# Java 8 Features

## Why Lambda?

* Eliminates Boilerplate of Code
* Enables functional programming in Java
* Readable and very concise code
* Easier to use APIs & libraries
* Enables support of parallel processing
* Implements functional interface

## Dev Setup:

Install Eclipse IDE

## Simple program to pring Hello World message

**Way 1:**

**public** **class** Greeter {

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

System.***out***.println("Hello word");

}

}

**Way 2 through Interface:**

**public** **interface** Greeting {

**void** wish();

}

**class** HelloWorldGreeting **implements** Greeting{

@Override

**public** **void** wish() {

System.***out***.println("Hello World");

}

}

**public** **class** Greeter {

**public** **void** greet(Greeting g) {

g.wish();

}

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

Greeter greeter = **new** Greeter();

Greeting hw = **new** HelloWorldGreeting(); //Implemented class

greeter.greet(hw);

}

}

**Way 3 through Anonymous inner class:**

**public** **class** Greeter {

**public** **void** greet(Greeting g) {

g.wish();

}

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

Greeter greeter = **new** Greeter();

Greeting greeting = **new** Greeting() {

@Override

**public** **void** wish() {

System.***out***.println("Hello World!");

}

};

greeter.greet(greeting);

}

}

## Functional vs OO Programming

Functional Programming enables Readable, concise, and maintainable code

## Code in OOP

Everything is an Object

All code blocks are associated with Class & Object

A lot of code in OOP

Don’t need a class, just need a piece of code???

# Lambda Expression syntax

() -> {//statements}

Greeting greeting = () -> System.***out***.println("Hello World!");

greeting.wish();

## Passing Behavior/Action as input in OOP

Passing thing that has behavior

Lambda expressions are functions don’t belong to anything

public void greet(action){

action()

}

Assign method to variable. It is possible with Lambda expression

**public** **interface** Greeting {

**void** wish();

}

**public** **class** Greeter {

**public** **void** greet(Greeting g) {

g.wish();

}

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

Greeter greeter = **new** Greeter();

Greeting greeting = () -> System.***out***.println("Hello World!");

// sent complete action as method argument

// greeter.greet(() -> System.out.println("Hello World!"));

// sending Functional interface reference

greeter.greet(greeting);

}

}

## Type Inference

Way 1:

**public** **class** TypeInferenceEx {

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

// StringLengthLambda lambda = (String s)-> s.length();

// StringLengthLambda lambda = (s)-> s.length(); //input type is optional

StringLengthLambda lambda = s -> s.length(); // paranthesis is option for signle parameter

**int** len = lambda.getLength("Hello World");

System.***out***.println("Given string length: " + len);

}

}

**interface** StringLengthLambda {

**int** getLength(String s);

}

Way 2: passing behavior as input to static method

**ublic** **class** TypeInferenceEx {

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

*printStringLenLambda*(s -> s.length());

}

**public** **static** **void** printStringLenLambda(StringLengthLambda sl) {

System.***out***.println("Length of String: "+sl.getLength("Hello Lambda@"));

}

}

**interface** StringLengthLambda {

**int** getLength(String s);

}

## Runnable using Lambdas

//Without Lambda

Thread t = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Hello World!");

}

});

t.run();

## With Lambda

**public** **class** RunnableEx {

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

Thread t = **new** Thread(() -> System.***out***.println("run method"));

t.run();

}

}

## Lambda Expressions Exercise

## Java 7 Exercise with Anonymous class

//printing list of sorted elements

//printing list of elements which have first name starts with “M”

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**public** **class** Java7Exercise {

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"),

**new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"),

**new** Person(4, "Kohli", "Virat"));

//sort list by first name

Collections.*sort*(people, **new** Comparator<Person>() {

@Override

**public** **int** compare(Person p1, Person p2) {

**return** p1.getFirstName().compareTo(p1.getLastName());

}

});

System.***out***.println("sorted list by firstName");

*printAll*(people);

//print all elements which first name starts with M

System.***out***.println("\*\*\*Elements that first name starts with M");

