JAVA 8

**Lambda Expressions**

Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

#### 1.1. Lambda Syntax

A typical lambda expression syntax will be like this:

|  |
| --- |
| (x, y) -> x + y  //This function takes two parameters                      //and return their sum. |

Now based on type of x and y, method may be used in multiple places. Parameters can match to int, or Integer or simply String also. Based on context, it will either add two integers or concat two strings.

Syntax:

The other possible syntaxes of a lambda expression are:

|  |
| --- |
| either  (parameters) -> expression           //1  or  (parameters) -> { statements; }  //2  or  () -> expression                     //3 |

#### 1.2. Lambda examples

Let’s see some examples of lambda expressions as well:

|  |
| --- |
| (int a, int b) ->    a \* b       // takes two integers and returns their multiplication    (a, b)          ->   a - b        // takes two numbers and returns their difference   () -> 99                                // takes no values and returns 99   (String a) -> System.out.println(a) // takes a string, prints its value to the console, and returns nothing   a -> 2 \* a                 // takes a number and returns the result of doubling it   c -> { //some complex statements }   // takes a collection and do some procesing |

Functional Interface : (**Single Abstract Method interfaces** (SAM Interfaces)

It means **interfaces with only one single method**. From java 8, they will also be **referred as functional interfaces as well**. Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. **@FunctionalInterface**.

Let’s understand it with an example. If we have to write a thread which will print “howtodoinjava” in console then simplest code will be:

|  |
| --- |
| new Thread(new Runnable() {      @Override      public void run() {          System.out.println("howtodoinjava");      }  }).start(); |

If we use the lambda expression for this task then code will be :

|  |
| --- |
| new Thread(              () ->   {                          System.out.println("My Runnable");                      }           ).start(); |

We have also see that Runnable is an functional interface with single method run(). So, when you pass lambda expression to constructor of Thread class, compiler tries to convert the expression into equivalent Runnable code as shown in first code sample.

**1) Iterating over a List and perform some operations**

|  |
| --- |
| List<String> pointList = new ArrayList();  pointList.add("1");  pointList.add("2");    pointList.forEach(p ->  {                              System.out.println(p);                              //Do more work                          }                   ); |

public class Java8Tester {

public static void main(String args[]) {

Java8Tester tester = new Java8Tester();

//with type declaration

MathOperation addition = (int a, int b) -> a + b;

//with out type declaration

MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces

MathOperation multiplication = (int a, int b) -> { return a \* b; };

//without return statement and without curly braces

MathOperation division = (int a, int b) -> a / b;

System.out.println("10 + 5 = " + tester.operate(10, 5, addition));

System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));

System.out.println("10 / 5 = " + tester.operate(10, 5, division));

//without parenthesis

GreetingService greetService1 = message ->

System.out.println("Hello " + message);

//with parenthesis

GreetingService greetService2 = (message) ->

System.out.println("Hello " + message);

greetService1.sayMessage("Mahesh");

greetService2.sayMessage("Suresh");

}

interface MathOperation {

int operation(int a, int b);

}

interface GreetingService {

void sayMessage(String message);

}

private int operate(int a, int b, MathOperation mathOperation) {

return mathOperation.operation(a, b);

}

}

# Java 8 - Method References

Method references help to point to methods by their names. A method reference is described using "::" symbol. A method reference can be used to point the following types of methods −

* Static methods
* Instance methods
* Constructors using new operator (TreeSet::new)

## Method Reference Example

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

### Java8Tester.java

import java.util.List;

import java.util.ArrayList;

public class Java8Tester {

public static void main(String args[]) {

List names = new ArrayList();

names.add("Mahesh");

names.add("Suresh");

names.add("Ramesh");

names.add("Naresh");

names.add("Kalpesh");

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

}

}

# Java 8 - Functional Interfaces

Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose.

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default**, static methods but can contain only one abstract method**. **It can also declare methods of object class**.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces.

### Example 1

1. @FunctionalInterface
2. **interface** sayable{
3. **void** say(String msg);
4. }
5. **public** **class** FunctionalInterfaceExample **implements** sayable{
6. **public** **void** say(String msg){
7. System.out.println(msg);
8. }
9. **public** **static** **void** main(String[] args) {
10. FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();
11. fie.say("Hello there");
12. }
13. }

A functional interface can have methods of object class. See in the following example.

### Example 2

2. @FunctionalInterface
3. **interface** sayable{
4. **void** say(String msg);   // abstract method
5. // It can contain any number of Object class methods.
6. **int** hashCode();
7. String toString();
8. **boolean** equals(Object obj);
9. }
10. **public** **class** FunctionalInterfaceExample2 **implements** sayable{
11. **public** **void** say(String msg){
12. System.out.println(msg);
13. }
14. **public** **static** **void** main(String[] args) {
15. FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();
16. fie.say("Hello there");
17. }
18. }

There are around **40+ functional interfaces** under **java.util.function** [package](http://package.in/). Let’s discuss the important ones:**Predicate, Consumer, Function, and Supplier.**

**Predicate :**A predicate is a function that returns a value that is either true or false. (Predicate takes as input some object and returns boolean. It’s a special case of Function.)

