in modern Java programming, especially when working with the functional features introduced in Java 8 and beyond.

Here are four example programs demonstrating the use of functional interfaces in Java. Each example uses a different aspect of functional interfaces to show their versatility.

Example 1: Basic Functional Interface

Interface Definition: A simple functional interface with one abstract method.

```
@FunctionalInterface
interface MyFunctionalInterface
 {
   void displayMessage(String message);
}

public class FunctionalInterfaceExample {
   public static void main(String[] args) {
      // Implementing the functional interface using a lambda expression
      MyFunctionalInterface messagePrinter = (message) ->
System.out.println("Message: " + message);
      messagePrinter.displayMessage("Hello, World!");
   }
}
```

Explanation: The `MyFunctionalInterface` interface has one abstract method `displayMessage()`. The lambda expression `(message) -> System.out.println("Message: " + message)` provides the implementation.

Example 2: Functional Interface with Return Value

Interface Definition: A functional interface that takes two integers and returns their sum.

```
@FunctionalInterface
interface Adder {
   int add(int a, int b);
}

public class FunctionalInterfaceExample {
   public static void main(String[] args) {
      // Implementing the functional interface using a lambda expression
      Adder adder = (a, b) -> a + b;
      int result = adder.add(5, 3);
      System.out.println("Sum: " + result);
   }
}
```

Explanation: The `Adder` interface has a method `add()` that takes two integers and returns their sum. The lambda expression `(a, b) -> a + b` provides the implementation.

Example 3: Functional Interface with Multiple Implementations

Interface Definition: A functional interface with a method to compare two strings.

```
@FunctionalInterface
interface StringComparator {
```

```
    boolean compare(String a, String b);
}

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Implementing the functional interface using lambda expressions
        StringComparator equalsComparator = (a, b) -> a.equals(b);
        StringComparator lengthComparator = (a, b) -> a.length() == b.length();

        System.out.println("Are 'hello' and 'hello' equal? " +
equalsComparator.compare("hello", "hello"));
        System.out.println("Do 'hello' and 'world' have the same length? " +
lengthComparator.compare("hello", "world"));
    }
}
```

Explanation: The `StringComparator` interface has a method `compare()` that compares two strings. Two lambda expressions provide different implementations: one for checking equality and the other for comparing lengths.

Example 4: Functional Interface with Method Reference

Interface Definition: A functional interface that performs an operation on a string.

```
@FunctionalInterface
interface StringOperation {
    void operate(String s);
}

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Using a method reference to implement the functional interface
```

```
        StringOperation printOperation = System.out::println;
        printOperation.operate("Hello, method reference!");
    }
}
```

Explanation: The `StringOperation` interface has a method `operate()` that performs an operation on a string. The method reference `System.out::println` is used to provide the implementation, printing the string to the console.

These examples illustrate different ways to use functional interfaces, from simple lambda expressions to method references, showcasing the flexibility and power of functional programming in Java.

Stream api :

Stream API: Definition, Usage, Importance, and Advantages

Definition

The Java 8 Stream API is a powerful feature introduced to facilitate functional-style operations on sequences of elements, such as collections. It allows for the processing of data in a declarative manner, meaning that the operations focus on "what" should be done rather than "how" it should be executed. Streams support operations like filtering, mapping, reducing, and collecting, making data manipulation more efficient and readable.

Usage

Streams are primarily used for processing collections of objects, such as lists, sets, and maps, in a functional style. Here is a basic usage example:

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

```
// Using Stream API to filter and collect names starting with 'J'
List<String> filteredNames = names.stream()
                        .filter(name -> name.startsWith("J"))
                        .collect(Collectors.toList());

System.out.println(filteredNames); // Output: [John, Jane, Jack]
```

In this example, the `stream()` method converts the collection into a stream, the `filter()` method applies a condition, and the `collect()` method gathers the filtered results back into a list.

Importance

The Stream API is important because it simplifies data processing by providing a concise and expressive syntax. It also enables developers to write cleaner, more maintainable code. Furthermore, the Stream API supports parallel operations, making it easier to take advantage of multi-core processors for better performance.

Advantages

1. Concise and Readable Code: The Stream API reduces boilerplate code, making data manipulation tasks easier to read and write.

2. Functional Programming: It encourages a functional programming approach, allowing developers to chain operations in a pipeline-like fashion.

3. Parallel Processing: Streams can be executed in parallel with minimal code changes, improving performance in multi-core environments.

4. Lazy Evaluation: Stream operations are performed lazily, meaning they are only executed when necessary. This improves efficiency, especially when working with large datasets.

5. Improved Performance: By using parallel streams and lazy evaluation, the Stream API can significantly enhance performance in data processing tasks.

The Stream API in Java 8 represents a significant step forward in simplifying and optimising data processing within Java applications.