*printAllFirstNameStartsWith*(people, **new** Condition() {

@Override

**public** **boolean** test(Person p) {

**return** p.getFirstName().startsWith("M");

}

});

}

**private** **static** **void** printAllFirstNameStartsWith(List<Person> person, Condition c) {

**for** (Person p : person) {

**if**(c.test(p))

System.***out***.println(p);

}

}

**private** **static** **void** printAll(List<Person> person) {

**for** (Person p : person) {

System.***out***.println(p);

}

}

}

**interface** Condition{

**public** **boolean** test(Person p);

}

## Java 8 Exercise with Lambda Expressions

**public** **class** Java8LambdaExercise {

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"), **new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"), **new** Person(4, "Kohli", "Virat"));

// sort list by first name

Collections.*sort*(people, (p1, p2) -> p1.getFirstName().compareTo(p1.getLastName()));

System.***out***.println("sorted list by firstName");

*printAllFirstNameStartsWith*(people, p -> **true**);

// print all elements which first name starts with M

System.***out***.println("\*\*\*Elements that first name starts with M");

*printAllFirstNameStartsWith*(people, p -> p.getFirstName().startsWith("M"));

}

**private** **static** **void** printAllFirstNameStartsWith(List<Person> person, Condition c) {

**for** (Person p : person) {

**if** (c.test(p)) {

System.***out***.println(p);

}

}

}

}

**interface** Condition{

**public** **boolean** test(Person p);

}

# Default Methods

Methods that are defined inside the interface and declared with default are known as default methods.

The concept of default method is used to define a method with default implementation

These methods are non-abstract methods.

# Static Method inside Interface

Static methods are used to define utility methods in Java 8 interface

# Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class almost similar except that you can **create constructor in the abstract class** whereas you can't do this in interface.

# Functional Interface

Interface contains only one abstract method.

It can contain zero or more static, default, and object methods.

Functional interface is also known as Single Abstract Method interface or SAM interface.

@FunctionalInterface

**interface** Greeting {

**void** wish();

// void bless(); - can't have two abstract methods

**public** **default** **double** piValue() {

**return** Math.***PI***;

}

@Override

**int** hashCode();

@Override

String toString();

**public** **default** **void** sayHello() {

System.***out***.println("say Hello");

}

**public** **default** **void** saySomething() {

System.***out***.println("say something");

}

**public** **static** **void** sayHi() {

System.***out***.println("Hi");

}

**public** **static** **void** sayBye() {

System.***out***.println("Bye");

}

}

**public** **class** FunctionalInterfaceEx **implements** Greeting {

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

//calling static method

Greeting.*sayHi*();

//calling default method

**new** FunctionalInterfaceEx().saySomething();

}

@Override

**public** **void** wish() {

System.***out***.println("Wish method is implemented");

}

}

## Java Predefined-Functional Interfaces

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

Own functional interface also can be implemented to deal with functional programming.

## Predefined Function Interfaces List

|  |  |
| --- | --- |
| Interface | Description |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

## Using Functional Interfaces – Predicate

Note: No need to create functional interface until predefined Functional interface unavailable to use as assignment target for Lambda expressions.

Predicate represents predicate (Boolean value) of the method.

It has test() and other methods.. Refer Java DOc

**private** **static** **void** printAllFirstNameStartsWith(List<Person> person, Predicate<Person> c) {

**for** (Person p : person) {

**if** (c.test(p)) {

System.***out***.println(p);

}

}

}

No change in other places.

## Using Functional Interfaces – Consumer

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side effects.