**Function :**This functional interface represents a function that accepts one argument and produces a result. (Function can take any object and returns any object.)

**Consumer :**This represents an operation that accepts a single input argument and returns no result. The real outcome is the side-effects it produces. (

**Supplier :**This does the opposite of the Consumer, it takes no arguments but it returns some value. It may return different values when it is being called more than once.

Java Predicate Interface

It is a functional interface which represents a predicate (boolean-valued function) of one argument. It is defined in the java.util.function package and contains test() a functional method.

### Java Predicate Interface Example 1

1. **import** java.util.function.Predicate;
2. **public** **class** PredicateInterfaceExample {
3. **public** **static** **void** main(String[] args) {
4. Predicate<Integer> pr = a -> (a > 18); // Creating predicate
5. System.out.println(pr.test(10));    // Calling Predicate method
6. }
7. }
8. **import** java.util.function.Predicate;
9. **public** **class** PredicateInterfaceExample {
10. **static** Boolean checkAge(**int** age){
11. **if**(age>17)
12. **return** **true**;
13. **else** **return** **false**;
14. }
15. **public** **static** **void** main(String[] args){
16. // Using Predicate interface
17. Predicate<Integer> predicate =  PredicateInterfaceExample::checkAge;
18. // Calling Predicate method
19. **boolean** result = predicate.test(25);
20. System.out.println(result);
21. }
22. }

Java Function Interface

It is a functional interface. It is used to refer method by specifying type of parameter. It returns a result back to the referred function.

1. // importing Function interface
2. **import** java.util.function.Function;
3. **public** **class** FunctionInterfaceExample {
4. **static** String show(String message){
5. **return** "Hello "+message;
6. }
7. **public** **static** **void** main(String[] args) {
8. // Function interface referring to a method
9. Function<String, String> fun = FunctionInterfaceExample::show;
10. // Calling Function interface method
11. System.out.println(fun.apply("Peter"));
12. }
13. }

### Java Function Interface Example 2

1. // importing Function interface
2. **import** java.util.function.Function;
3. **import** java.util.List;
4. **import** java.util.ArrayList;
5. **public** **class** FunctionInterfaceExample {
6. **static** Integer addList(List<Integer> list){
7. **return** list.stream()
8. .mapToInt(Integer::intValue)
9. .sum();
10. }
11. **public** **static** **void** main(String[] args) {
12. // Creating a list and adding values
13. List<Integer> list = **new** ArrayList<Integer>();
14. list.add(10);
15. list.add(20);
16. list.add(30);
17. list.add(40);
18. // Referring addList() method
19. Function<List<Integer>, Integer> fun = FunctionInterfaceExample::addList;
20. // Calling Function interface method
21. **int** result = fun.apply(list);
22. System.out.println("Sum of list values: "+result);
23. }
24. }

Output:

Sum of list values : 100

Java Consumer Interface

### Java Consumer Interface Example 1

1. // Importing Consumer interface
2. **import** java.util.function.Consumer;
3. **public** **class** ConsumerInterfaceExample {
4. **static** **void** printMessage(String name){
5. System.out.println("Hello "+name);
6. }
7. **static** **void** printValue(**int** val){
8. System.out.println(val);
9. }
10. **public** **static** **void** main(String[] args) {
11. // Referring method to String type Consumer interface
12. Consumer<String> consumer1 = ConsumerInterfaceExample::printMessage;
13. consumer1.accept("John");   // Calling Consumer method
14. // Referring method to Integer type Consumer interface
15. Consumer<Integer> consumer2 = ConsumerInterfaceExample::printValue;
16. consumer2.accept(12);   // Calling Consumer method
17. }
18. }

### Java Consumer Interface Example2

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **import** java.util.function.Consumer;
4. **public** **class** ConsumerInterfaceExample {
5. **static** **void** addList(List<Integer> list){
6. // Return sum of list values
7. **int** result = list.stream()
8. .mapToInt(Integer::intValue)
9. .sum();
10. System.out.println("Sum of list values: "+result);
11. }
12. **public** **static** **void** main(String[] args) {
13. // Creating a list and adding values
14. List<Integer> list = **new** ArrayList<Integer>();
15. list.add(10);
16. list.add(20);
17. list.add(30);
18. list.add(40);
19. // Referring method to String type Consumer interface
20. Consumer<List<Integer>> consumer = ConsumerInterfaceExample::addList;
21. consumer.accept(list);  // Calling Consumer method
23. }
24. }

Output:

Sum of list values: 100

Java BiConsumer Interface

BiConsumer Interface accepts two input arguments and does not return any result.

