Java 8 Features

Prepare by: Abdulmajeeth N – majeethiet@gmail.com

1) Lambda Expressions:

->It's a concise way to represent an anonymous function (function without a name).

-> It is used primarily to define the behaviour of functional interfaces, enabling a functional programming style in Java (An interface with a single abstract method is called a functional interface).

-> Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value.

Functionalities of Lambda Expression:

-> Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

* Enable to treat functionality as a method argument, or code as data.

* A function that can be created without belonging to any class.

* A lambda expression can be passed around as if it was an object and executed on demand.

Syntax:

(lambda operator) -> {body}

Lambda Expression Parameters:

There are three Lambda Expression Parameters are mentioned below:

-> Zero Parameter:

() -> System.out.println("Zero parameter lambda");

-> Single Parameter:

(p) -> System.out.println("One parameter: " + p);

-> Multiple Parameters:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Before Lambda Expression:

```
interface Drawable{
  public void draw();
}
public class LambdaExpressionExample {
  public static void main(String[] args) {
    int width=10;
    //without lambda, Drawable implementation using anonymous class
    Drawable d=new Drawable(){
      public void draw(){
                System.out.println("Drawing "+width);
                }
    };
    d.draw();
  }
}
```

After Lambda Expression:

```
@FunctionalInterface  //It is optional
interface Drawable{
  public void draw();
}

public class LambdaExpressionExample2 {
  public static void main(String[] args) {
    int width=10;
    //with lambda
    Drawable d2=()->{
      System.out.println("Drawing "+width);
    };
```

```
        d2.draw();
    }
}
```

Lambda Expression with Collections:

-> Lambda Expression with Collections is discussed with examples of sorting different collections like ArrayList, TreeSet, TreeMap, etc. Sorting Collections with Comparator (or without Lambda):

-> We can use Comparator interface to sort, It only contains one abstract method: – compare(). An interface that only contains only a single abstract method then it is called a Functional Interface.

While defining our own sorting, JVM is always going to call Comparator to compare() method.

* returns negative value(-1), if and only if obj1 has to come before obj2.

* returns positive value(+1), if and only if obj1 has to come after obj2.

* returns zero(0), if and only if obj1 and obj2 are equal.

-> In List, Set, Map, or anywhere else when we want to define our own sorting method, JVM will always call compare() method internally.

-> When there is Functional Interface concept used, then we can use Lambda Expression in its place. Sorting elements of List(I) with Lambda

Example: Using lambda expression in place of comparator object for defining our own sorting in collections.

```
import java.util.*;
public class Demo {
        public static void main(String[] args)
        {
                ArrayList<Integer> al = new ArrayList<Integer>();
                al.add(205);
                al.add(102);
                al.add(98);
                al.add(275);
                al.add(203);
                System.out.println("Elements of the ArrayList " +
                                                        "before sorting : " + al);
                // using lambda expression in place of comparator object

                Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 :(o1 < o2) ? 1 : 0);

                System.out.println("Elements of the ArrayList after" + " sorting : " + al);

        }

}
```

Thread using Lambda Expressions:

Here we make use of the Runnable Interface. As it is a Functional Interface, Lambda expressions can be used. The following steps are performed to achieve the task:

-> Create the Runnable interface reference and write the Lambda expression for the run() method.

-> Create a Thread class object passing the above-created reference of the Runnable interface since the start() method is defined in the Thread class its object needs to be created.

-> Invoke the start() method to run the thread.

Example:

```
public class Test {

    public static void main(String[] args)

    {

        Runnable myThread = () ->

        {

            Thread.currentThread().setName("myThread");

            System.out.println(Thread.currentThread().getName()+ " is running");

        };

        Thread run = new Thread(myThread);

        run.start();

    }

}
```

2) Functional Interface: Back To Home

-> A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

-> From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

-> A functional interface can have any number of default methods & Static methods

-> Functional Interface is additionally recognized as Single Abstract Method Interfaces. In short, they are also known as SAM interfaces.

-> Functional interfaces are used and executed by representing the interface with an annotation called @FunctionalInterface

Example:

```
// Java program to demonstrate lambda expressions to

// implement a user defined functional interface.

@FunctionalInterface

interface Square {

        int calculate(int x);

}


class Test {

        public static void main(String args[])

        {

                int a = 5;

                Square s = (int x) -> x * x;

                int ans = s.calculate(a);

                System.out.println(ans);

        }

}
```

All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

-> Runnable –> This interface only contains the run() method.

-> Comparable –> This interface only contains the compareTo() method.

-> ActionListener –> This interface only contains the actionPerformed() method.

-> Callable –> This interface only contains the call() method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

-> Consumer

-> Predicate

-> Function

-> Supplier

the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

Consumer -> Bi-Consumer

Predicate -> Bi-Predicate

Function -> Bi-Function, Unary Operator, Binary Operator

Consumer functional interface:

-> Represents an operation that accepts a single input argument and returns no result.

-> Used for performing operations on objects.

Internal Implementations:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
  default Consumer<T> andThen(Consumer<? super T> after) {
    Objects.requireNonNull(after);
    return (T t) -> { accept(t); after.accept(t); };
  }
}
```

Example:

```
public class Example{
public static void main(String args[]){
        Consumer<Employee> consumberObj=(employee)->{
                System.out.println(employee.getName());
                System.out.println(employee.getAge());
        };
        consumberObj.accept(new Employee("Abdulmajeeth","28");
        }
}
```

predicate functional interface:

-> Represents an operation that accepts a single argument to evalute a boolean result

-> Used to perform a test and return a boolean value as result

Internal Implementations:

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
  default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
  }

  default Predicate<T> negate() {
    return (t) -> !test(t);
  }

  default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
  }

  static <T> Predicate<T> isEqual(Object targetRef) {
    return (null == targetRef)
        ? Objects::isNull
        : targetRef::equals;
  }
}
```