Consumer interface has accept() and other methods.. Please refer Java doc

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"), **new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"), **new** Person(4, "Kohli", "Virat"));

// sort list by first name

Collections.*sort*(people, (p1, p2) -> p1.getFirstName().compareTo(p1.getLastName()));

System.***out***.println("sorted list by firstName");

*printAllFirstNameStartsWith*(people, p -> **true**, p-> System.***out***.println(p));

// print all elements which first name starts with M

System.***out***.println("\*\*\*Elements that first name starts with M");

*printAllFirstNameStartsWith*(people, p -> p.getFirstName().startsWith("M"), p-> System.***out***.println(p.getFirstName()));

}

**private** **static** **void** printAllFirstNameStartsWith(List<Person> person, Predicate<Person> c, Consumer<Person> consumer) {

**for** (Person p : person) {

**if** (c.test(p)) {

consumer.accept(p);

}

}

}

## Exception Handling in Lambda

**Way 1: Handling exception in process method – Not recommended**

**import** java.util.function.BiConsumer;

**public** **class** ExceptionHandlerLambda {

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

**int**[] numbers = { 10, 20, 30, 40 };

**int** div = 0; // change div value to Non-zero and check

// process(numbers, div, (k, v) -> System.out.println(k + "+" + v + "= " + (k + v)));

//process(numbers, div, (k, v) -> System.out.println(k + "-" + v + "= " + (k - v)));

//process(numbers, div, (k, v) -> System.out.println(k + "\*" + v + "= " + (k \* v)));

*process*(numbers, div, (k, v) -> System.***out***.println(k + "/" + v + "= " + (k / v)));

}

**public** **static** **void** process(**int**[] numbers, **int** div, BiConsumer<Integer, Integer> biConsumer) {

**for** (**int** i : numbers) {

**try** {

biConsumer.accept(i, div);

} **catch** (ArithmeticException e) {

System.***err***.println("Exception caught in process method");

}

}

}

}

**Way 2: Handling exception in Lambda expression (Not recommended)**

**Code increases**

*process*(numbers, div, (k, v) -> {

**try** {

System.***out***.println(k + "/" + v + "= " + (k / v));

} **catch** (ArithmeticException e) {

System.***err***.println("Exception caught in Lambda");

}

});

**Way 3: Handling the exception in separate method (wrapper method) – recommended**

**public** **class** ExceptionHandlerLambda {

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

**int**[] numbers = { 10, 20, 30, 40 };

**int** div = 10; // change div value to Non-zero and check

// process(numbers, div, (k, v) -> System.out.println(k + "+" + v + "= " + (k + v)));

//process(numbers, div, (k, v) -> System.out.println(k + "-" + v + "= " + (k - v)));

//process(numbers, div, (k, v) -> System.out.println(k + "\*" + v + "= " + (k \* v)));

*process*(numbers, div, *wrapperLambda*((k, v) -> System.***out***.println(k + "/" + v + "= " + (k / v))));

}

**public** **static** BiConsumer<Integer, Integer> wrapperLambda(BiConsumer<Integer, Integer> biConsumer){

System.***out***.println("Wrapper Lambda method is called");

// return biConsumer;

// return (k, v) -> System.out.println(k + "/" + v + "= " + (k / v));

**return** (k,v)->

{

biConsumer.accept(k, v);

};

}

**public** **static** **void** process(**int**[] numbers, **int** div, BiConsumer<Integer, Integer> biConsumer) {

**for** (**int** i : numbers) {

biConsumer.accept(i, div);

}

}

}

## Closures in Lambda

**public** **class** ClosureExx {

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

**int** a=10;

**int** b=20; //consider it as final

/\* Anonymous class

\* doProcess(a, new Process() { //b=30; //error

\*

\* @Override public void process() { System.out.println("Closure example sum: "+

\* (a+b) ); } });

\*/

//Lambda

*doProcess*(a, () -> System.***out***.println("Closure example :"+ (a+b)));

}

**public** **static** **void** doProcess(**int** i, Process p) {

p.process();

}

}

**interface** Process{

**void** process();

}

## this reference in Lambdas

**this usage in Anonymous implementation inside main(static) method**

**public** **class** ThisRefEx {

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

*doProcess*(10, **new** Process() {

@Override

**public** **void** process() {

System.***out***.println("Anonymous class ");

System.***out***.println("this object " + **this**);