Stream api example :

```
import java.util.*;
import java.util.stream.*;
class Employee {
   private int id;
   private String name;
   private int age;
   private String department;
   private double salary;

   public Employee(int id, String name, int age, String department, double salary) {
      this.id = id;
      this.name = name;
      this.age = age;
      this.department = department;
      this.salary = salary;
   }

   public int getId() { return id; }
   public String getName() { return name; }
   public int getAge() { return age; }
   public String getDepartment() { return department; }
   public double getSalary() { return salary; }

   @Override
```

```
    public String toString() {
        return String.format("Employee{id=%d, name='%s', age=%d,
department='%s', salary=%.2f}",
                id, name, age, department, salary);
    }
}

public class StreamAPIExample {

    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee(1, "Alice", 30, "HR", 50000),
            new Employee(2, "Bob", 24, "Engineering", 75000),
            new Employee(3, "Charlie", 28, "Engineering", 60000),
            new Employee(4, "David", 35, "HR", 90000),
            new Employee(5, "Eve", 40, "Management", 120000)
        );

        // Filtering employees older than 30
        List<Employee> olderEmployees = employees.stream()
            .filter(e -> e.getAge() > 30)
            .collect(Collectors.toList());
        System.out.println("Employees older than 30: " + olderEmployees);

        // Mapping employees' names
        List<String> employeeNames = employees.stream()
            .map(Employee::getName)
            .collect(Collectors.toList());
        System.out.println("Employee names: " + employeeNames);

        // FlatMap to merge lists of employees
        List<List<Employee>> employeeGroups = Arrays.asList(employees,
employees);
```

```
List<Employee> allEmployees = employeeGroups.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
System.out.println("All employees from flatMap: " + allEmployees);

// Distinct departments
Set<String> departments = employees.stream()
    .map(Employee::getDepartment)
    .distinct()
    .collect(Collectors.toSet());
System.out.println("Distinct departments: " + departments);

// Sorting employees by salary
List<Employee> sortedBySalary = employees.stream()
    .sorted((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()))
    .collect(Collectors.toList());
System.out.println("Employees sorted by salary: " + sortedBySalary);

// Limiting to top 3 highest paid employees
List<Employee> topPaidEmployees = employees.stream()
    .sorted((e1, e2) -> Double.compare(e2.getSalary(), e1.getSalary()))
    .limit(3)
    .collect(Collectors.toList());
System.out.println("Top 3 highest paid employees: " + topPaidEmployees);

// Skipping the first 2 employees
List<Employee> skipTwo = employees.stream()
    .skip(2)
    .collect(Collectors.toList());
System.out.println("Employees after skipping first two: " + skipTwo);

// Collecting employees into a list
List<Employee> employeeList = employees.stream()
```

```
        .collect(Collectors.toList());
    System.out.println("All employees: " + employeeList);

    // Grouping employees by department
    Map<String, List<Employee>> employeesByDept = employees.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
    System.out.println("Employees grouped by department: " +
employeesByDept);

    // Counting employees in Engineering department
    long countEngineering = employees.stream()
        .filter(e -> "Engineering".equals(e.getDepartment()))
        .count();
    System.out.println("Number of employees in Engineering: " +
countEngineering);

    // Finding the first employee
    Optional<Employee> firstEmployee = employees.stream()
        .findFirst();
    System.out.println("First employee: " + firstEmployee.orElse(null));

    // Finding any employee older than 30
    Optional<Employee> anyOlderEmployee = employees.stream()
        .filter(e -> e.getAge() > 30)
        .findAny();
    System.out.println("Any employee older than 30: " +
anyOlderEmployee.orElse(null));

    // Checking if all employees are older than 20
    boolean allOlderThan20 = employees.stream()
        .allMatch(e -> e.getAge() > 20);
    System.out.println("All employees older than 20: " + allOlderThan20);
```

```
    // Checking if any employee is in HR department
    boolean anyInHR = employees.stream()
       .anyMatch(e -> "HR".equals(e.getDepartment()));
    System.out.println("Any employee in HR department: " + anyInHR);

    // Checking if no employee is under 18
    boolean noneUnder18 = employees.stream()
       .noneMatch(e -> e.getAge() < 18);
    System.out.println("No employees under 18: " + noneUnder18);

    // Reducing to get total salary of all employees
    double totalSalary = employees.stream()
       .map(Employee::getSalary)
       .reduce(0.0, Double::sum);
    System.out.println("Total salary of all employees: " + totalSalary);

    // Finding employee with max salary
    Optional<Employee> maxSalaryEmployee = employees.stream()
       .reduce((e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2);
    System.out.println("Employee with highest salary: " +
maxSalaryEmployee.orElse(null));

    // Parallel Stream for filtering high salary employees
    List<Employee> parallelResult = employees.parallelStream()
       .filter(e -> e.getSalary() > 60000)
       .collect(Collectors.toList());
    System.out.println("Parallel stream result (salary > 60000): " + parallelResult);

    // Primitive stream for generating a range of numbers
    System.out.println("Numbers from 20 to 29:");
    IntStream.range(20, 30).forEach(System.out::println);

    // Stream from values
```

```
        System.out.println("Stream from values:");
        Stream.of("Alice", "Bob", "Charlie").forEach(System.out::println);

        // Stream iteration
        System.out.println("Stream iteration (first 5 even numbers):");
        Stream.iterate(0, n -> n + 2).limit(5).forEach(System.out::println);

        // Real-world example: Average salary in Engineering department for employees older than 25
        double averageSalary = employees.stream()
            .filter(e -> "Engineering".equals(e.getDepartment()))
            .filter(e -> e.getAge() > 25)
            .mapToDouble(Employee::getSalary)
            .average()
            .orElse(0.0);
        System.out.println("Average salary in Engineering for employees older than 25: " + averageSalary);
    }
}
```

==========

Default methods :

Default methods are a feature introduced in Java 8 that allows interfaces to have methods with a body. This means that interfaces can now contain concrete methods (methods with implementation), not just abstract methods. Default methods are declared using the `default` keyword.

Usage

1. Defining a Default Method:

You define a default method in an interface using the `default` keyword followed by the method signature and body.

```
interface MyInterface {
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
}
```

2. Implementing an Interface with Default Methods:

Classes implementing the interface inherit the default method implementation.

```
class MyClass implements MyInterface {
    // No need to override defaultMethod unless custom behaviour is required
}
```

3. Overriding Default Methods:

If a class wants to provide its own implementation for a default method, it can override it.

```
class MyClass implements MyInterface {
    @Override
    public void defaultMethod() {
        System.out.println("Overridden default method.");
    }
}
```

Importance

1. Backward Compatibility:

Default methods allow you to add new methods to interfaces without breaking the existing implementations of those interfaces. This is crucial for maintaining backward compatibility in libraries and frameworks.

2. Code Reusability:

Default methods enable code reusability by allowing common functionality to be provided in the interface itself. This avoids the need for classes to repeatedly implement the same functionality.

3. Enhanced Interfaces:

They enhance the capabilities of interfaces by allowing them to provide default behaviour, making interfaces more versatile and powerful.

Advantages

1. Non-breaking Changes:

Default methods enable developers to add new methods to an interface without requiring all implementing classes to provide an implementation immediately. This is particularly useful in evolving APIs and frameworks.