Example:

```
public class Example{
        public static void main(String args[]){
                Predicate<Employee> obj=(emp)->emp.getAge()>25;
                boolean result=obj.test(new Employee("Abdulmajeeth",28));
                System.out.println("The test result:"+result);
        }
}
```

Function functional interface:

->Represents a function that accepts one argument and produces a result

->It is used for transforming data

->It takes an argument (Object of type T) and returns an object(Object of type R)

Internal implementation:

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
  default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
  }

  default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
    Objects.requireNonNull(before);
    return (V v) -> apply(before.apply(v));
  }

  static <T> Function<T, T> identity() {
    return t -> t;
  }
```

}

Example:

public class Example{

public static void main(String args[]){

Function<Employee, String> employeeName = Employee::getName;

String name = employeeName.apply(emp1);

}

}

Supplier functional interface:

-> Represents a supplier of results, providing a result of type T without consuming any arguments as a input

-> Used for lazy initialization or providing objects.

Internal implementation:

@FunctionalInterface

public interface Supplier<T> {

  T get();

}

Example:

public class Example {

public static void main(String args[]){

Supplier<Employee> employeeSupplier = () -> new Employee("Jane Doe", 30, 60000);

Employee emp2 = employeeSupplier.get(); // returns new Employee("Jane Doe", 30, 60000)

}

}

UnaryOperator functional interface:

->Represents an operation on a single operand that produces a result of the same type as its operand.

-> Used for operations like increment, negate, etc.

Internal implementations:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

Example:

public class Example{

public static void main(String args[]){

UnaryOperator<Employee> giveRaise = e -> new Employee(e.getName(), e.getAge(), e.getSalary() * 1.10);

Employee emp3 = giveRaise.apply(emp1); // returns new Employee with a 10% salary increase

//or

UnaryOperator<Integer> increment = x -> x + 1;

int result = increment.apply(5); // returns 6

}

}

BinaryOperator Functional Interface:

-> Represents an operation upon two operands of the same type, producing a result of the same type.

-> Used for operations like addition, multiplication, etc

Internal implementation:

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {

    public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;
    }

    public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;
    }
}
```

Example:

public class Example{

public static void main(String args[]){

BinaryOperator<Employee> higherSalary = (e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2;

Employee higherPaid = higherSalary.apply(emp1, emp2); // returns the Employee with the higher salary

//or

BinaryOperator<Integer> add = (a, b) -> a + b;

int result = add.apply(3, 4); // returns 7

}

}

BiFunction Functional Interface:

-> Represents a function that accepts two arguments and produces a result.

-> Used for operations that require two inputs.

Internal implementations:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t, U u) -> after.apply(apply(t, u));
    }
}
```

Example:

A `BiFunction` can accept an `Employee` and a `Double` representing a raise percentage, and return a new `Employee` with the increased salary.

```
public class Example{

        public static void main(String args[]){

        BiFunction<Employee, Double, Employee> applyRaise = (e, percent) -> new
Employee(e.getName(), e.getAge(), e.getSalary() * (1 + percent));

        Employee emp4 = applyRaise.apply(emp1, 0.20); // returns new Employee with a 20% salary
increase

        //Normal Example

        BiFunction<Integer, Integer, String> concatenate = (a, b) -> String.valueOf(a) + b;

        String result = concatenate.apply(1, 2); // returns "12"

        }

}
```

BiConsumer Functional Interface:

-> Represents an operation that accepts two input arguments and returns no result.

-> Used for operations on two objects.

Internal Implementation:

```
@FunctionalInterface
```

```java
public interface BiConsumer<T, U> {

    void accept(T t, U u);

    default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {
        Objects.requireNonNull(after);

        return (l, r) -> {
            accept(l, r);
            after.accept(l, r);
        };
    }
}
```

Example:

A `BiConsumer` can accept an `Employee` and a `Double`, and perform an operation such as printing the employee's name and the raise percentage.

public class Example{

public static void main(String args[]){

BiConsumer<Employee, Double> printRaiseInfo = (e, percent) -> System.out.println(e.getName() + " will get a raise of " + (percent * 100) + "%");

printRaiseInfo.accept(emp1, 0.15); // prints "John Doe will get a raise of 15.0%"

//Normal Example

BiConsumer<String, Integer> print = (s, i) -> System.out.println(s + i);

print.accept("Number: ", 10); // prints "Number: 10"

}

}

BiPredicate Functional Interface:

-> Represents a predicate (boolean-valued function) of two arguments.

-> Used for testing conditions involving two parameters.

Internal Implementation:

```java
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u);

    default BiPredicate<T, U> and(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) && other.test(t, u);
    }

    default BiPredicate<T, U> negate() {
        return (T t, U u) -> !test(t, u);
    }


    default BiPredicate<T, U> or(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) || other.test(t, u);
    }
}
```

Example:

A `BiPredicate` can accept an `Employee` and an `Integer` representing an age, and return a boolean indicating if the employee's age matches the given age.

```
public class Example{

       public static void main(String args[]){

       BiPredicate<Employee, Integer> isAge = (e, age) -> e.getAge() == age;

       boolean ageMatch = isAge.test(emp1, 25); // returns true

       //Normal Example

       BiPredicate<String, Integer> checkLength = (s, i) -> s.length() == i;

       boolean result = checkLength.test("example", 7); // returns true

       }

}
```

3) Method Reference: Back To Home

-> provide a way to refer to methods of existing classes or objects without executing them

-> They are a shorthand notation of lambda expressions to call a method.

-> Method references can be used in conjunction with functional interfaces, making the code more readable and concise.

Types of Method Reference:

->Reference to a Static Method

->Reference to an Instance Method of a Particular Object

->Reference to an Instance Method of an Arbitrary Object of a Particular Type

->Reference to a Constructor

Reference to a Static Method:

To refer a static method will use this method reference.

Syntax: ClassName::methodName

Example:

```
import java.util.Arrays;

import java.util.List;


public class MethodReferenceExample {

   public static void main(String[] args) {

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

      // Using a method reference to a static method
```

```
        numbers.forEach(MethodReferenceExample::printNumber);
//Predefined classes static method reference
       List<String> messages = Arrays.asList("hello", "baeldung", "readers!");
       messages.forEach(StringUtils::capitalize);
    }
    public static void printNumber(Integer number) {
        System.out.println(number);
    }
}
```

Reference to an Instance Method of a Particular Object:

Will be used to refer an Instance Method of a object.

Syntax: instance::methodName

Example:

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

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        // Creating an instance of a class
        MethodReferenceExample example = new MethodReferenceExample();
        // Using a method reference to an instance method of a particular object
        names.forEach(example::printName);
    }
    public void printName(String name) {
        System.out.println(name);
    }
}
```

Reference to an Instance Method of an Arbitrary Object of a Particular Type:

Syntax: ClassName::methodName

Example:

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

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

        // Using a method reference to an instance method of an arbitrary object of a particular type
        names.sort(String::compareToIgnoreCase);

        // Printing the sorted names
        names.forEach(System.out::println);
    }
}
```

Reference to a Constructor:

We can reference a constructor in the same way that we referenced a static method in our first example. The only difference is that we'll use the new keyword.

Syntax: ClassName::new

Example:

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

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        // Using a method reference to a constructor
        Function<String, User> userCreator = User::new;

        // Creating User objects from names
        names.stream()
            .map(userCreator)
            .forEach(System.out::println);
    }
}


class User {
    private String name;


    public User(String name) {
        this.name = name;
    }


    @Override
    public String toString() {
        return "User{name='" + name + "'}";
    }
}
```

4) Optional class: Back To Home

-> It is a container object which may or may not contain a non-null value.

-> The primary purpose of **Optional** is to provide a type-level solution for representing optional values instead of using **null** references, thereby reducing the risk of **NullPointerExceptions** and making the code more readable and expressive.

-> While checking **null != object** is a valid and common approach, using **Optional** can make your code safer and more expressive, especially when dealing with optional return values.

Ways to create optional instance:

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

Optional<String> nonEmptyOptional = Optional.of("Hello, World!");

Optional<String> possiblyNullOptional = Optional.ofNullable(someNullableValue);
```

common methods:

1.isPresent():

Checks if a value is present.

Example:

```
if (nonEmptyOptional.isPresent()) {

    // Do something with the value

}
```

2.ifPresent():

Executes a lambda expression if a value is present.

Example:

```
nonEmptyOptional.ifPresent(value -> System.out.println(value));
```

3. orElse():

Returns the value if present, otherwise returns a default value.

Example:

```
String result = possiblyNullOptional.orElse("Default Value");
```

4. orElseGet():

Returns the value if present, otherwise executes a Supplier and returns its result.

Example:

```
String result = possiblyNullOptional.orElseGet(() -> "Generated Default Value");
```

5. orElseThrow():

Returns the value if present, otherwise throws an exception.

Example:

```
String result = possiblyNullOptional.orElseThrow(() -> new NoSuchElementException("No
value present"));
```

6. get():

Returns the value if present, otherwise throws NoSuchElementException.

Example:

```
String value = nonEmptyOptional.get();
```

Advanced Methods:

1. map():

Applies a function to the value if present, and returns a new Optional with the result.

Example:

```
Optional<Integer> lengthOptional = nonEmptyOptional.map(String::length);
```

2. flatMap():

Similar to map(), but the function must return an Optional. This avoids nested Optionals.

Example:

```
Optional<Integer> lengthOptional = possiblyNullOptional.flatMap(value ->
Optional.of(value.length()));
```

3. filter():

Returns an Optional containing the value if it matches the given predicate, otherwise returns an empty Optional.

Example:

```
Optional<String> filteredOptional = nonEmptyOptional.filter(value -> value.startsWith("H"));
```

Complete Example:

```
import java.util.Optional;

public class OptionalExample {

  public static void main(String[] args) {

    Optional<String> nonEmptyOptional = Optional.of("Hello, World!");

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

    Optional<String> nullableOptional = Optional.ofNullable(getNullableString());

    // isPresent

    if (nonEmptyOptional.isPresent()) {

      System.out.println("Value is present: " + nonEmptyOptional.get());

    } else {

      System.out.println("Value is not present");

    }


    // ifPresent

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


    // orElse

    String valueOrDefault = nullableOptional.orElse("Default Value");

    System.out.println("Value or Default: " + valueOrDefault);


    // orElseGet

    String valueOrGenerated = nullableOptional.orElseGet(() -> "Generated Value");

    System.out.println("Value or Generated: " + valueOrGenerated);
```

```
        // orElseThrow
        try {
            String valueOrException = emptyOptional.orElseThrow(() -> new IllegalArgumentException("No value present"));
            System.out.println("Value or Exception: " + valueOrException);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        // map
        Optional<Integer> lengthOptional = nonEmptyOptional.map(String::length);
        lengthOptional.ifPresent(length -> System.out.println("Length of string: " + length));

        // flatMap
        Optional<Integer> flatMappedOptional = nonEmptyOptional.flatMap(value -> Optional.of(value.length()));
        flatMappedOptional.ifPresent(length -> System.out.println("FlatMapped length: " + length));

        // filter
        Optional<String> filteredOptional = nonEmptyOptional.filter(value -> value.startsWith("H"));
        filteredOptional.ifPresent(value -> System.out.println("Filtered value: " + value));
    }

    private static String getNullableString() {
        // This can return a string or null based on some condition
        return null;
    }
}
```

Benefits of Using Optional

->Eliminates Null Checks: Using Optional encourages more explicit handling of null values, making the code less prone to NullPointerExceptions.

->Improves Readability: Code using Optional can be more readable and self-explanatory.

->Functional Programming Style: Optional aligns with the functional programming style introduced in Java 8, working seamlessly with lambda expressions and streams.

Traditional Null Check Approach:

```
public class NullCheckExample {
    public static void main(String[] args) {
        String value = getNullableString();
         if (value != null) {
            System.out.println("Value is present: " + value);
        } else {
            System.out.println("Value is null");
        }
    }:



    private static String getNullableString() {
        // This can return a string or null based on some condition
        return null;
    }
}
```

Using Optional

