Java 8 features:

1. **Lambda Expressions**
2. **Functional Interfaces**
3. **Streams API**
4. **Default and Static Methods in Interfaces**
5. **Optional Class**
6. **New Date and Time API (java.time package)**
7. **Collectors and Aggregate Functions in Streams**
8. **Method References**
9. **Enhancements in Java Collections (e.g., forEach, removeIf, etc.)**
10. **Concurrency Enhancements (CompletableFuture, parallel streams)**
11. **Nashorn JavaScript Engine**
12. **Base64 Encoding and Decoding API**

Lambda Expressions in Java 8 (Detailed Explanation)

Introduction

Lambda expressions are one of the most significant features introduced in Java 8. They provide a way to write more concise and readable code by enabling functional programming capabilities in Java.

What is a Lambda Expression?

A **lambda expression** is a short block of code that takes parameters and returns a value, without requiring a method declaration within a class. It allows us to pass behavior as an argument to methods.

Syntax of a Lambda Expression

A lambda expression consists of three parts:

```
(parameters) -> { body }
```

- **Parameters**: The input to the lambda expression.
- **Arrow (->)**: Separates the parameters from the body.
- **Body**: The actual logic to be executed.

Examples of Lambda Expressions

1. **Basic Lambda Expression**

```
// Without Lambda Expression
interface MyInterface {
    void show();
}

class Main {
    public static void main(String[] args) {
        MyInterface obj = new MyInterface() {
            public void show() {
                System.out.println("Hello from Anonymous Class!");
            }
        };
        obj.show();
    }
}
```

Using Lambda Expression:

```
public class LambdaExample {
    public static void main(String[] args) {
        MyInterface obj = () -> System.out.println("Hello from
Lambda!");
        obj.show();
    }
}

interface MyInterface {
    void show();
}
```

Here, we have replaced the anonymous class with a lambda expression.

2. **Lambda with Parameters**

```
// Traditional approach using an anonymous class
interface Sum {
    int add(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        Sum obj = (a, b) -> a + b;
        System.out.println("Sum: " + obj.add(10, 20)); // Output:
Sum: 30
    }
}
```

---

3. **Lambda Expression with Multiple Statements**

```
interface Message {
    void showMessage(String msg);
}

public class LambdaExample {
    public static void main(String[] args) {
        Message message = (msg) -> {
            System.out.println("Message received: " + msg);
            System.out.println("Processed Message: " +
msg.toUpperCase());
        };
        message.showMessage("hello java 8");
    }
}
```

Functional Interfaces and Lambda Expressions

Lambda expressions are often used with **Functional Interfaces**. A **functional interface** is an interface that contains only one abstract method. Java 8 provides several built-in functional interfaces in the `java.util.function` package, such as:

- `Consumer<T>` – Accepts a value but does not return anything.
- `Supplier<T>` – Returns a value but does not accept anything.
- `Function<T, R>` – Accepts a value and returns another value.
- `Predicate<T>` – Returns a boolean value based on a condition.

Example using **Predicate**:

```
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = (n) -> n % 2 == 0;

        System.out.println(isEven.test(4)); // true
        System.out.println(isEven.test(7)); // false
    }
}
```

Advantages of Lambda Expressions

1. **Concise Code** – Reduces boilerplate code.
2. **Improved Readability** – More readable compared to anonymous classes.
3. **Encourages Functional Programming** – Makes Java closer to functional programming.
4. **Better Performance** – Reduces overhead from anonymous classes.

Functional Interfaces in Java 8

Introduction

A **functional interface** is an interface in Java that contains exactly **one abstract method**. It may, however, contain **multiple default or static methods**. Functional interfaces are essential in Java 8 because they enable the use of **lambda expressions**, making the code more concise and readable.

Java 8 provides several built-in functional interfaces in the `java.util.function` package, and we can also define our own.

Functional Interface Syntax

A functional interface is simply an interface with a single abstract method. It is typically annotated with `@FunctionalInterface`, though this annotation is optional. However, using `@FunctionalInterface` ensures that the interface follows the functional interface contract (exactly one abstract method).

```
@FunctionalInterface

interface MyFunctionalInterface {

    void sayHello(); // Single abstract method

}
```

Examples of Functional Interfaces

1. Custom Functional Interface with Lambda Expression

```
@FunctionalInterface

interface MyFunctionalInterface {

    void greet(String name);

}

public class FunctionalInterfaceExample {

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

```
        // Implementing functional interface using a lambda
expression

        MyFunctionalInterface obj = (name) ->
System.out.println("Hello, " + name + "!");

        obj.greet("Java 8"); // Output: Hello, Java 8!

    }

}
```

Since `MyFunctionalInterface` has only **one abstract method**, it can be implemented using **lambda expressions**.

---

2. Built-in Functional Interfaces in Java 8

Java 8 provides several predefined functional interfaces in the `java.util.function` package:

Functional Interface	Description
`Predicate<T>`	Returns `true` or `false` based on a condition.
`Function<T, R>`	Accepts one argument and produces a result.
`Consumer<T>`	Accepts an argument and performs some action without returning a result.
`Supplier<T>`	Returns a result without accepting any input.

2.1 `Predicate<T>` – Used for Boolean Conditions

The `Predicate<T>` interface represents a function that takes an argument and returns a boolean result.

```
import java.util.function.Predicate;

public class PredicateExample {

    public static void main(String[] args) {

        Predicate<Integer> isEven = (n) -> n % 2 == 0;

        System.out.println(isEven.test(10)); // true

        System.out.println(isEven.test(15)); // false

    }

}
```

Here, the `test()` method evaluates whether the number is even.

2.2 `Function<T, R>` – Transforming Data

The `Function<T, R>` interface represents a function that takes one argument and returns a value.

```
import java.util.function.Function;

public class FunctionExample {

    public static void main(String[] args) {

        Function<String, Integer> lengthFunction = str ->
str.length();

        System.out.println(lengthFunction.apply("Hello")); //
Output: 5

    }

}
```

Here, the `apply()` method returns the length of a given string.

2.3 `Consumer<T>` – Performs an Action Without Returning a Result

The `Consumer<T>` interface is used when you want to perform an operation on an input without returning a value.

```
import java.util.function.Consumer;

public class ConsumerExample {

    public static void main(String[] args) {

        Consumer<String> printUpperCase = str ->
System.out.println(str.toUpperCase());

        printUpperCase.accept("hello java 8"); // Output: HELLO JAVA
8

    }

}
```

The `accept()` method consumes an input and performs an action.

2.4 `Supplier<T>` – Supplies a Value Without Any Input

The `Supplier<T>` interface is useful when you need to generate or supply values without taking any arguments.

```
import java.util.function.Supplier;

public class SupplierExample {

    public static void main(String[] args) {

        Supplier<Double> randomValue = () -> Math.random();

        System.out.println(randomValue.get()); // Output: A random
number

    }
```

```
}
```

The `get()` method supplies a new random number.

---

3. Functional Interface with Default & Static Methods

In Java 8, functional interfaces can have **default** and **static methods**.