2. Reduced Boilerplate Code:

By providing default implementations, you can avoid writing repetitive code in each class that implements the interface, thus reducing boilerplate and improving maintainability.

3. Multiple Inheritance Support:

Default methods offer a way to achieve multiple inheritance of behaviour in Java, as a class can implement multiple interfaces with default methods and override them if needed.

4. Enhanced API Design:

They allow for better API design by enabling interfaces to offer default implementations of methods that can be used directly by the implementing classes or overridden as required.

Here is a complete example demonstrating default methods:

```
interface Vehicle {
    // Default method
    default void start() {
        System.out.println("Vehicle is starting");
    }

    // Abstract method
    void stop();
}

class Car implements Vehicle {
    @Override
    public void stop() {
        System.out.println("Car is stopping");
    }

    // Overriding the default method (optional)
    @Override
    public void start() {
        System.out.println("Car is starting with a key");
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        Vehicle myCar = new Car();
        myCar.start(); // Calls overridden method
        myCar.stop();  // Calls implemented abstract method
    }
}
```

In this example, the `Vehicle` interface defines a default method `start` and an abstract method `stop`. The `Car` class implements the `Vehicle` interface, providing its own implementation for `stop` and overriding the default `start` method.

Example program :

```
// Define an interface for user accounts with common operations

interface UserAccount {
    // Default method to activate a user account
    default void activate() {
        System.out.println("User account is activated");
    }

    // Default method to deactivate a user account
    default void deactivate() {
        System.out.println("User account is deactivated");
    }

    // Abstract method to be implemented by specific types of user accounts
    void login();
}

// Implement the UserAccount interface for a regular user account
class RegularUser implements UserAccount {
```

```
    @Override
    public void login() {
        System.out.println("Regular user is logging in");
    }

    // Optionally override the default activate method for specific behaviour
    @Override
    public void activate() {
        System.out.println("Regular user account is now active");
    }
}

// Implement the UserAccount interface for an admin account
class AdminUser implements UserAccount {
    @Override
    public void login() {
        System.out.println("Admin user is logging in");
    }

    // Optionally override the default deactivate method for specific behaviour
    @Override
    public void deactivate() {
        System.out.println("Admin user account is suspended");
    }
}

// Main class to test the user accounts
public class Main {
    public static void main(String[] args) {
        // Create instances of RegularUser and AdminUser
        UserAccount regularUser = new RegularUser();
        UserAccount adminUser = new AdminUser();
```

```
        // Test RegularUser operations
        System.out.println("Regular User Operations:");
        regularUser.activate();       // Output: Regular user account is now active
        regularUser.login();          // Output: Regular user is logging in
        regularUser.deactivate();     // Output: User account is deactivated

        // Test AdminUser operations
        System.out.println("\nAdmin User Operations:");
        adminUser.activate();         // Output: User account is activated
        adminUser.login();            // Output: Admin user is logging in
        adminUser.deactivate();       // Output: Admin user account is suspended
    }
}
```

Code Explanation:

1. Interface Definition (UserAccount):

default void activate(): Provides a default implementation for activating a user account.

default void deactivate(): Provides a default implementation for deactivating a user account.

void login(): An abstract method that must be implemented by all user account types to define their login behaviour.

2. Class Implementation (RegularUser):

public void login(): Implements the specific login behaviour for a regular user.

public void activate(): Optionally overrides the default activate method to provide a more specific activation message for regular users.

3. Class Implementation (AdminUser):

public void login(): Implements the specific login behaviour for an admin user.

public void deactivate(): Optionally overrides the default deactivate method to provide a specific deactivation message for admin users.

4. Main Class (Main):
Creates instances of RegularUser and AdminUser and demonstrates their operations, showing both default and overridden behaviours.

Optional :

The `Optional` class in Java 8 is a container object that may or may not contain a value. It is used to represent the presence or absence of a value without using null references.

Usage

`Optional` is used to explicitly handle cases where a value might be absent. Instead of returning null, methods can return an `Optional` to indicate that the result might be missing. This allows developers to handle the absence of a value more gracefully.

Importance

Avoids NullPointerException: Provides a way to handle missing values without resorting to null, thereby reducing the risk of `NullPointerException`.
Clarifies Intent: Makes it clear when a value may be absent, improving code readability and maintenance.

Advantages :

1. Reduces Null Checks: Eliminates the need for explicit null checks and provides a more elegant way to handle optional values.
2. Improves Code Safety: Helps in writing safer code by enforcing checks on the presence of a value before accessing it.

3. Supports Functional Programming: Facilitates functional programming techniques by supporting operations like `map`, `flatMap`, and `filter`.
4. Enhances API Design: Provides a clear and expressive way to indicate the possibility of missing values in method signatures, improving the design of APIs.

Example program :

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {

      // Creating an Optional that contains a value
      Optional<String> optionalValue = Optional.of("Hello, World!");

      // Check if the Optional contains a value
      // This is equivalent to checking if a value is null
      if (optionalValue.isPresent()) {
         // If the value is present, get it and print it
         System.out.println("Value is present: " + optionalValue.get());
      }

      // Using orElse() to provide a default value if Optional is empty
      // This helps to avoid returning null or performing null checks
      String value = optionalValue.orElse("Default Value");
      System.out.println("Value: " + value); // Prints the actual value since it's
present

      // Creating an empty Optional
      Optional<String> emptyOptional = Optional.empty();

      // Since the Optional is empty, it will return the default value
```

```
        String emptyValue = emptyOptional.orElse("Default Value");
        System.out.println("Empty Optional Value: " + emptyValue);

        // Using ifPresent() to perform an action only if a value is present
        // This is a more concise way to handle optional values
        optionalValue.ifPresent(val -> System.out.println("Value via ifPresent: " +
val));