```
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        Optional<String> optionalValue = Optional.ofNullable(getNullableString());
        // Using isPresent()
        if (optionalValue.isPresent()) {
            System.out.println("Value is present: " + optionalValue.get());
        } else {
            System.out.println("Value is not present");
        }
        // Using ifPresent()
        optionalValue.ifPresent(value -> System.out.println("Value using ifPresent: " + value));

        // Using orElse()
        String valueOrDefault = optionalValue.orElse("Default Value");
        System.out.println("Value or Default: " + valueOrDefault);

        // Using orElseGet()
        String valueOrGenerated = optionalValue.orElseGet(() -> "Generated Value");
        System.out.println("Value or Generated: " + valueOrGenerated);

        // Using orElseThrow()
        try {
            String valueOrException = optionalValue.orElseThrow(() -> new IllegalArgumentException("No
value present"));
            System.out.println("Value or Exception: " + valueOrException);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    private static String getNullableString() {
        // This can return a string or null based on some condition
        return null;
    }
}
```

5) Default & Static method: Back To Home

-> Default methods and static methods in interfaces enhance the flexibility and capabilities of interfaces, allowing for more functional and modular code

Default Methods

Default methods, also known as defender methods or virtual extension methods, are methods in an interface that have a default implementation.

They enable interfaces to evolve without breaking the classes that implement them.

Syntax:

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

Key Points

Backward Compatibility: Default methods allow you to add new methods to interfaces without breaking existing implementations. This is particularly useful for evolving APIs.

Multiple Inheritance: Default methods provide a way to achieve multiple inheritance of behavior (method implementations) in Java.

Example:

```
public interface Vehicle {
  void start();

  default void stop() {
    System.out.println("Vehicle is stopping");
  }
}

public class Car implements Vehicle {
  @Override
  public void start() {
    System.out.println("Car is starting");
  }
  // No need to override the stop() method unless custom behavior is needed
}
```

```
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start(); // Output: Car is starting
        myCar.stop();  // Output: Vehicle is stopping
    }
}
```

Static Methods:

->Static methods in interfaces are similar to static methods in classes.

->They belong to the interface itself rather than to instances of the interface.

Syntax:

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

Key Points

Utility Methods: Static methods can be used to define utility methods related to the interface, such as factory methods.

No Inheritance: Static methods are not inherited by implementing classes. They can be called directly on the interface.

Example:

```
public interface MathUtil {
    static int add(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        int sum = MathUtil.add(5, 3); // Output: 8
        System.out.println("Sum: " + sum);
    }
}
```

Detailed Comparison:

Default Methods

Purpose: Provide default behavior for methods in an interface.

Invocation: Called on instances of implementing classes.

Usage: Useful for extending interfaces without breaking existing code.

Inheritance: Can be overridden by implementing classes.

Static Methods

Purpose: Provide utility methods related to the interface.

Invocation: Called on the interface itself, not on instances.

Usage: Useful for utility functions or factory methods.

Inheritance: Not inherited by implementing classes.

Practical Use Cases:

Default Methods

API Evolution: Adding methods to interfaces in large codebases without forcing all implementing classes to provide implementations.

Multiple Behaviour Inheritance: Combining behaviours from multiple interfaces.

Static Methods

Utility Methods: Providing common functionality related to the interface, such as validation or calculation.

Factory Methods: Creating instances of the implementing classes.

Conflict Resolution in Default Methods:

When multiple interfaces provide conflicting default methods, the implementing class must resolve the conflict:

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

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

public class C implements A, B {
    @Override
    public void show() {
        A.super.show(); // Or B.super.show();
    }

    public static void main(String[] args) {
        new C().show(); // Output: Default A (or Default B, depending on resolution)
    }
}
```

Conclusion

Default and static methods in Java 8 interfaces provide significant flexibility and power. They enable backward-compatible evolution of interfaces, reduce boilerplate code, and facilitate better modularization and reusability of code. Understanding how to use these features effectively can lead to more maintainable and robust Java applications.

6) Stream API: Back To Home

->It introduced to support functional-style operations on sequences of elements, such as collections

->It provides a more declarative way to process data compared to traditional loops, emphasizing what to do rather than how to do it

->It returns another stream as a result, they can be chained together to form a pipeline of operations

Stream: A sequence of elements supporting sequential and parallel aggregate operations.

Intermediate Operations: Transform a stream into another stream. They are lazy, meaning they are not executed until a terminal operation is invoked.

Terminal Operations: Produce a result or a side-effect and mark the end of the stream processing.

Creating Streams:

Streams can be created from various data sources such as collections, arrays, or generating functions.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;


public class StreamCreationExample {

  public static void main(String[] args) {

//From a Collection

    List<String> myList = Arrays.asList("a", "b", "c");

    Stream<String> stream = myList.stream();

    stream.forEach(System.out::println);


//From an Array

    String[] myArray = {"a", "b", "c"};

    Stream<String> streamFromArray = Arrays.stream(myArray);

    streamFromArray.forEach(System.out::println);


//Using Stream.of

    Stream<String> streamOf = Stream.of("a", "b", "c");

    streamOf.forEach(System.out::println);


//Infinite Streams
```

```
        Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);
        infiniteStream.limit(10).forEach(System.out::println);
    }
}
```

Intermediate Operations:

Intermediate operations return a new stream and are lazy:

1.map():

->Transforms each element using a function.

->Will apply the function to the elements of the stream and transform the same and will return the elements as a stream

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class MapExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("a", "b", "c");
        List<String> upperCaseList = myList.stream()
                                   .map(String::toUpperCase)
                                   .collect(Collectors.toList());
        System.out.println(upperCaseList);
    }
}
```

2.filter():

->It is used to filter elements as per the Predicate passed as an argument.