```
@FunctionalInterface

interface MyInterface {

    void show(); // Single abstract method

    default void display() {

        System.out.println("This is a default method in a functional interface.");

    }

    static void staticMethod() {

        System.out.println("This is a static method in a functional interface.");

    }

}


public class FunctionalInterfaceExample {

    public static void main(String[] args) {

        MyInterface obj = () -> System.out.println("Hello from Lambda!");

        obj.show();

        obj.display(); // Calling default method

        MyInterface.staticMethod(); // Calling static method
```

```java
    }
}
```

Key Points:

- `display()` is a **default method**, which means it has an implementation inside the interface.
- `staticMethod()` is a **static method**, so it can be called using the interface name.

Advantages of Functional Interfaces

1. **Improves Code Readability** – Reduces the amount of code written.
2. **Enhances Reusability** – Can be used in multiple places without modification.
3. **Enables Functional Programming** – Supports lambda expressions.
4. **Simplifies Development** – No need for unnecessary boilerplate code.

Streams API in Java 8

Introduction

The **Streams API** in Java 8 provides a powerful way to process collections of data efficiently. It enables functional-style operations on collections (like filtering, mapping, and reducing) without modifying the original data. Streams make it easier to work with data in a **declarative, concise, and parallelizable** manner.

---

1. What is a Stream?

A **Stream** is a sequence of elements supporting sequential and parallel aggregate operations. It does not store data but processes data from a source such as **collections, arrays, or I/O channels**.

Characteristics of Streams:

1. **Not a Data Structure** – It does not store elements; it processes them.
2. **Functional-Style Operations** – Uses declarative operations like `map`, `filter`, and `reduce`.
3. **Lazy Evaluation** – Intermediate operations are not executed until a terminal operation is invoked.
4. **Parallel Processing** – Can be processed in parallel to improve performance.

---

2. Creating Streams in Java

There are multiple ways to create a stream:

From a Collection (List, Set)

```
import java.util.*;

import java.util.stream.*;

public class StreamExample {

    public static void main(String[] args) {

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

        Stream<String> stream = names.stream(); // Creating a Stream
```

```java
        stream.forEach(System.out::println);  // Output: John, Jane, Jack, Jill
    }
}
```

From an Array

```java
import java.util.Arrays;
import java.util.stream.Stream;
public class StreamExample {
    public static void main(String[] args) {
        String[] arr = {"Java", "Python", "C++"};
        Stream<String> stream = Arrays.stream(arr);
        stream.forEach(System.out::println);
    }
}
```

Using `Stream.of()`

```java
Stream<String> stream = Stream.of("Apple", "Banana", "Cherry");
stream.forEach(System.out::println);
```

Generating an Infinite Stream

```java
Stream<Integer> infiniteStream = Stream.iterate(1, n -> n + 1);
infiniteStream.limit(5).forEach(System.out::println); // Output: 1 2 3 4 5
```

---

3. Stream Operations

There are **two types of operations** in streams:

1. **Intermediate Operations** – Transform a stream but do not execute until a terminal operation is called (e.g., `filter`, `map`, `sorted`).
2. **Terminal Operations** – Produce a result or a side-effect (e.g., `collect`, `forEach`, `reduce`).

3.1 Intermediate Operations

a) `filter()` – Filters elements based on a condition

```
import java.util.*;

import java.util.stream.*;

public class StreamFilterExample {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(10, 20, 25, 30, 35);

        List<Integer> evenNumbers = numbers.stream()

                                          .filter(n -> n % 2 == 0)

               .collect(Collectors.toList());

        System.out.println(evenNumbers); // Output: [10, 20, 30]

    }

}
```

b) `map()` – Transforms each element

```
List<String> names = Arrays.asList("john", "jane", "jack");

List<String> upperCaseNames = names.stream()

                                   .map(String::toUpperCase)

                                   .collect(Collectors.toList());

System.out.println(upperCaseNames); // Output: [JOHN, JANE, JACK]
```

c) `sorted()` – Sorts the elements

```
List<Integer> numbers = Arrays.asList(5, 2, 8, 3);
List<Integer> sortedNumbers = numbers.stream()
                                     .sorted()
                                     .collect(Collectors.toList());
System.out.println(sortedNumbers); // Output: [2, 3, 5, 8]
```

3.2 Terminal Operations

a) `forEach()` – Iterates over elements

```
Stream.of("A", "B", "C").forEach(System.out::println);
```

b) `collect()` – Converts a stream into a collection

```
List<String> list = Stream.of("Java", "Python", "C++")
                         .collect(Collectors.toList());

System.out.println(list); // Output: [Java, Python, C++]
```

c) `reduce()` – Reduces elements to a single value

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
int sum = numbers.stream().reduce(0, Integer::sum);
System.out.println(sum); // Output: 10
```

d) `count()` – Counts the number of elements

```
long count = Stream.of("Apple", "Banana", "Cherry").count();

System.out.println(count); // Output: 3
```

---

4. Parallel Streams

Java 8 allows parallel processing using the **parallelStream()** method.

```
import java.util.*;

public class ParallelStreamExample {

    public static void main(String[] args) {

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7,
8, 9, 10);

        numbers.parallelStream()

               .filter(n -> n % 2 == 0)

               .forEach(System.out::println);

    }

}
```

This code filters even numbers in parallel, improving performance on large datasets.

---

5. Key Benefits of Streams API

1. **Improved Readability** – Functional-style operations make the code easier to read.
2. **Less Boilerplate Code** – No need for explicit loops.
3. **Lazy Evaluation** – Operations are executed only when needed, improving efficiency.
4. **Better Performance** – Supports parallel execution for faster processing.
5. **Encourages Functional Programming** – Promotes an expressive and declarative approach.

Default and Static Methods in Interfaces (Java 8)

Introduction

Before Java 8, interfaces in Java could only contain **abstract methods** (methods without a body), and any implementation of those methods had to be provided by the classes that implemented the interface. Java 8 introduced two new kinds of methods in interfaces:

1. **Default Methods**
2. **Static Methods**

These changes enhance the flexibility of interfaces and allow for **evolution of interfaces** without breaking backward compatibility.

---

1. Default Methods in Interfaces

What is a Default Method?

A **default method** is a method in an interface that has a **default implementation**. This allows you to add new methods to an interface without breaking the existing implementations of that interface in other classes. The **default** keyword is used to define such methods.

Syntax of Default Method:

```
interface MyInterface {
    // Abstract method (must be implemented by implementing class)
    void abstractMethod();

    // Default method (can have a body)
    default void defaultMethod() {
        System.out.println("This is a default method");
    }
}
```

Example of Default Method:

```
interface MyInterface {
    // Abstract method
    void sayHello();

    // Default method with an implementation
    default void sayGoodbye() {
```

```
        System.out.println("Goodbye from Default Method");
    }
}

class MyClass implements MyInterface {
    @Override
    public void sayHello() {
        System.out.println("Hello from MyClass");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.sayHello();      // Output: Hello from MyClass
        obj.sayGoodbye();    // Output: Goodbye from Default Method
    }
}
```

Key Points:

- Default methods allow interfaces to have method implementations.
- Classes implementing the interface are **not required** to implement default methods (they are optional).
- You can have **multiple default methods** in an interface.

Why Use Default Methods?

1. **Backward Compatibility**: Allows you to add new methods to an interface without affecting existing classes that implement the interface.
2. **Code Reusability**: Common method functionality can be provided directly in the interface.
3. **Enhances Interfaces**: Interfaces can be more flexible and maintainable.

---

2. Static Methods in Interfaces

What is a Static Method?