        // Using map() to transform the value if present
        // The function inside map() is applied only if the Optional contains a value
        Optional<String> transformedValue = optionalValue.map(val ->
val.toUpperCase());
        System.out.println("Transformed Value: " + transformedValue.orElse("No
Value"));
    }
}
```

Explanation:

1. Optional Creation:

Optional.of("Hello, World!"): Creates an Optional that contains a non-null value.
Optional.empty(): Creates an empty Optional with no value inside.

2. isPresent():

Checks if the Optional contains a value, similar to a null check but more explicit and safer.

3. get():

Retrieves the value if present. If the Optional is empty, calling get() would throw a NoSuchElementException.

4. orElse():

Provides a default value in case the Optional is empty, avoiding null returns.

5. ifPresent():

Executes the provided lambda expression if the Optional contains a value, providing a concise way to handle optional values.

6. map():

Transforms the value inside the Optional if it is present, allowing you to apply functions directly to the contained value.

This example demonstrates the primary features of the Optional class in Java 8, showing how it can be used to write safer and more readable code.

New Date and Time API (java.time) :

The java.time package in Java 8 introduced a new Date and Time API to overcome the limitations of the previous java.util.Date and java.util.Calendar classes. This new API is part of the Java 8 release and provides a more consistent, comprehensive, and user-friendly approach to handling date and time.

Usage

The java.time package includes various classes such as LocalDate, LocalTime, LocalDateTime, ZonedDateTime, and Instant. These classes are designed to represent dates, times, date-times with time zones, and timestamps. Here are some common usage scenarios:

1. Working with Local Dates:

LocalDate represents a date without a time zone, such as "2024-08-30".

2. Working with Local Times:

LocalTime represents a time without a date or time zone, such as "10:15:30".

3. Working with Date and Time:

LocalDateTime combines both date and time without a time zone.

4. Handling Time Zones:

ZonedDateTime handles date and time with a time zone.

5. Timestamps:

Instant represents a timestamp, which is a point on the timeline.

Importance

Immutability: All classes in the java.time package are immutable and thread-safe, which prevents bugs related to mutable date-time objects in multithreaded environments.

Precision: The new API offers more precision and flexibility in handling date and time, including nanosecond precision.

Clear and Consistent API: The java.time package provides a clear and consistent API for working with dates and times, avoiding the confusion and inconsistencies of the old Date and Calendar classes.

ISO-8601 Standard: The new API follows the ISO-8601 standard by default, making it easier to work with international date-time formats.

Advantages

1. Improved Readability and Usability: The API provides a more intuitive and readable approach to date and time manipulation, reducing the complexity of working with dates and times.

2. Eliminates Common Pitfalls: The new API resolves issues with time zone calculations, daylight saving time, and leap years, which were problematic in older APIs.

3. Better Performance: Due to immutability, instances of date and time objects are safer and can be shared between threads without synchronization, leading to better performance in multithreaded applications.

4. Support for Different Calendars: The java.time package supports different calendar systems, making it versatile for international applications.

5. Easy Time Zone Handling: The API provides robust support for time zones and offsets, simplifying time zone conversions and ensuring accuracy in global applications.
6. Fluent API: The fluent API design allows method chaining, making date-time manipulation more concise and readable.
7. Compatibility with Legacy Code: The new API provides methods to convert between the old Date/Calendar classes and the new classes, facilitating migration from older code.

Overall, the java.time package is a major improvement in Java 8, offering a more robust, flexible, and user-friendly way to handle date and time operations.

Example program :

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.util.Set;

public class MaxDateTimeScenariosExample {
   public static void main(String[] args) {

      // Example 1: LocalDate - Represents a date without time (e.g., 2024-08-30)
      LocalDate today = LocalDate.now(); // Get the current date
      System.out.println("Today's Date: " + today);
      // Output: Today's Date: 2024-08-30 (example output, will vary based on the current date)

      LocalDate specificDate = LocalDate.of(2024, Month.DECEMBER, 25); // Create a specific date
      System.out.println("Specific Date: " + specificDate);
      // Output: Specific Date: 2024-12-25
```

// Adding/Subtracting days to a date

LocalDate nextWeek = today.plusDays(7); // Add 7 days to the current date

System.out.println("Date Next Week: " + nextWeek);

// Output: Date Next Week: 2024-09-06 (example output, will vary based on current date)

LocalDate lastMonth = today.minusMonths(1); **// Subtract 1 month from the current date**

System.out.println("Date Last Month: " + lastMonth);

// Output: Date Last Month: 2024-07-30 (example output, will vary based on current date)

// Checking if a date is before or after another date

boolean isBefore = today.isBefore(specificDate); **// Check if today is before specificDate**

System.out.println("Is today before 25th Dec 2024? " + isBefore);

// Output: Is today before 25th Dec 2024? true (example output)

// Example 2: LocalTime - Represents a time without a date (e.g., 10:15:30)

LocalTime now = LocalTime.now(); // Get the current time

System.out.println("Current Time: " + now);

// Output: Current Time: 14:45:12 (example output, will vary based on the current time)

LocalTime specificTime = LocalTime.of(14, 30, 15); **// Create a specific time**

System.out.println("Specific Time: " + specificTime);

// Output: Specific Time: 14:30:15

// Adding/Subtracting time

LocalTime twoHoursLater = now.plusHours(2); **// Add 2 hours to the current time**

System.out.println("Time Two Hours Later: " + twoHoursLater);

// Output: Time Two Hours Later: 16:45:12 (example output, will vary based on current time)

LocalTime thirtyMinutesAgo = now.minusMinutes(30); **// Subtract 30 minutes from the current time**
System.out.println("Time Thirty Minutes Ago: " + thirtyMinutesAgo);
// Output: Time Thirty Minutes Ago: 14:15:12 (example output, will vary based on current time)

// Comparing times
boolean isAfter = now.isAfter(specificTime); **// Check if the current time is after specificTime**
System.out.println("Is the current time after 14:30:15? " + isAfter);
// Output: Is the current time after 14:30:15? true (example output)

// Example 3: LocalDateTime - Combines date and time without a time zone (e.g., 2024-08-30T10:15:30)
LocalDateTime currentDateTime = LocalDateTime.now(); // Get the current date and time
System.out.println("Current Date and Time: " + currentDateTime);
// Output: Current Date and Time: 2024-08-30T14:45:12 (example output, will vary)