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class FilterExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("abc", "bcd", "cde");
        List<String> filteredList = myList.stream()
                                .filter(s -> s.startsWith("b"))
                                .collect(Collectors.toList());
        System.out.println(filteredList);
    }
}
```

3.sorted():

->The sorted method is used to sort the stream.

->Sorts elements based on a comparator.

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class SortedExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("b", "a", "c");
        List<String> sortedList = myList.stream()
                                .sorted()
                                .collect(Collectors.toList());
        System.out.println(sortedList);
    }
}
```

4.flatMap():

->Transforms each element to a stream, and flattens the resulting streams into a single stream.

->A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations:

Example:

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

```
public class FlatMapExample {
    public static void main(String[] args) {
        List<List<String>> list = Arrays.asList(
            Arrays.asList("a", "b"),
            Arrays.asList("c", "d")
        );
        List<String> flatMapList = list.stream()
                                .flatMap(List::stream)
                                .collect(Collectors.toList());
        System.out.println(flatMapList);
    }
}
```

5.peek():

-> It's an intermediate operation that allows you to perform a side-effect action on each element as it is processed in the pipeline.

-> It is typically used for debugging purposes, such as logging or inspecting elements during stream operations.

Example:

```
import java.util.Arrays;
import java.util.List;
public class PeekExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("one", "two", "three", "four");
        myList.stream()
              .filter(s -> s.length() > 3) // Intermediate operation: filter
              .peek(s -> System.out.println("Filtered value: " + s)) // Intermediate operation: peek
              .map(String::toUpperCase) // Intermediate operation: map
              .peek(s -> System.out.println("Mapped value: " + s)) // Intermediate operation: peek
              .forEach(System.out::println); // Terminal operation: forEach
    }
}
```

6.distinct():

-> It returns a stream with duplicate elements removed, based on the equals() method of the elements.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class DistinctExample {

  public static void main(String[] args) {

    List<String> myList = Arrays.asList("one", "two", "three", "one", "two");

    List<String> distinctList = myList.stream()

                                      .distinct() // Intermediate operation: distinct

                                      .collect(Collectors.toList()); // Terminal operation: collect

    System.out.println(distinctList);

  }

}
```

7.limit():

->It returns a stream truncated to be no longer than the specified number of elements.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class LimitExample {

  public static void main(String[] args) {

    List<String> myList = Arrays.asList("one", "two", "three", "four", "five");

    List<String> limitedList = myList.stream()

                                     .limit(3) // Intermediate operation: limit

                                     .collect(Collectors.toList()); // Terminal operation: collect

    System.out.println(limitedList);

  }

}
```

Terminal Operations:

Terminal operations produce a result or a side-effect and trigger the intermediate operations:

1.forEach:

Performs an action for each element.

Example:

```
import java.util.Arrays;

import java.util.List;

public class ForEachExample {

  public static void main(String[] args) {

    List<String> myList = Arrays.asList("a", "b", "c");

    myList.stream().forEach(System.out::println);

  }

}
```

2.collect:

Accumulates the elements into a collection or other containers.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class CollectExample {

  public static void main(String[] args) {

    List<String> myList = Arrays.asList("a", "b", "c");

    List<String> collectedList = myList.stream().collect(Collectors.toList());

    System.out.println(collectedList);

  }

}
```

3.reduce:

Reduces the elements to a single value using an associative accumulator.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class ReduceExample {

  public static void main(String[] args) {

    List<String> myList = Arrays.asList("a", "b", "c");

    Optional<String> concatenated = myList.stream()
```

```
                                    .reduce((s1, s2) -> s1 + s2);
        concatenated.ifPresent(System.out::println);
    }
}
```

4.count:

Counts the number of elements.

Example:

```
import java.util.Arrays;
import java.util.List;
public class CountExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("a", "b", "c");
        long count = myList.stream().count();
        System.out.println(count);
    }
}
```

5.anyMatch, allMatch, noneMatch:

Check if any, all, or none of the elements match a given predicate.

Example:

```
import java.util.Arrays;
import java.util.List;
public class MatchExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("a", "b", "c");

        boolean anyMatch = myList.stream().anyMatch(s -> s.startsWith("a"));
        System.out.println("Any match: " + anyMatch);

        boolean allMatch = myList.stream().allMatch(s -> s.length() == 1);
        System.out.println("All match: " + allMatch);

        boolean noneMatch = myList.stream().noneMatch(s -> s.startsWith("z"));
        System.out.println("None match: " + noneMatch);
    }
}
```

6.findAny():

-> The findAny() method is a terminal operation that returns an Optional<T> describing some element of the stream, or an empty Optional if the stream is empty.

-> This method is particularly useful in parallel streams because it can return any element from the stream, potentially providing better performance.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class FindAnyExample {

    public static void main(String[] args) {

        List<String> myList = Arrays.asList("one", "two", "three", "four");

        Optional<String> anyElement = myList.stream()

                                   .filter(s -> s.length() > 3)

                                   .findAny(); // Terminal operation: findAny

        anyElement.ifPresent(System.out::println); // Print the found element, if present

    }

}
```

7.findFirst():

-> It returns an Optional<T> describing the first element of the stream, or an empty Optional if the stream is empty.

-> It guarantees to return the first element in the encounter order of the stream.

Example:

```
import java.util.Arrays;

import java.util.List;

import java.util.Optional;

public class FindFirstExample {

    public static void main(String[] args) {

        List<String> myList = Arrays.asList("one", "two", "three", "four");

        Optional<String> firstElement = myList.stream()
```

```
                                    .filter(s -> s.length() > 3)
                                    .findFirst(); // Terminal operation: findFirst
        firstElement.ifPresent(System.out::println); // Print the first element, if present
    }
}
```

8.min():

-> It returns the minimum element of the stream according to the provided Comparator.

-> It returns an Optional<T> containing the minimum element, or an empty Optional if the stream is empty.

Example:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
public class MinExample {
    public static void main(String[] args) {
        List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
        Optional<Integer> minElement = myList.stream()
                                    .min(Comparator.naturalOrder()); // Terminal operation: min
        minElement.ifPresent(System.out::println); // Print the minimum element, if present
    }
}
```

9.max():

-> It returns the maximum element of the stream according to the provided Comparator.