A **static method** in an interface is a method that belongs to the interface itself rather than any instance of the class implementing the interface. Static methods can be **called directly on the interface** and are not inherited by implementing classes.

Syntax of Static Method:

```
interface MyInterface {
    // Static method in an interface
    static void staticMethod() {
        System.out.println("This is a static method");
    }
}
```

Example of Static Method:

```
interface MyInterface {
    static void display() {
        System.out.println("Static method in interface");
    }
}

public class TestStaticMethod {
    public static void main(String[] args) {
        // Calling the static method directly on the interface
        MyInterface.display();  // Output: Static method in
interface
    }
}
```

Key Points:

- Static methods are **not inherited** by implementing classes.
- They must be **called on the interface** itself.
- Useful for providing utility or helper methods that are related to the interface but not intended to be overridden.

---

3. Differences Between Default and Static Methods

Feature	Default Methods	Static Methods
Implemented by	Can have a default implementation.	Must have a full implementation.
Inheritance	Inherited by implementing classes (can be overridden).	Not inherited by implementing classes.

Access	Accessed through the class implementing the interface or the instance.	Accessed through the interface itself.
Purpose	Provides default behavior that can be overridden.	Provides utility methods related to the interface.

---

4. Multiple Inheritance of Default Methods (Diamond Problem)

Java 8 allows an interface to **extend multiple interfaces** with default methods. This may cause ambiguity if two parent interfaces have the same default method. In this case, Java will throw a **compilation error** unless you explicitly **override** the method.

Example of Diamond Problem:

```
interface A {
    default void show() {
        System.out.println("A");
    }
}

interface B {
    default void show() {
        System.out.println("B");
    }
}

class C implements A, B {
    @Override
    public void show() {
        // Resolving the ambiguity by overriding the method
        System.out.println("C");
    }

    public static void main(String[] args) {
        C obj = new C();
        obj.show();  // Output: C
    }
}
```

Solution: The class `C` resolves the conflict by overriding the `show()` method. If it doesn't, the compiler will show an error due to the ambiguity.

The introduction of **default** and **static** methods in Java 8 interfaces enhances the flexibility and usability of interfaces:

- **Default methods** allow you to add functionality to interfaces without breaking existing implementations.
- **Static methods** provide utility methods related to the interface but don't require implementation by classes.

Optional Class in Java 8

Introduction

Before Java 8, handling **null values** was often done using explicit null checks, leading to **NullPointerException (NPE)** if not handled properly. Java 8 introduced the **`Optional`** class (from `java.util` package) to address this issue.

`Optional<T>` is a **container object** that may or may not contain a non-null value. It helps **avoid null checks**, making the code cleaner and reducing runtime errors.

---

1. Why Use Optional?

Problems Before Java 8

Consider the following example without `Optional`:

```
public class WithoutOptional {

    public static String getCustomerName(Customer customer) {

        return customer.getName(); // May throw NullPointerException if customer is null

    }


    public static void main(String[] args) {

        Customer customer = null;

        System.out.println(getCustomerName(customer)); // Throws NullPointerException

    }

}
```

Issue: If `customer` is null, `getCustomerName(customer)` will throw **NullPointerException**.

Solution with Optional

```java
import java.util.Optional;

public class WithOptional {

    public static String getCustomerName(Optional<Customer> customer) {

        return customer.map(Customer::getName).orElse("Unknown");

    }


    public static void main(String[] args) {

        Optional<Customer> customer = Optional.empty();

        System.out.println(getCustomerName(customer)); // Output: Unknown

    }

}
```

Fix: `Optional` ensures that `null` values are handled gracefully without throwing an exception.

---

2. Creating Optional Objects

Java provides multiple ways to create an `Optional` object:

a) `Optional.of(value)` – Creates an Optional with a non-null value

```java
Optional<String> optional = Optional.of("Hello, Java 8");

System.out.println(optional.get()); // Output: Hello, Java 8
```

Throws NullPointerException if the value is null:

```
Optional<String> optional = Optional.of(null); // Throws
NullPointerException
```

---

b) `Optional.ofNullable(value)` – Allows both null and non-null values

```
Optional<String> optional = Optional.ofNullable(null); // No
Exception

System.out.println(optional.isPresent()); // Output: false
```

Use `ofNullable()` if the value might be null.

---

c) `Optional.empty()` – Creates an empty Optional

```
Optional<String> optional = Optional.empty();

System.out.println(optional.isPresent()); // Output: false
```

---

3. Checking Optional Values

a) `isPresent()` – Checks if a value is present

```
Optional<String> optional = Optional.of("Java");

System.out.println(optional.isPresent()); // Output: true

Optional<String> emptyOptional = Optional.empty();

System.out.println(emptyOptional.isPresent()); // Output: false
```

---

b) `ifPresent()` – Executes code if a value is present

```
Optional<String> optional = Optional.of("Java 8");

optional.ifPresent(value -> System.out.println("Value: " + value));

// Output: Value: Java 8
```

Use case: Avoids explicit null checks and makes the code more readable.

---

4. Retrieving Values from Optional

a) `get()` – Returns the value if present (Throws exception if empty)

```
Optional<String> optional = Optional.of("Hello");

System.out.println(optional.get()); // Output: Hello
```

Avoid using `get()` directly, as it throws `NoSuchElementException` if Optional is empty.

```
Optional<String> emptyOptional = Optional.empty();

System.out.println(emptyOptional.get()); // Throws
NoSuchElementException
```

---

b) `orElse(defaultValue)` – Returns the value or a default

```
Optional<String> optional = Optional.empty();

System.out.println(optional.orElse("Default Value")); // Output:
Default Value
```

---

c) `orElseGet(Supplier)` – Returns the value or calls a supplier function

```
Optional<String> optional = Optional.empty();

System.out.println(optional.orElseGet(() -> "Generated Value")); //
Output: Generated Value
```

Use `orElseGet()` instead of `orElse()` when computing the default value is expensive.

---

d) `orElseThrow(Supplier)` – Throws an exception if no value is present

```
Optional<String> optional = Optional.empty();

System.out.println(optional.orElseThrow(() -> new
RuntimeException("Value is missing")));
```

Throws `RuntimeException: Value is missing` if Optional is empty.

---

5. Transforming Optional Values

a) `map(Function)` – Transforms the value if present

```
Optional<String> optional = Optional.of("hello");

Optional<String> upperCase = optional.map(String::toUpperCase);

System.out.println(upperCase.orElse("Default")); // Output: HELLO
```

If Optional is empty, `map()` simply returns an empty Optional.

---

b) `flatMap(Function)` – Similar to `map()`, but used when the function returns an Optional

```
class Person {

    private Optional<String> email;


    public Person(String email) {

        this.email = Optional.ofNullable(email);

    }


    public Optional<String> getEmail() {

        return email;

    }

}


public class FlatMapExample {

    public static void main(String[] args) {

        Person person = new Person("person@example.com");

        Optional<Person> optionalPerson = Optional.of(person);


        // Using flatMap to get the email

        Optional<String> email =
optionalPerson.flatMap(Person::getEmail);

        System.out.println(email.orElse("No email provided")); //
Output: person@example.com

    }
```

```
}
```

Use `flatMap()` when the function already returns an `Optional`, to avoid nested Optionals (`Optional<Optional<T>>`).