LocalDateTime specificDateTime = LocalDateTime.of(2024, Month.DECEMBER, 25, 14, 30, 15); // Create a specific date and time
System.out.println("Specific Date and Time: " + specificDateTime);
// Output: Specific Date and Time: 2024-12-25T14:30:15

// Adding/Subtracting days, hours, minutes, etc.
LocalDateTime tomorrow = currentDateTime.plusDays(1); // Add 1 day to the current date and time
System.out.println("Date and Time Tomorrow: " + tomorrow);

// Output: Date and Time Tomorrow: 2024-08-31T14:45:12 (example output)

LocalDateTime lastYear = currentDateTime.minusYears(1); **// Subtract 1 year from the current date and time**
System.out.println("Date and Time Last Year: " + lastYear);
// Output: Date and Time Last Year: 2023-08-30T14:45:12 (example output)

// Example 4: ZonedDateTime - Represents date and time with a time zone (e.g., 2024-08-30T10:15:30+01:00[Europe/London])
ZonedDateTime currentZonedDateTime = ZonedDateTime.now(); // Get the current date, time, and time zone
System.out.println("Current Zoned Date and Time: " + currentZonedDateTime);
// Output: Current Zoned Date and Time: 2024-08-30T14:45:12+01:00[Europe/London] (example output, will vary)

// Getting available time zones
Set<String> allZones = ZoneId.getAvailableZoneIds(); // Get all available time zones
System.out.println("Available Time Zones: " + allZones.size());
// Output: Available Time Zones: 600+ (depends on the system and JDK)

// Example 5: Instant - Represents a timestamp (e.g., 2024-08-30T10:15:30Z)
Instant currentInstant = Instant.now(); // Get the current timestamp (UTC time)
System.out.println("Current Instant: " + currentInstant);
// Output: Current Instant: 2024-08-30T13:45:12Z (example output, will vary)

// Adding/Subtracting seconds, minutes, etc.
Instant tenSecondsLater = currentInstant.plusSeconds(10); **// Add 10 seconds to the current timestamp**

System.out.println("Instant Ten Seconds Later: " + tenSecondsLater);
// Output: Instant Ten Seconds Later: 2024-08-30T13:45:22Z (example output)

Instant fiveMinutesAgo = currentInstant.minus(5, ChronoUnit.MINUTES); **// Subtract 5 minutes from the current timestamp**
System.out.println("Instant Five Minutes Ago: " + fiveMinutesAgo);
// Output: Instant Five Minutes Ago: 2024-08-30T13:40:12Z (example output)

// Example 6: Period - Represents a date-based amount of time (e.g., "2 years, 3 months, and 4 days")
Period period = Period.between(LocalDate.of(2022, Month.JANUARY, 1), today); **// Calculate the period between two dates**
System.out.println("Period Between 01-Jan-2022 and Today: " + period.getYears() + " years, " + period.getMonths() + " months, " + period.getDays() + " days");
// Output: Period Between 01-Jan-2022 and Today: 2 years, 7 months, 29 days

// Example 7: Duration - Represents a time-based amount of time (e.g., "2 hours, 30 minutes")
Duration duration = Duration.between(LocalTime.of(9, 0), now); // Calculate the duration between two times
System.out.println("Duration Between 09:00 and Now: " + duration.toHours() + " hours, " + duration.toMinutes() % 60 + " minutes");
// Output: Duration Between 09:00 and Now: 5 hours, 45 minutes (example output, will vary based on current time)

// Example 8: Formatting and Parsing Date and Time
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd-MM-yyyy"); // Define a custom date format

String formattedDate = today.format(dateFormatter); // Format the current date

System.out.println("Formatted Date: " + formattedDate);

// Output: Formatted Date: 30-08-2024 (example output, will vary based on current date)

DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("HH:mm:ss"); // Define a custom time format

String formattedTime = now.format(timeFormatter); **// Format the current time**

System.out.println("Formatted Time: " + formattedTime);

// Output: Formatted Time: 14:45:12 (example output, will vary based on current time)

// Parsing strings to dates/times

LocalDate parsedDate = LocalDate.parse("25-12-2024", dateFormatter);

// Parse a date string using the custom format

System.out.println("Parsed Date: " + parsedDate);

// Output: Parsed Date: 2024-12-25

LocalTime parsedTime = LocalTime.parse("14:30:15", timeFormatter);

// Parse a time string using the custom format

System.out.println("Parsed Time: " + parsedTime);

// Output: Parsed Time: 14:30:15

// Example 9: Converting Between Types

LocalDateTime dateTimeFromDateAndTime = LocalDateTime.of(today, now); **// Convert LocalDate and LocalTime to LocalDateTime**

System.out.println("LocalDateTime from LocalDate and LocalTime: " + dateTimeFromDateAndTime);

// Output: LocalDateTime from LocalDate and LocalTime: 2024-08-30T14:45:12 (example output)

```
        Instant instantFromDateTime = currentDateTime.toInstant(ZoneOffset.UTC);
// Convert LocalDateTime to Instant
        System.out.println("Instant from LocalDateTime: " + instantFromDateTime);
        // Output: Instant from LocalDateTime: 2024-08-30T13:45:12Z (example
output)

        ZonedDateTime zonedDateTimeFromInstant =
instantFromDateTime.atZone(ZoneId.of("America/New_York")); // Convert
Instant to ZonedDateTime
        System.out.println("ZonedDateTime from Instant: " +
zonedDateTimeFromInstant);
        // Output: ZonedDateTime from Instant:
2024-08-30T09:45:12-04:00[America/New_York]

            // Example 11: Leap Year
        boolean isLeapYear = specificDate.isLeapYear(); // Check if the year is a leap
year
        System.out.println("Is 2024 a Leap Year? " + isLeapYear);
        // Output: Is 2024 a Leap Year? true

    }
}
```

Method references :

Method references in Java 8 provide a shorthand syntax for calling a method directly by referring to it with the help of the :: operator. Method references allow you to use existing methods as lambda expressions, enhancing code readability and reducing boilerplate.