-> It returns an Optional<T> containing the maximum element, or an empty Optional if the stream is empty.

Example:

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
public class MaxExample {
    public static void main(String[] args) {
        List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
        Optional<Integer> maxElement = myList.stream()
                                  .max(Comparator.naturalOrder()); // Terminal operation: max
        maxElement.ifPresent(System.out::println); // Print the maximum element, if present
    }
}
```

10.toArray():

-> It returns an array containing the elements of the stream.

-> You can either use the default method to get an Object[] or provide an array generator to get an array of a specific type.

Example:

```
import java.util.Arrays;
import java.util.List;
public class ToArrayExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("one", "two", "three", "four");
        // Using default toArray method
        Object[] objectArray = myList.stream().toArray(); // Terminal operation: toArray
        System.out.println(Arrays.toString(objectArray));

        // Using array generator to get a String array
        String[] stringArray = myList.stream().toArray(String[]::new); // Terminal operation: toArray
        System.out.println(Arrays.toString(stringArray));
    }
}
```

Parallel Streams

-> provide a powerful and flexible way to leverage multi-core processors for parallel processing.

-> By processing collections of data in parallel, you can achieve significant performance improvements for large data sets or computationally intensive operations.

Creation of Parallel Streams:

-> You can create a parallel stream from an existing collection or stream using the parallelStream() method

or by converting a sequential stream to a parallel stream with the parallel() method.

Example:

```
import java.util.Arrays;
import java.util.List;
public class ParallelStreamExample {
  public static void main(String[] args) {
    List<String> myList = Arrays.asList("one", "two", "three", "four", "five");

    // Create a parallel stream from a collection
    myList.parallelStream()
        .forEach(System.out::println);

    // Convert a sequential stream to a parallel stream
    myList.stream()
        .parallel()
        .forEach(System.out::println) ;
  }
}
```

Example: Sum of Square:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class ParallelStreamSumOfSquares {
  public static void main(String[] args) {
    // Generate a list of integers from 1 to 1,000,000
    List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000)
                .boxed()
                .collect(Collectors.toList());

    // Sequential Stream
    long startTime = System.currentTimeMillis();
    int sumOfSquaresSequential = numbers.stream()
                .mapToInt(x -> x * x)
                .sum();
    long endTime = System.currentTimeMillis();
```

```
        System.out.println("Sequential Sum of Squares: " + sumOfSquaresSequential);
        System.out.println("Time taken (Sequential): " + (endTime - startTime) + " ms");

        // Parallel Stream
        startTime = System.currentTimeMillis();
        int sumOfSquaresParallel = numbers.parallelStream()
                                 .mapToInt(x -> x * x)
                                 .sum();
        endTime = System.currentTimeMillis();
        System.out.println("Parallel Sum of Squares: " + sumOfSquaresParallel);
        System.out.println("Time taken (Parallel): " + (endTime - startTime) + " ms");
    }
}
```

Customizing the ForkJoinPool:

-> You can customize the ForkJoinPool used by parallel streams to control the number of threads or other parameters

Example:

```
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class CustomForkJoinPoolExample {
    public static void main(String[] args) {
        // Generate a list of integers from 1 to 1,000,000
        List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000)
                                .boxed()
                                .collect(Collectors.toList());
        ForkJoinPool customThreadPool = new ForkJoinPool(4); // Customize the number of threads
        try {
            customThreadPool.submit(() -> {
                int sumOfSquaresParallel = numbers.parallelStream()
                                         .mapToInt(x -> x * x)
                                         .sum();
                System.out.println("Custom Thread Pool Sum of Squares: " + sumOfSquaresParallel);
            }).get();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            customThreadPool.shutdown();
        }
    }
}
```

7) New Date and Time API: Back To Home

-> Introduced a comprehensive new Date and Time API to address many of the shortcomings of the previous java.util.Date and java.util.Calendar classes.

-> The new API is located in the java.time package and its subpackages. This API is based on the ISO-8601 calendar system and is designed to be intuitive, flexible, and less error-prone.

-> It represents a local date-time object without timezone information.

-> The LocalDateTime class in Java is an immutable date-time object that represents a date in the yyyy-MM-dd-HH-mm-ss.zzz format.

-> It implements the ChronoLocalDateTime interface and inherits the object class.

-> Wherever we need to represent time without a timezone reference, we can use the LocalDateTime instances.

-> LocalDateTime, for example, can be used to start batch jobs in any application. Jobs will be run at a fixed time in the timezone in which the server is located.

Key Classes:

1. LocalDate:

Represents a date without a time zone, such as 2024-06-22.

Creation:

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

LocalDate specificDate = LocalDate.of(2022, Month.JUNE, 22);
```

Methods:

```
int year = specificDate.getYear();

Month month = specificDate.getMonth();
```

```
int dayOfMonth = specificDate.getDayOfMonth();

LocalDate nextWeek = today.plusWeeks(1);
LocalDate previousMonth = today.minusMonths(1);
boolean isLeapYear = specificDate.isLeapYear();
```

2. LocalTime

Represents a time without a date and time zone, such as 13:45:30.

Creation:

```
LocalTime now = LocalTime.now();
LocalTime specificTime = LocalTime.of(13, 45, 30);
```

Methods:

```
int hour = specificTime.getHour();
int minute = specificTime.getMinute();
int second = specificTime.getSecond();

LocalTime nextHour = now.plusHours(1);
LocalTime previousMinute = now.minusMinutes(1);
```

3. LocalDateTime

Represents a date-time without a time zone, combining LocalDate and LocalTime, such as 2024-06-22T13:45:30.

Creation:

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime specificDateTime = LocalDateTime.of(2024, Month.JUNE, 22, 13, 45, 30);
```

Methods:

```
LocalDate datePart = specificDateTime.toLocalDate();
LocalTime timePart = specificDateTime.toLocalTime();

LocalDateTime nextDay = now.plusDays(1);
```

LocalDateTime previousHour = now.minusHours(1);

4. ZonedDateTime

Represents a date-time with a time zone, such as 2024-06-22T13:45:30+02:00[Europe/Paris].