---

6. Filtering Optional Values

Using `filter(Predicate)`

```
Optional<Integer> age = Optional.of(25);

Optional<Integer> validAge = age.filter(a -> a > 18);

System.out.println(validAge.isPresent()); // Output: true

Optional<Integer> age = Optional.of(15);

Optional<Integer> validAge = age.filter(a -> a > 18);

System.out.println(validAge.isPresent()); // Output: false
```

Use case: Helps apply conditional checks inside `Optional` without needing `if` statements.

---

7. Optional in Real-World Scenario

Consider a **Customer** class where a customer may or may not have an email.

Without Optional

```
public class Customer {

    private String email;

    public String getEmail() {

        return email;

    }
```

```java
}


Customer customer = new Customer();

if (customer != null && customer.getEmail() != null) {

    System.out.println(customer.getEmail());

} else {

    System.out.println("No Email Provided");

}
```

Issue: Requires multiple null checks.

With Optional

```java
import java.util.Optional;

public class Customer {

    private Optional<String> email;

    public Optional<String> getEmail() {

        return email;

    }

}


Customer customer = new Customer();

System.out.println(customer.getEmail().orElse("No Email Provided"));
```

No explicit null checks, making the code more readable.

---

8. Summary

Method	Description
`of(value)`	Creates an Optional with a non-null value.
`ofNullable(value)`	Allows null and non-null values.
`empty()`	Returns an empty Optional.
`isPresent()`	Checks if a value is present.
`ifPresent(Consumer)`	Executes an action if a value is present.
`orElse(defaultValue)`	Returns value if present, otherwise returns default value.
`orElseGet(Supplier)`	Returns value if present, otherwise executes supplier function.
`orElseThrow(Supplier)`	Throws an exception if no value is present.
`map(Function)`	Transforms the value if present.
`flatMap(Function)`	Similar to `map()`, but avoids nested Optionals.
`filter(Predicate)`	Returns an Optional if the condition is met.

Java 8 Date and Time API (`java.time` Package)

1. Introduction

Before Java 8, Java's date and time handling was done using `java.util.Date`, `java.util.Calendar`, and `java.text.SimpleDateFormat`. However, these classes had **several issues**:

- **Mutable objects:** `Date` and `Calendar` were mutable, making them **not thread-safe**.
- **Complex APIs:** Working with `Calendar` and `Date` was difficult due to unclear and error-prone methods.
- **Poor timezone support:** `Date` had **no direct support for time zones**.

To address these issues, **Java 8 introduced the `java.time` package**, which provides a **modern, immutable, and thread-safe** API for date and time manipulation.

---

2. Key Classes in `java.time` Package

The `java.time` package introduced several important classes:

Class	Description
`LocalDate`	Represents a **date** (year, month, day) **without time**.
`LocalTime`	Represents a **time** (hour, minute, second) **without date**.
`LocalDateTime`	Represents both **date and time**, but **without time zone**.
`ZonedDateTime`	Represents both **date and time with time zone**.
`Instant`	Represents an **instant timestamp** (e.g., Unix timestamp).
`Duration`	Represents **time-based duration** (e.g., 5 hours, 30 minutes).
`Period`	Represents **date-based duration** (e.g., 2 years, 3 months).

`ZoneId` & `ZoneOffset`	Represents **time zones and offsets**.
`DateTimeFormatter`	Formats and parses date-time objects.

---

3. `LocalDate` – Working with Dates (No Time Component)

The `LocalDate` class represents **a date without a time zone** (e.g., `2025-02-06`).

Creating `LocalDate` Instances

```
import java.time.LocalDate;

public class LocalDateExample {
    public static void main(String[] args) {
        // Current date
        LocalDate today = LocalDate.now();
        System.out.println("Current Date: " + today);

        // Specific date
        LocalDate specificDate = LocalDate.of(2023, 12, 25);
        System.out.println("Specific Date: " + specificDate);

        // Parse date from String
        LocalDate parsedDate = LocalDate.parse("2022-05-10");
        System.out.println("Parsed Date: " + parsedDate);
    }
}
```

Output Example:

```
Current Date: 2025-02-06
Specific Date: 2023-12-25
Parsed Date: 2022-05-10
```

`LocalDate` Operations

```
import java.time.LocalDate;

public class LocalDateOperations {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2024, 2, 6);

        // Adding and subtracting days, months, and years
        System.out.println("After 10 days: " + date.plusDays(10));
        System.out.println("After 2 months: " + date.plusMonths(2));
        System.out.println("Before 1 year: " + date.minusYears(1));

        // Extracting values
        System.out.println("Year: " + date.getYear());
        System.out.println("Month: " + date.getMonth()); // FEBRUARY
        System.out.println("Day of Week: " + date.getDayOfWeek());
// TUESDAY
    }
}
```

Output Example:

```
After 10 days: 2024-02-16
After 2 months: 2024-04-06
Before 1 year: 2023-02-06
Year: 2024
Month: FEBRUARY
Day of Week: TUESDAY
```

---

4. `LocalTime` – Working with Time (No Date Component)

The `LocalTime` class represents **time without a date**.

Creating and Using `LocalTime`

```
import java.time.LocalTime;
```

```
public class LocalTimeExample {
    public static void main(String[] args) {
        // Current time
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);

        // Specific time
        LocalTime specificTime = LocalTime.of(14, 30, 45);
        System.out.println("Specific Time: " + specificTime);

        // Adding and subtracting time
        System.out.println("After 2 hours: " + now.plusHours(2));
        System.out.println("Before 30 minutes: " +
now.minusMinutes(30));
    }
}
```

---

5. `LocalDateTime` – Working with Date and Time

`LocalDateTime` combines both `LocalDate` and `LocalTime`.

```
import java.time.LocalDateTime;

public class LocalDateTimeExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current DateTime: " + now);

        LocalDateTime specificDateTime = LocalDateTime.of(2024, 2,
6, 14, 45, 30);
        System.out.println("Specific DateTime: " +
specificDateTime);
    }
}
```

---

6. `ZonedDateTime` – Handling Time Zones

To work with time zones, use `ZonedDateTime`.

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class ZonedDateTimeExample {
    public static void main(String[] args) {
        ZonedDateTime now = ZonedDateTime.now();
        System.out.println("Current Zoned DateTime: " + now);

        ZonedDateTime newYorkTime =
ZonedDateTime.now(ZoneId.of("America/New_York"));
        System.out.println("New York Time: " + newYorkTime);
    }
}
```

---

7. `Instant` – Representing Timestamps

`Instant` represents a **specific moment in time** (like Unix timestamps).