Usage:

Method references can be used in various contexts, typically with functional interfaces (interfaces with a single abstract method). They are commonly employed in lambda expressions when you need to pass the reference of a method instead of defining a lambda body.

There are four types of method references:

1.Static Method Reference: Refers to a static method.
Syntax: ClassName::staticMethodName
Example: Math::abs

2.Instance Method Reference of a Particular Object: Refers to an instance method of a particular object.
Syntax: instance::instanceMethodName
Example: String::length

3.Instance Method Reference of an Arbitrary Object of a Particular Type: Refers to an instance method of an arbitrary object of a specific type.
Syntax: ClassName::instanceMethodName
Example: String::toUpperCase

4.Constructor Reference: Refers to a constructor to create a new object.
Syntax: ClassName::new
Example: ArrayList::new

Importance:
Method references simplify the process of using lambda expressions by directly referencing existing methods, improving code clarity and reducing duplication. They encourage the use of functional programming concepts, making the code more concise and expressive.

Advantages:

1.Code Readability: Method references reduce the verbosity of lambda expressions, making the code more readable and easier to understand.

2.Reusability: They promote reusability by allowing the direct use of existing methods without redefining logic in lambda expressions.

3.Conciseness: Method references eliminate unnecessary boilerplate code, enabling developers to write more concise and compact code.

4.Encourages Functional Programming: By using method references with functional interfaces, developers are encouraged to adopt a more functional programming approach, which is beneficial in scenarios like stream processing and event handling.

5.Maintainability: With reduced code complexity, method references make the code easier to maintain, debug, and extend in the future.

Method references, introduced in Java 8, provide a powerful mechanism to simplify code by leveraging existing methods in a more concise and functional manner.

Example program

Here's the example code with output comments added for better understanding:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.function.Supplier;

public class MethodReferencesExample {
```

```
    public static void main(String[] args) {

        // Example 1: Static Method Reference
        // Refers to a static method of a class
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using lambda expression
        names.forEach(name -> System.out.println(name.toUpperCase()));
        // Output:
        // ALICE
        // BOB
        // CHARLIE

        // Using method reference
        System.out.println("Static Method Reference:");
        names.forEach(MethodReferencesExample::printUpperCase); // MethodReferencesExample::printUpperCase
        // Output:
        // ALICE
        // BOB
        // CHARLIE

        // Example 2: Instance Method Reference of a Particular Object
        // Refers to an instance method of a particular object
        MethodReferencesExample example = new MethodReferencesExample();

        // Using lambda expression
        names.forEach(name -> example.printWithPrefix(name, "Hello, "));
        // Output:
        // Hello, Alice
        // Hello, Bob
        // Hello, Charlie
```

```
        // Using method reference
        System.out.println("Instance Method Reference:");
        names.forEach(example::printWithPrefix); // example::printWithPrefix
        // Output:
        // Hello, Alice
        // Hello, Bob
        // Hello, Charlie

        // Example 3: Instance Method Reference of an Arbitrary Object of a Particular Type
        // Refers to an instance method of an arbitrary object of a particular type
        // Using lambda expression
        names.sort((a, b) -> a.compareTo(b));

        // Using method reference
        System.out.println("Instance Method Reference of Arbitrary Object:");
        names.sort(String::compareTo); // String::compareTo
        System.out.println(names);
        // Output: [Alice, Bob, Charlie]

        // Example 4: Constructor Reference
        // Refers to a constructor to create new instances
        // Using lambda expression
        Supplier<ArrayList<String>> supplierLambda = () -> new ArrayList<>();
        ArrayList<String> listLambda = supplierLambda.get();

        // Using method reference
        Supplier<ArrayList<String>> supplierMethodRef = ArrayList::new; // ArrayList::new
        ArrayList<String> listMethodRef = supplierMethodRef.get();

        System.out.println("Constructor Reference:");
        System.out.println(listLambda); // Output: []
```

```
        System.out.println(listMethodRef); // Output: []

        // Example 5: Method Reference with a Functional Interface
        // Using method reference in a functional interface context
        Function<String, Integer> stringLengthLambda = str -> str.length();
        System.out.println("Length of 'Hello': " + stringLengthLambda.apply("Hello"));
        // Output: Length of 'Hello': 5

        // Using method reference
        Function<String, Integer> stringLengthMethodRef = String::length; //
String::length
        System.out.println("Length of 'Hello': " +
stringLengthMethodRef.apply("Hello"));
        // Output: Length of 'Hello': 5
    }

    // Static method for static method reference example
    public static void printUpperCase(String name) {
        System.out.println(name.toUpperCase());
    }

    // Instance method for instance method reference example
    public void printWithPrefix(String name, String prefix) {
        System.out.println(prefix + name);
    }
}
```

Output Breakdown:

1. Static Method Reference:

Output:

```
ALICE
```

BOB
CHARLIE

Explanation: Each name in the list is converted to uppercase using the static method `printUpperCase`.

2. Instance Method Reference of a Particular Object:

Output:

Hello, Alice
Hello, Bob
Hello, Charlie

Explanation: Each name in the list is printed with the prefix "Hello, " using the instance method `printWithPrefix`.

3. Instance Method Reference of an Arbitrary Object of a Particular Type:

Output:

[Alice, Bob, Charlie]

Explanation: The list is sorted using the `compareTo` method of `String`, resulting in a list sorted in natural order.

4. Constructor Reference:

Output:

[]
[]

Explanation: Two empty `ArrayList` instances are created, one using a lambda expression and the other using a constructor reference.

5. Method Reference with a Functional Interface:

Output:

```
Length of 'Hello': 5
Length of 'Hello': 5
```

Explanation: The length of the string "Hello" is calculated using the `length` method of `String`.

This code demonstrates how method references can be used in place of lambda expressions, making the code more concise and readable.

Collectors class :

The Collectors class in Java is a utility class in the java.util.stream package that provides various implementations of the Collector interface. A Collector is used to accumulate elements of a stream into a summary result, such as a List, Set, Map, or even a single value.

Usage:
The Collectors class provides static factory methods that return various Collector implementations. These implementations are used in conjunction with the Stream API to collect results from a stream into collections or other forms of output.

Important Methods in Collectors Class:
1. toList(): Collects the elements of the stream into a List.
2. toSet(): Collects the elements of the stream into a Set.
3. toMap(): Collects the elements of the stream into a Map with custom key and value mappings.
4. joining(): Concatenates the elements of the stream into a single String.
5. groupingBy(): Groups the elements of the stream by a classifier function into a Map.

6. partitioningBy(): Partitions the elements of the stream into two groups based on a predicate.
7. counting(): Counts the number of elements in the stream.
8. summarizingInt(), summarizingDouble(), summarizingLong(): Collects statistics, such as count, sum, min, average, and max, for numerical values.
9. reducing(): Performs a reduction on the elements of the stream using an associative accumulation function.

Importance:
The Collectors class is crucial for transforming and aggregating stream data into various forms. It simplifies the process of collecting results from streams and is integral to functional-style programming in Java. By providing predefined collectors, it reduces the need for manual implementation of common collection operations.

Advantages:
1. Conciseness: Simplifies the collection of stream results into different formats (e.g., lists, sets, maps) with concise and readable code.
2. Flexibility: Offers a wide range of predefined collectors, enabling complex data aggregation and transformation operations with minimal code.
3. Readability: Makes code more readable and expressive by providing high-level operations for collecting and processing stream data.
4. Functionality: Supports various aggregation and transformation operations, such as grouping, partitioning, and summarizing, which are commonly used in data processing.
5. Performance: Leveraging stream collectors can lead to optimized performance for data processing and aggregation tasks.

Example program :

```
import java.util.*;
import java.util.stream.Collectors;
```

```
public class CollectorsExample {

    public static void main(String[] args) {
        // Sample list of names
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Alice");

        // Example 1: Collecting to a List
        List<String> namesList = names.stream().collect(Collectors.toList());
        System.out.println("List: " + namesList);
        // Output: List: [Alice, Bob, Charlie, David, Alice]

        // Example 2: Collecting to a Set (removes duplicates)
        Set<String> namesSet = names.stream().collect(Collectors.toSet());
        System.out.println("Set: " + namesSet);
        // Output: Set: [Alice, Bob, Charlie, David]

        // Example 3: Collecting to a Map
        Map<String, Integer> namesMap =
names.stream().collect(Collectors.toMap(name -> name, String::length));
        System.out.println("Map: " + namesMap);
        // Output: Map: {Alice=5, Bob=3, Charlie=7, David=5}

        // Example 4: Joining elements into a single String
        String joinedNames = names.stream().collect(Collectors.joining(", "));
        System.out.println("Joined: " + joinedNames);
        // Output: Joined: Alice, Bob, Charlie, David, Alice

        // Example 5: Grouping elements by length
        Map<Integer, List<String>> groupedByLength =
names.stream().collect(Collectors.groupingBy(String::length));
        System.out.println("Grouped by length: " + groupedByLength);
        // Output: Grouped by length: {3=[Bob], 5=[Alice, David, Alice], 7=[Charlie]}
```

```
        // Example 6: Partitioning elements by length (less than 5 characters)
        Map<Boolean, List<String>> partitionedByLength =
names.stream().collect(Collectors.partitioningBy(name -> name.length() < 5));
        System.out.println("Partitioned by length < 5: " + partitionedByLength);
        // Output: Partitioned by length < 5: {false=[Alice, Charlie, David, Alice],
true=[Bob]}

        // Example 7: Counting elements
        long count = names.stream().collect(Collectors.counting());
        System.out.println("Count: " + count);
        // Output: Count: 5

        // Example 8: Summarizing length of names
        IntSummaryStatistics stats =
names.stream().collect(Collectors.summarizingInt(String::length));
        System.out.println("Summary Statistics: " + stats);
        // Output: Summary Statistics: IntSummaryStatistics{count=5, sum=22,
min=3, average=4.400000, max=7}