Creation:

ZonedDateTime now = ZonedDateTime.now();

ZonedDateTime specificZonedDateTime = ZonedDateTime.of(2024, Month.JUNE, 22, 13, 45, 30, 0, ZoneId.of("Europe/Paris"));

Methods:

ZoneId zone = specificZonedDateTime.getZone();

ZonedDateTime nextHour = now.plusHours(1);

ZonedDateTime previousDay = now.minusDays(1);

ZonedDateTime utcDateTime = specificZonedDateTime.withZoneSameInstant(ZoneId.of("UTC"));

5. Instant

Represents a moment on the timeline in UTC, such as 2024-06-22T11:45:30Z.

Creation:

Instant now = Instant.now();

Instant specificInstant = Instant.ofEpochSecond(1624273530);

Methods:

long epochSecond = now.getEpochSecond();

Instant nextSecond = now.plusSeconds(1);

Instant previousSecond = now.minusSeconds(1);

6. Duration and Period

Duration: Measures an amount of time in seconds and nanoseconds, suitable for machine time.

Duration twoHours = Duration.ofHours(2);

Duration betweenInstants = Duration.between(instant1, instant2);

Period: Measures an amount of time in years, months, and days, suitable for human time.

```
Period tenDays = Period.ofDays(10);
Period betweenDates = Period.between(date1, date2);
```

Parsing and Formatting:

DateTimeFormatter: Used for parsing and formatting dates and times.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
LocalDateTime dateTime = LocalDateTime.parse("2024-06-22 13:45:30", formatter);
String formattedDate = dateTime.format(formatter);
```

Example Complete Code:

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
public class DateTimeExample {
  public static void main(String[] args) {
    // LocalDate example
    LocalDate today = LocalDate.now();
    LocalDate birthDate = LocalDate.of(1990, Month.JANUARY, 1);
    System.out.println("Today: " + today);
    System.out.println("Birth Date: " + birthDate);

    // LocalTime example
    LocalTime now = LocalTime.now();
    LocalTime specificTime = LocalTime.of(15, 30);
    System.out.println("Now: " + now);
    System.out.println("Specific Time: " + specificTime);

    // LocalDateTime example
    LocalDateTime dateTime = LocalDateTime.of(today, now);
    System.out.println("DateTime: " + dateTime);

    // ZonedDateTime example
```

```
        ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("America/New_York"));
        System.out.println("ZonedDateTime: " + zonedDateTime);

        // Instant example
        Instant instant = Instant.now();
        System.out.println("Instant: " + instant);

        // Duration and Period example
        Duration duration = Duration.ofHours(5);
        Period period = Period.ofDays(10);
        System.out.println("Duration: " + duration);
        System.out.println("Period: " + period);

        // DateTimeFormatter example
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedDateTime = dateTime.format(formatter);
        System.out.println("Formatted DateTime: " + formattedDateTime);

        // Parsing example
        LocalDateTime parsedDateTime = LocalDateTime.parse("2024-06-22 15:30:00", formatter);
        System.out.println("Parsed DateTime: " + parsedDateTime);
    }
}
```

8) Type Annotations: Back To Home

-> Type Annotations were introduced in Java 8 as part of JSR 308, which extended the Java annotation system to allow annotations to be written in more places than was previously possible.

-> Type Annotations can be applied to any use of a type, including declarations of class instances, method parameters, and even generic type parameters. This makes them useful for tools and frameworks that need more detailed type information.

Use Cases for Type Annotations:

Static Analysis Tools: Type Annotations can be used by static analysis tools to detect errors or enforce coding standards. For example, nullability annotations (@NonNull, @Nullable) help tools check for null pointer exceptions.

Code Generation and Transformation: Tools that generate or transform code can use Type Annotations to guide their operations. For instance, frameworks can use them to inject dependencies or to perform aspect-oriented programming tasks.

Runtime Checking: Some annotations can be used to perform checks at runtime. However, this requires that the annotations be retained at runtime and the appropriate runtime libraries be available to process them

Examples of Type Annotations

To illustrate how Type Annotations work, let's consider some practical examples:

1. Declaring Type Annotations:

First, define some custom annotations:

```
import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

@Target(ElementType.TYPE_USE)

@interface NonNull {}

@Target(ElementType.TYPE_USE)

@interface Nullable {}
```

The @Target(ElementType.TYPE_USE) specifies that these annotations can be used in any place where a type is used.

2. Applying Type Annotations

Here are several examples of how to apply these annotations in different contexts:

```
import java.util.List;

public class TypeAnnotationExample {

  // Annotation on a field
```

```
    private @NonNull String nonNullField;

    // Annotation on a local variable
    public void exampleMethod() {
        @Nullable String possibleNull = getNullableString();
    }

    // Annotation on a method return type
    public @Nullable String getNullableString() {
        return null;
    }

    // Annotation on a method parameter
    public void setNonNullField(@NonNull String nonNullField) {
        this.nonNullField = nonNullField;
    }

    // Annotation on a generic type parameter
    public void processStrings(List<@NonNull String> strings) {
        for (String string : strings) {
            // Processing string
        }
    }
}
```

Retention Policies:

Annotations have different retention policies that determine where the annotations are available:

SOURCE: The annotation is retained only in the source code and is discarded by the compiler.

CLASS: The annotation is retained in the class file but is not available at runtime.

RUNTIME: The annotation is retained in the class file and is available at runtime through reflection.