```
import java.time.Instant;

public class InstantExample {
    public static void main(String[] args) {
        Instant now = Instant.now();
        System.out.println("Current Timestamp: " + now);
    }
}
```

---

8. `Duration` and `Period` – Measuring Time Differences

Using `Duration` (Time-based differences)

```
import java.time.Duration;
import java.time.LocalTime;

public class DurationExample {
    public static void main(String[] args) {
        LocalTime start = LocalTime.of(10, 0);
        LocalTime end = LocalTime.of(12, 30);
```

```java
        Duration duration = Duration.between(start, end);
        System.out.println("Duration: " + duration.toHours() + "
hours, " + duration.toMinutes() + " minutes");
    }
}
```

Using `Period` (Date-based differences)

```java
import java.time.LocalDate;
import java.time.Period;

public class PeriodExample {
    public static void main(String[] args) {
        LocalDate birthDate = LocalDate.of(1995, 5, 20);
        LocalDate today = LocalDate.now();

        Period age = Period.between(birthDate, today);
        System.out.println("Age: " + age.getYears() + " years, " +
age.getMonths() + " months");
    }
}
```

---

9. Formatting Dates (`DateTimeFormatter`)

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeFormatterExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

        System.out.println("Formatted Date: " +
now.format(formatter));
    }
}
```

- **Immutable & Thread-Safe:** Unlike `Date`, `java.time` classes are immutable.
- **Better API Design:** Clear method names and better support for calculations.
- **Time Zone Handling:** `ZonedDateTime` provides excellent support.
- **Avoids Legacy Date Issues:** Eliminates `NullPointerException` from `Date`.

Collectors and Aggregate Functions in Java 8 Streams

1. Introduction

Java 8 introduced the **Stream API**, which allows performing bulk operations on collections in a functional way. A key feature of Streams is **collecting** results using `Collectors`.

Why Use Collectors?

- Used to **accumulate, transform, and summarize** stream elements.
- Allows **grouping, partitioning, reducing, and mapping** data.
- Works efficiently with parallel streams.

The **`Collectors`** class (from `java.util.stream.Collectors`) provides **ready-made methods** for common operations like:

- Converting a stream into a **List, Set, or Map**.
- Computing **count, sum, min, max, and average**.
- **Grouping and partitioning** data.

---

2. Basic Collectors Methods

a) `collect(Collectors.toList())` – Convert Stream to List

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

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

        List<String> filteredNames = names.stream()
                                          .filter(name ->
name.startsWith("A"))

.collect(Collectors.toList());
```

```
        System.out.println(filteredNames); // Output: [Alice]
    }
}
```

Converts a filtered stream into a `List`.

---

b) `collect(Collectors.toSet())` – Convert Stream to Set

```
import java.util.Arrays;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectToSetExample {
    public static void main(String[] args) {
        Set<Integer> numbers = Arrays.asList(1, 2, 3, 4, 3, 2, 1)
                                     .stream()
                                     .collect(Collectors.toSet());

        System.out.println(numbers); // Output: [1, 2, 3, 4]
(removes duplicates)
    }
}
```

Ensures unique elements.

---

c) `collect(Collectors.toMap())` – Convert Stream to Map

```
import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectToMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = Arrays.asList("Alice", "Bob",
"Charlie")
                                         .stream()

.collect(Collectors.toMap(String::length, name -> name, (existing,
replacement) -> existing));
```

```
        System.out.println(map); // Output: {5=Alice, 3=Bob,
7=Charlie}
    }
}
```

Handles duplicate keys using `(existing, replacement) -> existing`.

---

3. Aggregate Functions Using Collectors

Aggregate functions perform calculations such as **count, sum, min, max, and average**.

a) `count()` – Count the Number of Elements

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

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

        long count = names.stream().collect(Collectors.counting());
        System.out.println("Count: " + count); // Output: Count: 4
    }
}
```

Returns the total number of elements in the stream.

---

b) `maxBy()` and `minBy()` – Find Maximum and Minimum Values

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Collectors;

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

```java
        List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);

        Optional<Integer> max =
numbers.stream().collect(Collectors.maxBy(Comparator.naturalOrder())
);
        Optional<Integer> min =
numbers.stream().collect(Collectors.minBy(Comparator.naturalOrder())
);

        System.out.println("Max: " + max.get()); // Output: Max: 50
        System.out.println("Min: " + min.get()); // Output: Min: 10
    }
}
```

Uses `Comparator` to find the largest and smallest elements.

---

c) `summingInt()` – Compute Sum of Elements

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

public class CollectSumExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 10, 15, 20);

        int sum =
numbers.stream().collect(Collectors.summingInt(Integer::intValue));

        System.out.println("Sum: " + sum); // Output: Sum: 50
    }
}
```

Sums all values in the stream.

---

d) `averagingInt()` – Compute Average

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

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

public class CollectAverageExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 20, 30, 40);

        double avg =
numbers.stream().collect(Collectors.averagingInt(Integer::intValue))
;

        System.out.println("Average: " + avg); // Output: Average:
25.0
    }
}
```

Computes the average of numbers.

---

4. Grouping and Partitioning

a) `groupingBy()` – Group Elements by a Property

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

class Employee {
    String name;
    String department;

    Employee(String name, String department) {
        this.name = name;
        this.department = department;
    }

    public String getDepartment() {
        return department;
    }
}
```

```
public class CollectGroupingExample {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("Alice", "HR"),
                new Employee("Bob", "IT"),
                new Employee("Charlie", "HR"),
                new Employee("David", "IT")
        );

        Map<String, List<Employee>> groupedByDept =
employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

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

Groups employees by department.

---

b) `partitioningBy()` – Divide Elements into Two Groups

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectPartitioningExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 15, 20, 25, 30);

        Map<Boolean, List<Integer>> partitioned = numbers.stream()
                .collect(Collectors.partitioningBy(n -> n % 2 ==
0));

        System.out.println("Even numbers: " +
partitioned.get(true));
        System.out.println("Odd numbers: " +
partitioned.get(false));
    }
```

```
}
```

Separates numbers into even and odd groups.

---

5. Joining Strings Using `joining()`

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

public class CollectJoiningExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob",
"Charlie");

        String result = names.stream().collect(Collectors.joining(",
", "[", "]"));

        System.out.println(result); // Output: [Alice, Bob, Charlie]
    }
}
```

Concatenates elements into a formatted string.

Collector Method	Description
`toList()`	Collects elements into a List.
`toSet()`	Collects elements into a Set (removes duplicates).
`toMap()`	Collects elements into a Map.
`counting()`	Counts the elements.

`summingInt()`, `averagingInt()`	Computes sum or average.
`maxBy()`, `minBy()`	Finds max or min value.
`groupingBy()`	Groups elements based on a property.
`partitioningBy()`	Splits elements into two groups.
`joining()`	Joins elements into a string.

Method References in Java 8

1. Introduction

Method references in Java 8 provide a way to **refer to existing methods** by name instead of invoking them directly. They make **lambda expressions more readable and concise**.

Why Use Method References?

- **Improves Readability** – More concise than lambda expressions.
- **Avoids Redundant Code** – Directly calls existing methods.
- **Reusability** – Can reuse existing static or instance methods.

---

2. Syntax of Method References

The general syntax of a method reference is:

```
ClassName::methodName
```

Here, `::` is the **method reference operator**.

---

3. Types of Method References

Java 8 provides **four types** of method references:

Type	Syntax	Example
Reference to a **static method**	`ClassName::staticMethodName`	`Math::sqrt`
Reference to an **instance method of a particular object**	`instance::instanceMethodName`	`obj::toString`
Reference to an **instance method of an arbitrary object of a particular type**	`ClassName::instanceMethodName`	`String::length`
Reference to a **constructor**	`ClassName::new`	`ArrayList::new`

---

4. Method Reference Examples

a) Reference to a Static Method

We can replace a **lambda expression** with a method reference if the method being called is **static**.