}

@Override

**public** String toString() {

**return** "this object from Anonymous implementation";

}

});

}

**public** **static** **void** doProcess(**int** i, Process p) {

p.process();

}

}

**interface** Process {

**void** process();

}

**this usage in Lambda**

**public** **class** ThisRefEx {

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

**int** i = 10;

ThisRefEx thisRefEx=**new** ThisRefEx();

thisRefEx.execute(i);

/\*

\* doProcess(i, () -> { System.out.println("Anonymous class ");

\* System.out.println("this object "+this); //error });

\*/

}

**private** **void** execute(**int** i) {

*doProcess*(i, () -> {

System.***out***.println("Lambda expression i="+i);

System.***out***.println("this object "+**this**); //error

});

}

**public** **static** **void** doProcess(**int** i, Process p) {

p.process();

}

@Override

**public** String toString() {

**return** " from ThisRefEx";

}

}

# Method References

Method Reference is alternative syntax for Lambda expressions.

It is used to refer a method of Functional interface.

It increases the readability of the code.

Each time when you are using Lamda expression just to refer a method, you can replace the Lambda expression with Method reference.

() -> method()

(p) -> method(p)

Method is called with Instance or class name

## Types of Method References

* Reference to a static method.
* Reference to an instance method.
* Reference to a constructor.

## Method References Examples

## Ex1: Reference to a Static Method by Referring User Defined functional interface

@FunctionalInterface

**interface** Say {

**void** sayHi();

}

**public** **class** MethodRefEx {

**public** **static** **void** doSay() {

System.***out***.println("Hi...");

}

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

Say s = MethodRefEx::*doSay*;

}

}

## Ex2: Reference to a Instance Method by Referring User Defined functional interface

public class MethodRefEx {

public void doSay() {

System.out.println("Hi...");

}

public static void main(String[] args) {

Say s = new MethodRefEx()::doSay;

}

}

## Ex3: Referring predefined functional interface Runnable

**public** **class** MethodRefEx {

**public** **static** **void** doSay() {

System.***out***.println("Hi.....");

}

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

Thread t = **new** Thread(MethodRefEx::*doSay*);

t.start();

}

}

## Ex4: Referring predefined functional interface BiFunction

**public** **class** MethodRefEx {

**public** **static** **int** add(**int** n1, **int** n2) {

**return** n1+n2;

}

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

BiFunction<Integer, Integer, Integer> biFun = MethodRefEx::*add*;

**int** res = biFun.apply(10, 20);

System.***out***.println(res);

}

}

## Ex5: Reference to Constructor

@FunctionalInterface

**interface** Say {

MethodRefEx sayHi();

}

**public** **class** MethodRefEx {

**public** MethodRefEx() {

System.***out***.println("MethodRefEx is called");

}

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

Say s = MethodRefEx::**new**;

s.sayHi();

}

}

**public** **class** MethodRefEx {

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

//Thread t = new Thread(() -> doProcess());

Thread t = **new** Thread(MethodRefEx::*doProcess*);

t.start();

}

**private** **static** **void** doProcess() {

System.***out***.println("Hello");

}

}

## Ex6: Advanced Example

**public** **class** MethodRefEx2{

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"), **new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"), **new** Person(4, "Kohli", "Virat"));

// print all elements which first name starts with M

System.***out***.println("\*\*\*Elements that first name starts with M");

//printAllFirstNameStartsWith(people, p -> p.getFirstName().startsWith("M"), p-> System.out.println(p));

*printAllFirstNameStartsWith*(people, p -> p.getFirstName().startsWith("M"), System.***out***::println);

}

**private** **static** **void** printAllFirstNameStartsWith(List<Person> person, Predicate<Person> c, Consumer<Person> consumer) {

**for** (Person p : person) {

**if** (c.test(p)) {

consumer.accept(p);

}

}

}

}

# Collections

**Foreach consumer**

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** ForEachIterationLambdaEx {

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"), **new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"), **new** Person(4, "Kohli", "Virat"));