        // Example 9: Reducing to a single concatenated String
        String reducedString = names.stream().collect(Collectors.reducing((a, b) -> a +
", " + b).orElse("No Names"));
        System.out.println("Reduced String: " + reducedString);
        // Output: Reduced String: Alice, Bob, Charlie, David, Alice
    }
}
```

Output Breakdown:

1. Collecting to a List:

- Output: List: [Alice, Bob, Charlie, David, Alice]
- Explanation: Collects all elements into a List.

2. Collecting to a Set:
- Output: Set: [Alice, Bob, Charlie, David]
- Explanation: Collects elements into a Set, removing duplicates.

3. Collecting to a Map:
- Output: Map: {Alice=5, Bob=3, Charlie=7, David=5}
- Explanation: Maps each name to its length.

4. Joining elements into a single String:
- Output: Joined: Alice, Bob, Charlie, David, Alice
- Explanation: Concatenates all names into a single string, separated by ", ".

5. Grouping elements by length:
- Output: Grouped by length: {3=[Bob], 5=[Alice, David, Alice], 7=[Charlie]}
- Explanation: Groups names by their length.

6. Partitioning elements by length:
- Output: Partitioned by length < 5: {false=[Alice, Charlie, David, Alice], true=[Bob]}
- Explanation: Partitions names into those with length less than 5 and those with length 5 or more.

7. Counting elements:
- Output: Count: 5
- Explanation: Counts the total number of elements in the stream.

8. Summarizing length of names:
- Output: Summary Statistics: IntSummaryStatistics{count=5, sum=22, min=3, average=4.400000, max=7}
- Explanation: Provides statistics about the lengths of names.

9. Reducing to a single concatenated String:

- Output: Reduced String: Alice, Bob, Charlie, David, Alice
- Explanation: Concatenates all names into a single string using reduction.

This example illustrates how the Collectors class can be used to perform various collection and aggregation tasks on streams, making data processing more intuitive and efficient.

forEach Method :

The forEach method is a terminal operation provided by the java.util.stream.Stream interface and is also available in the java.util.Iterable interface. It is used to iterate over each element in a stream or collection and apply a specified action.

Usage:
The forEach method takes a Consumer functional interface as its parameter. A Consumer represents an operation that accepts a single input argument and returns no result. It is used to execute side effects, such as printing or updating elements.

Importance:
1. Iteration: Provides a straightforward way to iterate over elements in a stream or collection.
2. Side Effects: Ideal for performing actions with side effects, such as logging or modifying external states.
3. Integration with Streams: Fits seamlessly with the Stream API, allowing for more functional and declarative data processing.

Advantages:
1. Declarative Style: Allows for a more declarative approach to iteration, enhancing code readability and reducing boilerplate.

2. Readability: Simplifies code by removing the need for explicit loop constructs.
3. Lambda Expressions: Supports lambda expressions and method references, making the code more concise and expressive.
4. Stream Processing: Works well with streams, enabling complex data processing pipelines in a functional style.
5. Parallel Processing: When used with parallel streams, forEach can leverage parallelism to improve performance for large datasets.

Example program :

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class ForEachExample {

    public static void main(String[] args) {

        // Example 1: Using forEach with a Collection
        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry");

        // Using lambda expression to print each fruit
        System.out.println("Using lambda expression:");
        fruits.forEach(fruit -> System.out.println(fruit));
        // Output:
        // Apple
        // Banana
        // Cherry

        // Using method reference to print each fruit
        System.out.println("Using method reference:");
        fruits.forEach(System.out::println);
```

```
// Output:
// Apple
// Banana
// Cherry

// Example 2: Using forEach with a Stream
Stream<String> fruitStream = Stream.of("Apple", "Banana", "Cherry");

// Using lambda expression to print each fruit
System.out.println("Using lambda expression with stream:");
fruitStream.forEach(fruit -> System.out.println(fruit));
// Output:
// Apple
// Banana
// Cherry

// Example 3: Using forEach with a Parallel Stream
Stream<String> parallelFruitStream = Stream.of("Apple", "Banana",
"Cherry").parallel();

// Using lambda expression to print each fruit
System.out.println("Using lambda expression with parallel stream:");
parallelFruitStream.forEach(fruit -> System.out.println(fruit));
// Output:
// The order of elements may vary due to parallel execution
// Possible output:
// Banana
// Cherry
// Apple

// Example 4: Using forEach to Perform Side Effects
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
        // Performing side effect: summing the numbers
        int[] sum = {0}; // Use an array to allow modification within lambda
        numbers.forEach(number -> sum[0] += number);
        System.out.println("Sum of numbers: " + sum[0]);
        // Output: Sum of numbers: 15

        // Example 5: Using forEach with Custom Action
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Custom action: adding "Hello, " prefix to each name
        System.out.println("Names with greeting:");
        names.forEach(name -> System.out.println("Hello, " + name));
        // Output:
        // Hello, Alice
        // Hello, Bob
        // Hello, Charlie
    }
}
```

Explanation of Code and Outputs:

1. Using forEach with a Collection:

Lambda Expression: Prints each fruit in the list using a lambda expression.
Method Reference: Prints each fruit using a method reference.
Output:

```
Using lambda expression:
Apple
Banana
Cherry

Using method reference:
Apple
```

Banana
Cherry

2. Using forEach with a Stream:

Lambda Expression: Prints each fruit in the stream.

Output:

Using lambda expression with stream:
Apple
Banana
Cherry

3. Using forEach with a Parallel Stream:

Lambda Expression: Prints each fruit in the parallel stream. The order of elements may vary due to concurrent execution.

Output (order may vary):

Using lambda expression with parallel stream:
Banana
Cherry
Apple

4. Using forEach to Perform Side Effects:

Side Effect: Calculates the sum of numbers using forEach and modifies an external variable.

Output:

Sum of numbers: 15

5. Using forEach with Custom Action:

Custom Action: Adds a greeting prefix to each name and prints it.

Output:

Names with greeting:
Hello, Alice
Hello, Bob
Hello, Charlie

This code covers various use cases of the forEach method, including its application with collections, streams, parallel streams, and performing side effects.

Nashorn JavaScript Engine

Nashorn is a JavaScript engine introduced in Java 8, which is part of the Java Platform, Standard Edition. It is used to execute JavaScript code within Java applications. Nashorn replaces the older Rhino JavaScript engine and provides a more efficient and faster execution of JavaScript code.

Usage:

Nashorn can be used to execute JavaScript code from within a Java application. It provides a way to integrate JavaScript scripting capabilities with Java code, allowing Java programs to evaluate and execute JavaScript scripts dynamically. Nashorn supports the execution of scripts, calling Java methods from JavaScript, and accessing Java objects in JavaScript.

Usage Example:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class NashornExample {
   public static void main(String[] args) {
      // Create a ScriptEngineManager
```

```
        ScriptEngineManager manager = new ScriptEngineManager();

        // Get Nashorn script engine
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // JavaScript code to be executed
        String script = "var greeting = 'Hello, World!'; greeting;";

        try {
            // Execute JavaScript code
            Object result = engine.eval(script);
            System.out.println(result); // Output: Hello, World!
        } catch (ScriptException e) {
            e.printStackTrace();
        }
    }
}
```

Importance:

1. Integration: Enables integration of JavaScript scripting within Java applications, allowing dynamic scripting and greater flexibility.

2. Performance: Provides improved performance over the older Rhino engine, with faster execution and better optimization.

3. Java Interoperability: Facilitates calling Java methods and accessing Java objects from JavaScript, making it easier to interact between Java and JavaScript code.

Advantages:

1. Speed: Nashorn offers better performance compared to its predecessor Rhino, with faster script execution and improved efficiency.

2. Compatibility: Provides compatibility with Java 8 and allows smooth execution of JavaScript code in Java applications.

3. Enhanced Features: Supports Java 8 features such as lambda expressions and method references, allowing for more modern and efficient scripting.

4. Ease of Use: Simplifies embedding and executing JavaScript within Java applications, enhancing the capability for dynamic scripting and integration.
5. Direct API Access: Allows JavaScript to directly interact with Java APIs, making it easier to use Java libraries and frameworks from JavaScript.