Example 1: Using `Math::sqrt`

```
import java.util.function.Function;

public class StaticMethodReference {
    public static void main(String[] args) {
        // Using Lambda Expression
        Function<Double, Double> lambdaSqrt = x -> Math.sqrt(x);
        System.out.println("Lambda Result: " +
lambdaSqrt.apply(25.0));

        // Using Method Reference
        Function<Double, Double> methodRefSqrt = Math::sqrt;
        System.out.println("Method Reference Result: " +
methodRefSqrt.apply(25.0));
    }
}
```

Output:

```
Lambda Result: 5.0
Method Reference Result: 5.0
```

---

b) Reference to an Instance Method of a Particular Object

If we have an **existing object**, we can refer to its methods.

Example 2: Using `toUpperCase()`

```
import java.util.function.Supplier;
```

```
public class InstanceMethodReference {
    public static void main(String[] args) {
        String message = "hello";

        // Using Lambda Expression
        Supplier<String> lambdaUpper = () -> message.toUpperCase();
        System.out.println("Lambda: " + lambdaUpper.get());

        // Using Method Reference
        Supplier<String> methodRefUpper = message::toUpperCase;
        System.out.println("Method Reference: " +
methodRefUpper.get());
    }
}
```

✅ **Output:**

```
Lambda: HELLO
Method Reference: HELLO
```

---

c) Reference to an Instance Method of an Arbitrary Object of a Specific Type

If we are working with a stream of objects, we can use method references for **instance methods**.

Example 3: Using `String::length`

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

public class ArbitraryObjectMethodReference {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob",
"Charlie");

        // Using Lambda Expression
        names.stream().map(name ->
name.length()).forEach(System.out::println);
```

```
        // Using Method Reference

names.stream().map(String::length).forEach(System.out::println);
    }
}
```

Output:

```
5
3
7
```

`String::length` replaces `name -> name.length()`.

---

d) Reference to a Constructor

A method reference can also point to a **constructor**, which is useful for creating new objects.

Example 4: Using `ArrayList::new`

```
import java.util.ArrayList;
import java.util.function.Supplier;

public class ConstructorReference {
    public static void main(String[] args) {
        // Using Lambda Expression
        Supplier<ArrayList<String>> lambdaList = () -> new
ArrayList<>();
        System.out.println("Lambda: " + lambdaList.get());

        // Using Constructor Reference
        Supplier<ArrayList<String>> methodRefList = ArrayList::new;
        System.out.println("Method Reference: " +
methodRefList.get());
    }
}
```

Output:

```
Lambda: []
Method Reference: []
```

`ArrayList::new` replaces `() -> new ArrayList<>()`.

---

5. Method References vs Lambda Expressions

Feature	Lambda Expression	Method Reference
Readability	Explicit but sometimes verbose	More concise
Usage	Can perform multiple operations	Directly calls a method
When to Use?	When more complex logic is needed	When a method already exists

---

6. Practical Use Case: Sorting a List

Example 5: Sorting a List Using Method References

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

public class SortingWithMethodReference {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Charlie", "Alice",
"Bob");

        // Using Lambda Expression
        names.sort((s1, s2) -> s1.compareTo(s2));
        System.out.println("Lambda Sorted: " + names);

        // Using Method Reference
        names.sort(String::compareTo);
        System.out.println("Method Reference Sorted: " + names);
    }
}
```

Output:

```
Lambda Sorted: [Alice, Bob, Charlie]
Method Reference Sorted: [Alice, Bob, Charlie]
```

`String::compareTo` replaces `(s1, s2) -> s1.compareTo(s2)`.

---

7. When to Use Method References?

Use Case	Example
Static method reference	`Math::max`
Instance method of a specific object	`str::toLowerCase`
Instance method of an arbitrary object	`List<String>::size`
Constructor reference	`ArrayList::new`

---

8. Summary

- **Method references** make lambda expressions **more readable**.
- They refer to **static methods, instance methods, or constructors**.
- They **must match** the expected functional interface **method signature**.
- They **replace simple lambdas** that only call an existing method.

---

9. Quick Recap with Examples

Type	Example	Equivalent Lambda
Static Method Reference	`Math::sqrt`	`(x) -> Math.sqrt(x)`
Instance Method of Object	`str::toUpperCase`	`() -> str.toUpperCase()`
Instance Method of Arbitrary Object	`String::length`	`(s) -> s.length()`

Constructor Reference	`ArrayList::new`	`() -> new ArrayList<>()`

Enhancements in Java Collections in Java 8

Java 8 introduced several **enhancements** to the **Collections Framework**, making it easier to work with collections using **functional programming** features such as **lambda expressions** and **streams**. Some of the key improvements include:

1. **`forEach()`** – Iterating over collections efficiently
2. **`removeIf()`** – Removing elements based on a condition
3. **`replaceAll()`** – Updating all elements in a list
4. **`computeIfAbsent()` and `computeIfPresent()`** – Simplified map operations
5. **`merge()`** – Combining values in a `Map`
6. **`getOrDefault()`** – Handling default values in a `Map`

---

1. `forEach()` Method – Improved Iteration

The **`forEach()`** method was introduced in `java.lang.Iterable` and `java.util.Map`, allowing easy iteration over collections using **lambda expressions**.

Example 1: Using `forEach()` on a `List`

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

public class ForEachExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob",
"Charlie");

        // Using forEach() with a lambda
        names.forEach(name -> System.out.println(name));

        // Using method reference
        names.forEach(System.out::println);
    }
}
```

Output:

```
Alice
```

```
Bob
Charlie
```

Replaces traditional for-loops and iterators.

Example 2: Using `forEach()` on a `Map`

```
import java.util.HashMap;
import java.util.Map;

public class ForEachMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Java");
        map.put(2, "Python");
        map.put(3, "C++");

        // Iterating using forEach()
        map.forEach((key, value) -> System.out.println("Key: " + key
+ ", Value: " + value));
    }
}
```

Output:

```
Key: 1, Value: Java
Key: 2, Value: Python
Key: 3, Value: C++
```

Efficient way to iterate over `Map` entries.

---

2. `removeIf()` – Conditional Removal from Collections

The **`removeIf()`** method allows removing elements from a `Collection` based on a **given condition (predicate)**.

Example 3: Removing Elements from a `List`

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

public class RemoveIfExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(10,
15, 20, 25, 30));

        // Remove all numbers greater than 20
        numbers.removeIf(n -> n > 20);

        System.out.println(numbers); // Output: [10, 15, 20]
    }
}
```

Removes elements matching the condition (`n > 20`).

Example 4: Removing Entries from a `Map`

```java
import java.util.HashMap;
import java.util.Map;

public class RemoveIfMapExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Remove entries where value is greater than 28
        map.entrySet().removeIf(entry -> entry.getValue() > 28);

        System.out.println(map); // Output: {Alice=25}
    }
}
```

Removes key-value pairs where the value is greater than 28.

---

3. `replaceAll()` – Updating Elements in a List

The **`replaceAll()`** method allows updating **each element** in a `List` based on a function.

Example 5: Doubling All Elements in a List

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

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

        // Multiply each element by 2
        numbers.replaceAll(n -> n * 2);

        System.out.println(numbers); // Output: [2, 4, 6, 8, 10]
    }
}
```

Efficiently updates all elements in a collection.

---

4. `computeIfAbsent()` and `computeIfPresent()` – Efficient Map Operations

These methods simplify **conditional modifications** in a `Map`.

Example 6: Using `computeIfAbsent()`

- Adds a value **only if the key is missing**.