System.***out***.println("Printing elements with for index");

**for** (**int** i = 0; i < people.size(); i++) {

System.***out***.println(people.get(i));

}

System.***out***.println("Printing elements with for in..");

**for** (Person person : people) {

System.***out***.println(person);

}

System.***out***.println("Printing elements with Lambda");

people.forEach(p -> System.***out***.println(p));

System.***out***.println("Printing elements with Method reference");

people.forEach(System.***out***::println);

}

}

## Streams

Stream is a new view of Collection. It comprises of 3 elements

1. Source - Collection
2. Operations that to be performed on stream - Filter
3. End condition

Streams bring functional programming to Java, and are supported starting Java 8.

## Advantages of Streams:

- Efficient Java programming

- Make heavy use of lambda expressions

- parallelstreams make it very east to multi-thread operations

A stream pipeline consists of source followed by zero or intermediate operations; and a terminal point.

Source-> Filter -> Sort -> Map -> Collect

## Stream source:

Collections, Lists, Sets, ints, doubles, arrays, lines of a file

Streams operations are either intermediate or terminal

-Intermediate operations: Filter, map or sort return a stream. Order matters for large datasets: filter first and then sort or map. For very large datasets, use parallelstreams to enable multi threads.

operations include: filter(), map(), flatmap(),skip(),sorted() anyMatch(), distinct(), findFirst().

- Terminal operations: forEach, collect or reduce return void or non-stream results.

forEach(), collect() - collects all elements into collection, min(),max(),count(), reduce(),

summaryStatistics()

### Ex1: foreach

IntStream.*range*(1, 10).forEach(System.***out***::println);

### Ex2: skip

IntStream.*range*(1, 10).skip(5).forEach(System.***out***::println);

### Ex3: sum

**int** sum = IntStream.*range*(1, 5).sum();

System.***out***.println("Sum: "+sum);

### Ex4: sort

Stream.*of*("Banana","Pine apple", "Apple").sorted().forEach(System.***out***::println);

### Ex5: sort findFirst

Stream.*of*("Banana","Pine apple", "Apple").sorted().findFirst().ifPresent(System.***out***::println);

### Ex6: filter

String[] names= {"Kohli", "Dhoni", "Karthik", "Pandey"};

Arrays.*stream*(names).filter(x -> x.startsWith("K")).sorted().forEach(a->System.***out***.println(a));

### Ex7: map – transforms object from one form to another form

Arrays.*stream*(**new** **int**[] {1,2,3}).map(a->a\*10).forEach(a->System.***out***.println(a));

### Ex8: map with average

Arrays.*stream*(**new** **int**[] {1,2,3}).map(a->a\*10).average().ifPresent(System.***out***::println);

### Ex9: Stream with Filter and collect

List<String> newList = Arrays.*stream*(**new** String[] {"Kohli","Dhoni","Karthik", "Pandey",}).filter(a-> a.startsWith("K")).collect(Collectors.*toList*());

newList.forEach(System.***out***::println);

### Ex10: Reduce – reduces stream into a single value

//(((2\*3)\*4)\*5)

//ignore last one

**int** product = IntStream.*of*(2,6).reduce((num1,num2)-> num1\*num2).orElse(-1);

System.***out***.println("The product is: "+product);

### Ex11: Stream, Filter, forEach

**public** **class** StreamEx {

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

List<Person> people = Arrays.*asList*(**new** Person(1, "Manga", "Rao"), **new** Person(2, "Dhoni", "Mahindra"),

**new** Person(3, "Mohan", "Kumar"), **new** Person(4, "Kohli", "Virat"));

// people.stream().forEach(p -> System.out.println(p));

people.stream().filter(p -> p.getFirstName().startsWith("M")).forEach(p -> System.***out***.println(p));

### Ex12: Stream, filter, count //count ()

**long** count = people.stream().filter(p-> p.getFirstName().startsWith("M")).count();

System.***out***.println("Count: "+count);

// stream - sequential stream

System.***out***.println("Stream output");

people.stream().forEach(p -> System.***out***.println(p));

System.***out***.println("Parallel Stream..");

people.parallelStream().forEach(p -> System.***out***.println(p));

}

}

# Optional Class

Introduced in Java 8

It is a final class, present in Java.util package

Not participated in any Inheritance

It avoids NullPointerException by dealing with NullPointerException in Java application

Without Optional Interface, not null check to be added where the object is being used.