Example:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```java
@Retention(RetentionPolicy.RUNTIME)
@interface NonNull {}
```

Tools for Type Annotation Processing:

Several tools and libraries can process Type Annotations:

Checker Framework: A powerful tool for adding pluggable type-checking to Java. It provides many built-in type systems (e.g., nullness, immutability) and allows you to create custom type systems.

Lombok: A library that reduces boilerplate code in Java. While not directly related to Type Annotations, it makes use of annotations to generate code at compile time.

Example with Checker Framework:

Using the Checker Framework to check for nullability issues might look like this:

Add the Checker Framework to your project (e.g., via Maven):

```xml
<dependency>
  <groupId>org.checkerframework</groupId>
  <artifactId>checker</artifactId>
  <version>3.11.0</version>
</dependency>
```

Annotate your code:

```java
import org.checkerframework.checker.nullness.qual.NonNull;
import org.checkerframework.checker.nullness.qual.Nullable;
public class Example {
  private @NonNull String nonNullField;
  public void setNonNullField(@NonNull String nonNullField) {
    this.nonNullField = nonNullField;
  }
  public @Nullable String getNullableString() {
    return null;
  }
}
```

Run the Checker Framework:

javac -processor org.checkerframework.checker.nullness.NullnessChecker Example.java

This will analyze your code for nullability issues based on the annotations.

9) Concurrency API Improvements: Back To Home

Java 8 also brought significant improvements to the Concurrency API, making it easier to write concurrent and parallel programs.

1. CompletableFuture:

CompletableFuture is a versatile and powerful feature for asynchronous programming.

Manually Completing a Future:

```
CompletableFuture<String> future = new CompletableFuture<>();

// In some other thread

future.complete("Hello World");
```

Using Factory Methods:

supplyAsync(Supplier<T> supplier): Runs a task asynchronously and returns a CompletableFuture holding the task's result.

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello World");
```

runAsync(Runnable runnable): Runs a task asynchronously without returning a result.

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

   // Perform some asynchronous computation

});
```

Combining Futures:

You can chain multiple CompletableFuture instances to perform sequential operations.

thenApply: Transforms the result of a CompletableFuture when it completes.

```
future.thenApply(result -> result + " from CompletableFuture");
```

thenAccept: Consumes the result of a CompletableFuture when it completes.

```
future.thenAccept(result -> System.out.println(result));
```

thenCombine: Combines the results of two CompletableFutures.

```
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> 10);

CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 20);


CompletableFuture<Integer> combinedFuture = future1.thenCombine(future2, Integer::sum);
```

```
combinedFuture.thenAccept(System.out::println);  // Output: 30
```

Handling Exceptions:

You can handle exceptions that occur during the computation.

exceptionally: Handles exceptions and provides an alternative result.

```
future.exceptionally(ex -> "Error: " + ex.getMessage());
```

handle: Processes both result and exceptions.

```
future.handle((result, ex) -> {
    if (ex != null) {
        return "Error: " + ex.getMessage();
    } else {
        return result;
    }
});
```

CompletableFuture in REST API Calls for Asynchronous Processing:

We can use this completableFuture object for Resp API process in asynchronous way.

Controller Code:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.CompletableFuture;

@RestController
public class AsyncController {
    private final AsyncService asyncService;

    @Autowired
    public AsyncController(AsyncService asyncService) {
        this.asyncService = asyncService;
    }

    @GetMapping("/posts/{id}")
    public CompletableFuture<ResponseEntity<String>> getPost(@PathVariable int id) {
        return asyncService.fetchPost(id)
                .thenApply(ResponseEntity::ok)
                .exceptionally(ex -> ResponseEntity.status(500).body("Error: " + ex.getMessage()));
    }

    @GetMapping("/users/{id}")
    public CompletableFuture<ResponseEntity<String>> getUser(@PathVariable int id) {
        return asyncService.fetchUser(id)
                .thenApply(ResponseEntity::ok)
                .exceptionally(ex -> ResponseEntity.status(500).body("Error: " + ex.getMessage()));
    }

    @GetMapping("/posts/{id}/comments")
    public CompletableFuture<ResponseEntity<String>> getComments(@PathVariable int id) {
        return asyncService.fetchComments(id)
                .thenApply(ResponseEntity::ok)
                .exceptionally(ex -> ResponseEntity.status(500).body("Error: " + ex.getMessage()));
    }
}
```

Service Code:

Here we have used Web client for Async call.

```
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

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

```
@Service
public class AsyncService {

  private final WebClient webClient;

  public AsyncService(WebClient.Builder webClientBuilder) {
    this.webClient = webClientBuilder.baseUrl("https://jsonplaceholder.typicode.com").build();
  }

  public CompletableFuture<String> fetchPost(int postId) {
    return webClient.get()
        .uri("/posts/{id}", postId)
        .retrieve()
        .bodyToMono(String.class)
        .toFuture();
  }

  public CompletableFuture<String> fetchUser(int userId) {
    return webClient.get()
        .uri("/users/{id}", userId)
        .retrieve()
        .bodyToMono(String.class)
        .toFuture();
  }

  public CompletableFuture<String> fetchComments(int postId) {
    return webClient.get()
        .uri("/posts/{id}/comments", postId)
        .retrieve()
        .bodyToMono(String.class)
        .toFuture();
  }
}
```

2. Parallel Streams

Streams can be parallelized to leverage multi-core architectures.

Creating Parallel Streams: You can create parallel streams using parallelStream() or by converting a sequential stream to a parallel stream using parallel().

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

int sum = list.parallelStream()

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

        .mapToInt(Integer::intValue)

        .sum();
```

3. New Classes and Methods

ConcurrentHashMap: Enhanced with methods like forEach(), reduce(), search(), and merge() for bulk operations.

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

map.put("A", 1);

map.put("B", 2);

map.forEach(1, (key, value) -> System.out.println(key + ": " + value));
```

StampedLock: A new type of lock that offers an alternative to ReadWriteLock with improved performance for certain scenarios.

```
StampedLock lock = new StampedLock();

long stamp = lock.readLock();

try {

  // read operations

} finally {

  lock.unlockRead(stamp);

}
```

ForkJoinPool: Enhanced to support a common pool, which can be used by parallel streams and CompletableFuture.

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```