```
import java.util.HashMap;
import java.util.Map;

public class ComputeIfAbsentExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
```

```
        // If "Bob" is missing, compute and add it
        map.computeIfAbsent("Bob", key -> 30);

        System.out.println(map); // Output: {Alice=25, Bob=30}
    }
}
```

Adds a value only if the key is absent.

Example 7: Using `computeIfPresent()`

- Updates a value **only if the key exists**.

```
import java.util.HashMap;
import java.util.Map;

public class ComputeIfPresentExample {

 Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);

        // If "Alice" exists, update her age
        map.computeIfPresent("Alice", (key, value) -> value + 5);

        System.out.println(map); // Output: {Alice=30}
    }
}
```

Updates only if the key exists.

---

5. `merge()` – Combining Values in a Map

The `merge()` method helps in merging values efficiently.

Example 8: Merging Values in a Map

```
import java.util.HashMap;
import java.util.Map;
```

```
public class MergeExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 10);

        // Merge "Alice" by adding 5
        map.merge("Alice", 5, Integer::sum);

        // Merge a new key "Bob"
        map.merge("Bob", 10, Integer::sum);

        System.out.println(map); // Output: {Alice=15, Bob=10}
    }
}
```

Adds values for existing keys and inserts new ones.

---

6. `getOrDefault()` – Handling Missing Keys in a Map

The **`getOrDefault()`** method prevents `null` checks.

Example 9: Avoiding `null` Values

```
import java.util.HashMap;
import java.util.Map;

public class GetOrDefaultExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);

        // Get existing value
        System.out.println(map.getOrDefault("Alice", 0)); // Output:
25

        // Get default for missing key
        System.out.println(map.getOrDefault("Bob", 0)); // Output: 0
    }
}
```

Returns a default value if the key is missing.

Summary Table

Method	Usage
`forEach()`	Iterate over collections easily
`removeIf()`	Remove elements based on a condition
`replaceAll()`	Update all elements in a list
`computeIfAbsent()`	Add a value if the key is missing
`computeIfPresent()`	Update a value if the key exists
`merge()`	Combine values in a map
`getOrDefault()`	Handle missing values in a map

Concurrency Enhancements in Java 8: CompletableFuture and Parallel Streams

Java 8 introduced significant improvements in concurrency to make asynchronous programming and parallel computation more efficient and readable. The two major enhancements are:

1. **`CompletableFuture` (java.util.concurrent)**
   - Provides an easy way to write **asynchronous and non-blocking code**.
   - Supports **callback chaining** and exception handling.
   - Helps avoid complex manual thread management.
2. **Parallel Streams (java.util.stream)**
   - Enables **parallel processing** of collections to improve performance.
   - Leverages **multi-core CPUs** automatically.
   - Reduces the need for explicit thread management.

---

1. `CompletableFuture` – Asynchronous Programming

`CompletableFuture` is an extension of `Future` that allows handling asynchronous computations more effectively.

1.1 Why Use `CompletableFuture`?

- Avoids **blocking** the main thread.
- Supports **chaining multiple tasks**.
- Provides built-in **exception handling**.
- Allows combining multiple async operations.

1.2 Basic Example – Running an Async Task

```
import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {
    public static void main(String[] args) {
        CompletableFuture<Void> future =
CompletableFuture.runAsync(() -> {
            System.out.println("Running in: " +
Thread.currentThread().getName());
```

```
        });

        // Wait for completion
        future.join();
    }
}
```

Output:

```
Running in: ForkJoinPool.commonPool-worker-1
```

Key Points:

- `runAsync()` runs the task asynchronously.
- The task executes in a different thread (`ForkJoinPool`).
- `join()` waits for completion.

1.3 Returning a Value with `supplyAsync()`

```
import java.util.concurrent.CompletableFuture;

public class SupplyAsyncExample {
    public static void main(String[] args) {
        CompletableFuture<String> future =
CompletableFuture.supplyAsync(() -> {
            return "Hello from " + Thread.currentThread().getName();
        });

        System.out.println(future.join());  // Blocking call
    }
}
```

Output:

```
Hello from ForkJoinPool.commonPool-worker-1
```

Key Points:

- `supplyAsync()` returns a result (`String`).
- **No need to manually create threads**.

1.4 Chaining Tasks using `thenApply()`

```
import java.util.concurrent.CompletableFuture;

public class ThenApplyExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> 10)
                .thenApply(n -> n * 2)  // Apply function
                .thenApply(n -> n + 5); // Another transformation

        System.out.println(future.join()); // Output: 25
    }
}
```

Transforms result step by step.

1.5 Combining Two `CompletableFutures`

```
import java.util.concurrent.CompletableFuture;

public class CombineFuturesExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future1 =
CompletableFuture.supplyAsync(() -> 10);
        CompletableFuture<Integer> future2 =
CompletableFuture.supplyAsync(() -> 20);

        CompletableFuture<Integer> combined =
future1.thenCombine(future2, (a, b) -> a + b);

        System.out.println(combined.join()); // Output: 30
    }
}
```

Combines results from two async tasks.

1.6 Handling Errors with `exceptionally()`

```
import java.util.concurrent.CompletableFuture;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(() -> {
            if (true) throw new RuntimeException("Something went
wrong");
            return 10;
        }).exceptionally(ex -> {
            System.out.println("Exception: " + ex.getMessage());
            return 0;
        });

        System.out.println(future.join()); // Output: 0
    }
}
```

Handles exceptions without crashing the program.

---

2. Parallel Streams – Faster Data Processing

Parallel Streams use **multiple CPU cores** to process collections concurrently.

2.1 How Parallel Streams Work?

- **Sequential Stream** (Default) – Processes items one by one.
- **Parallel Stream** – Splits data into chunks and processes them in multiple threads.

2.2 Basic Example: Sequential vs Parallel Stream

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

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

        System.out.println("Sequential Stream:");
        names.stream().forEach(name ->
            System.out.println(Thread.currentThread().getName() + "
- " + name));

        System.out.println("\nParallel Stream:");
        names.parallelStream().forEach(name ->
            System.out.println(Thread.currentThread().getName() + "
- " + name));
    }
}
```

Output:

```
Sequential Stream:
main - Alice
main - Bob
main - Charlie
main - David

Parallel Stream:
ForkJoinPool.commonPool-worker-1 - Alice
ForkJoinPool.commonPool-worker-2 - Bob
ForkJoinPool.commonPool-worker-3 - Charlie
ForkJoinPool.commonPool-worker-4 - David
```

Key Points:

- `stream()` runs in a **single thread (`main`)**.
- `parallelStream()` runs **on multiple threads**.

2.3 Processing Large Collections Faster

```
import java.util.stream.IntStream;

public class ParallelProcessingExample {
    public static void main(String[] args) {
        long startTime, endTime;

        // Sequential processing
        startTime = System.currentTimeMillis();
        int sum1 = IntStream.rangeClosed(1, 1000000).sum();
        endTime = System.currentTimeMillis();
        System.out.println("Sequential Sum: " + sum1 + " Time: " +
(endTime - startTime) + " ms");

        // Parallel processing
        startTime = System.currentTimeMillis();
        int sum2 = IntStream.rangeClosed(1,
1000000).parallel().sum();
        endTime = System.currentTimeMillis();
        System.out.println("Parallel Sum: " + sum2 + " Time: " +
(endTime - startTime) + " ms");
    }
}
```

Output (faster in parallel mode)

```
Sequential Sum: 500000500000 Time: 15 ms
Parallel Sum: 500000500000 Time: 4 ms
```

Parallel Streams process large data much faster.