String, Wrapper types, Reference types may throw the Null Pointer Exception.

Optional class has a set of methods to check presence of value in a particular variable.

## Optional class Methods:

|  |  |
| --- | --- |
| Methods | Description |
| public static <T> Optional<T> empty() | It returns an empty Optional object. No value is present for this Optional. |
| public static <T> Optional<T> of(T value) | It returns an Optional with the specified present non-null value. |
| public static <T> Optional<T> ofNullable(T value) | It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.  ofNullable() = of() + empty() methods |
| public T get() | If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| public boolean isPresent() | It returns true if there is a value present, otherwise false. |
| public void ifPresent(Consumer<? super T> consumer) | If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| public Optional<T> filter(Predicate<? super T> predicate) | If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| public <U> Optional<U> map(Function<? super T,? extends U> mapper) | If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional. |
| public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper) | If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| public T orElse(T other) | It returns the value if present, otherwise returns other. |
| public T orElseGet(Supplier<? extends T> other) | It returns the value if present, otherwise invoke other and return the result of that invocation. |
| public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable | It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| public boolean equals(Object obj) | Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:   * It is also an Optional and; * Both instances have no value present or; * the present values are "equal to" each other via equals(). |
| public int hashCode() | It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present. |
| public String toString() | It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions. |

## Ex1 - Java Program Without Optional Class Usage

String[] names = **new** String[5];

System.***out***.println(names[2].toLowerCase());

Output: NullPointerException

## Ex2 - Java Program with Optional Class If value is absent

String[] names = **new** String[5];

Optional<String> CheckNames = Optional.*ofNullable*(names[2]);

**if**(CheckNames.isPresent()) {

System.***out***.println(names[2].toLowerCase());

}**else** {

System.***out***.println("No value is present");

}

Output: No value is present

## Ex3 - Java Program with Optional Class If value is present

String[] names = **new** String[5];

names[2] = "Java optional class practice";

Optional<String> CheckNames = Optional.*ofNullable*(names[2]);

**if**(CheckNames.isPresent()) {

System.***out***.println(names[2].toLowerCase());

//System.out.println(CheckNames.get().toLowerCase());

}**else** {

System.***out***.println("No value is present");

}

Output: Java optional class practice

## Ex4 - Java Program with Optional Class using ifPresent(), get(), Lambda & Method reference

String[] names = **new** String[5];

//names[2] = "Java optional class practice";

Optional<String> CheckNames = Optional.*ofNullable*(names[2]);

//using lambda expression

CheckNames.ifPresent(val -> System.***out***.println(val));

//using Method reference

CheckNames.ifPresent(System.***out***::println);

**if**(CheckNames.isPresent())

System.***out***.println(CheckNames.get());

**else**

System.***out***.println("No value");

## Ex5 - Java Program with Optional Class using empty(), of(), ofNullable(), filter(), hashCode(), orElse()

String[] names = **new** String[5];

names[2] = "Java optional class practice";

//empty method usage

Optional<Object> empty = Optional.*empty*();

System.***out***.println(empty);

System.***out***.println(empty.isEmpty());

//of method usage

Optional<String> of = Optional.*of*(names[2]);

System.***out***.println(of);

//OfNullable

Optional<String> ofNullable = Optional.*ofNullable*(names[2]);

//filter example

System.***out***.println(ofNullable.filter(s -> s.equals("abc")));

//hashCode

System.***out***.println(ofNullable.hashCode());

//orElse

System.***out***.println(empty.orElse("value is not present on empty"));

System.***out***.println(ofNullable.orElse("value is not present on ofNullable"));