2.4 When to Use Parallel Streams?

Good Cases for Parallel Streams:

- Large datasets (`> 10,000 elements`).
- **CPU-intensive** tasks (sorting, calculations).

- Independent operations (no dependencies).

Avoid Parallel Streams for:

- Small datasets (overhead > performance gain).
- **Mutable shared state** (risk of race conditions).
- IO-bound operations (disk, network).

---

3. Summary: `CompletableFuture` vs Parallel Streams

Feature	`CompletableFuture`	`Parallel Streams`
Purpose	Asynchronous task execution	Parallel processing of collections
Thread Management	Uses `ForkJoinPool` or custom executors	Uses `ForkJoinPool` automatically
Use Case	Running independent tasks, API calls	Processing large datasets faster
Best For	IO-bound tasks (network, DB calls)	CPU-bound tasks (calculations, sorting)

- `CompletableFuture` is ideal for **non-blocking async programming**.
- **Parallel Streams** leverage **multi-core CPUs** for faster collection processing.
- **Both improve performance** but should be used **carefully based on the scenario**.

Base64 Encoding and Decoding API in Java 8

Java 8 introduced the **Base64 API** in the `java.util` package, allowing developers to **encode and decode data** easily without using third-party libraries. This is useful for encoding binary data (such as images, files, and sensitive data) into a text format that can be safely transmitted.

---

1. What is Base64 Encoding?

- **Base64 is a binary-to-text encoding scheme**.
- It converts **binary data** into an ASCII string using **64 characters (A-Z, a-z, 0-9, +, /)**.
- Used in **data transmission, authentication tokens (JWT), and file encoding**.

Example:
Binary → Base64
`Hello` → `SGVsbG8=`

---

2. Base64 API in Java 8

Java 8 provides the `Base64` class with three variants:

1. **Basic Encoding (`Base64.getEncoder()`)** – Standard Base64 encoding.
2. **URL Encoding (`Base64.getUrlEncoder()`)** – Safe for URLs (avoids `+` and `/`).
3. **MIME Encoding (`Base64.getMimeEncoder()`)** – Encodes data in MIME format.

---

3. Base64 Encoding and Decoding in Java

3.1 Basic Encoding and Decoding

```
import java.util.Base64;


public class Base64BasicExample {

    public static void main(String[] args) {

        String originalText = "Hello, Java 8!";


        // Encoding

        String encodedText =
Base64.getEncoder().encodeToString(originalText.getBytes());

        System.out.println("Encoded: " + encodedText);


        // Decoding

        byte[] decodedBytes =
Base64.getDecoder().decode(encodedText);

        String decodedText = new String(decodedBytes);

        System.out.println("Decoded: " + decodedText);

    }

}
```

Output:

```
Encoded: SGVsbG8sIEphdmEgOCE=
```

```
Decoded: Hello, Java 8!
```

Explanation:

- `encodeToString()` converts **bytes to Base64 string**.
- `decode()` converts **Base64 back to original text**.

3.2 Encoding and Decoding Byte Arrays

Base64 is often used to **encode binary data** like images or files.

```
import java.util.Base64;


public class Base64ByteArrayExample {

    public static void main(String[] args) {

        byte[] binaryData = { 1, 2, 3, 4, 5 };


        // Encoding byte array

        String encoded =
Base64.getEncoder().encodeToString(binaryData);

        System.out.println("Encoded: " + encoded);


        // Decoding back to byte array

        byte[] decoded = Base64.getDecoder().decode(encoded);

        System.out.print("Decoded Bytes: ");

        for (byte b : decoded) {

            System.out.print(b + " ");

        }
```

```
    }
}
```

Output:

```
Encoded: AQIDBAU=
Decoded Bytes: 1 2 3 4 5
```

Key Points:

- Base64 is **useful for storing and transmitting binary data** in text format.

3.3 URL-Safe Encoding (`Base64.getUrlEncoder()`)

- Base64 **default encoding uses `+` and `/`**, which are **not URL-safe**.
- **URL Encoding replaces**:
  - `+` → `-`
  - `/` → `_`

```
import java.util.Base64;


public class Base64UrlExample {
    public static void main(String[] args) {
        String url = "https://example.com/?query=java 8";


        // Encoding
        String encodedUrl =
Base64.getUrlEncoder().encodeToString(url.getBytes());
        System.out.println("Encoded URL: " + encodedUrl);
```

```
        // Decoding

        String decodedUrl = new
String(Base64.getUrlDecoder().decode(encodedUrl));

        System.out.println("Decoded URL: " + decodedUrl);

    }

}
```

Output:

```
Encoded URL: aHR0cHM6Ly9leGFtcGxlLmNvbS8_cXVlcnk9amF2YSA4

Decoded URL: https://example.com/?query=java 8
```

Key Points:

- `Base64.getUrlEncoder()` makes encoded strings **safe for URLs and filenames**.

3.4 MIME Encoding (`Base64.getMimeEncoder()`)

- **MIME encoding** is used for **email attachments, large text files**.
- Splits encoded text into **76-character lines**.

```
import java.util.Base64;


public class Base64MimeExample {

    public static void main(String[] args) {

        String longText = "Java 8 introduced Base64 encoding and
decoding.";
```

```java
        // MIME Encoding
        String encodedMime =
Base64.getMimeEncoder().encodeToString(longText.getBytes());
        System.out.println("MIME Encoded:\n" + encodedMime);


        // MIME Decoding
        String decodedMime = new
String(Base64.getMimeDecoder().decode(encodedMime));
        System.out.println("Decoded MIME: " + decodedMime);
    }
}
```

Output:

```
MIME Encoded:
SmF2YSA4IGludHJvZHVjZWQgQmFzZTY0IGVuY29kaW5nIGFuZCBkZWNvZGluZy4=


Decoded MIME: Java 8 introduced Base64 encoding and decoding.
```

Use Cases:

- MIME encoding is useful for **emails and large text blocks**.

---

4. When to Use Base64?

Good Use Cases:

- **Encoding binary data** for safe transmission (images, files).
- **Storing passwords securely** (with additional hashing).
- **Encoding JWT tokens** in authentication.

Not Recommended for:

- **Large files** (Base64 increases size by ~33%).
- **Storing passwords without hashing** (use BCrypt or PBKDF2).

---

5. Summary

Encoding Type	Description	Use Case
Basic (`Base64.getEncoder()`)	Standard Base64 encoding	General-purpose encoding
URL (`Base64.getUrlEncoder()`)	URL-safe encoding	URLs, filenames, JWT tokens
MIME (`Base64.getMimeEncoder()`)	Encodes in **76-char lines**	Emails, large text

---

6. Final Thoughts

- Java 8 Base64 API **eliminates the need for third-party libraries**.
- **Easy and efficient** for encoding and decoding text & binary data.
- Be mindful of **increased data size (33%)** and **avoid storing passwords in Base64**.