1. **import** java.util.function.BiConsumer;
2. **public** **class** BiConsumerInterfaceExample {
3. **static** **void** ShowDetails(String name, Integer age){
4. System.out.println(name+" "+age);
5. }
6. **public** **static** **void** main(String[] args) {
7. // Referring method
8. BiConsumer<String, Integer> biCon = BiConsumerInterfaceExample::ShowDetails;
9. biCon.accept("Rama", 20);
10. biCon.accept("Shyam", 25);
11. // Using lambda expression
12. BiConsumer<String, Integer> biCon2 = (name, age)->System.out.println(name+" "+age);
13. biCon2.accept("Peter", 28);
14. }
15. }
16. **import** java.util.function.BiConsumer;
17. **import** java.util.HashMap;
18. **import** java.util.Map;
19. **public** **class** BiConsumerInterfaceExample {
20. **static** **void** ShowDetails(Map<Integer, String> map, String mapName){
21. System.out.println("----------"+mapName+" records-----------");
22. map.forEach((key, val)->System.out.println(key+" "+val));
23. }
24. **public** **static** **void** main(String[] args) {
25. Map<Integer, String> map = **new** HashMap<Integer,String>();
26. map.put(100, "Mohan");
27. map.put(110, "Sujeet");
28. map.put(115, "Tom");
29. map.put(120, "Danish");
30. // Referring method
31. BiConsumer<Map<Integer, String>, String> biCon = BiConsumerInterfaceExample::ShowDetails;
32. biCon.accept(map, "Student");
33. }
34. }

Output :

----------Student records-----------

115 Tom

100 Mohan

120 Danish

110 Sujeet

Java 8 - Default Methods

Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8.

For example, **‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration**. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

## Syntax

public interface vehicle {

default void print() {

System.out.println("I am a vehicle!");

}

}

## Multiple Defaults

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.

public interface vehicle {

default void print() {

System.out.println("I am a vehicle!");

}

}

public interface fourWheeler {

default void print() {

System.out.println("I am a four wheeler!");

}

}

First solution is to create an own method that overrides the default implementation.

public class car implements vehicle, fourWheeler {

public void print() {

System.out.println("I am a four wheeler car vehicle!");

}

}

Second solution is to call the default method of the specified interface using super.

public class car implements vehicle, fourWheeler {

default void print() {

vehicle.super.print();

}

}

## Static Default Methods

An interface can also have static helper methods from Java 8 onwards.

public interface vehicle {

default void print() {

System.out.println("I am a vehicle!");

}

static void blowHorn() {

System.out.println("Blowing horn!!!");

}

}

## Default Method Example

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

### Java8Tester.java

[Live Demo](http://tpcg.io/xYtZNn)

public class Java8Tester {

public static void main(String args[]) {

Vehicle vehicle = new Car();

vehicle.print();

}

}

interface Vehicle {

default void print() {

System.out.println("I am a vehicle!");

}

static void blowHorn() {

System.out.println("Blowing horn!!!");

}

}

interface FourWheeler {

default void print() {

System.out.println("I am a four wheeler!");

}

}

class Car implements Vehicle, FourWheeler {

public void print() {

Vehicle.super.print();

FourWheeler.super.print();

Vehicle.blowHorn();

System.out.println("I am a car!");

}

}

It should produce the following output −

I am a vehicle!

I am a four wheeler!

Blowing horn!!!

I am a car!

Java 8 - Streams

Using collections framework in Java, a developer has to use loops and make repeated checks.

## What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream –

* **Sequence of elements** − A **stream provides a set of elements of specific type in a sequential manner.** A stream gets/computes elements on demand. It never stores the elements.
* **Source** − Stream takes **Collections, Arrays, or I/O resources** as input source.
* **Aggregate operations** − Stream supports aggregate operations like **filter, map, limit**, reduce, find, match, and so on.
* **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. **collect() method is a terminal operation** which is normally present at the end of the pipelining operation to mark the end of the stream.
* **Automatic iterations** − **Stream operations do the iterations internally over** the source elements provided, in contrast to Collections where explicit iteration is required.

With Java 8, Collection interface has two methods to generate a Stream.

* **stream()** − Returns a sequential stream considering collection as its source.
* **parallelStream()** − Returns a parallel Stream considering collection as its source.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

## forEach

Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## map

The ‘map’ method is used to map each element to its corresponding result. **The following code segment prints unique squares of numbers using map.**

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares

List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());

## filter

The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.stream().filter(string -> string.isEmpty()).count();

## limit

The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## sorted

The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

## Parallel Processing

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

long count = strings.parallelStream().filter(string -> string.isEmpty()).count();

It is very easy to switch between sequential and parallel streams.

Stream implementation in Java is by default sequential unless until it is explicitly mentioned in parallel. When a stream executes in parallel, the Java runtime partitions the stream into multiple sub-streams. Aggregate operations iterate over and process these sub-streams in parallel and then combine the results.

## Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

## Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

## Stream Example

import java.util.ArrayList;

import java.util.Arrays;

import java.util.IntSummaryStatistics;

import java.util.List;

import java.util.Random;

import java.util.stream.Collectors;

import java.util.Map;

public class Java8Tester {

public static void main(String args[]) {

System.out.println("Using Java 7: ");

// Count empty strings

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

System.out.println("List: " +strings);

long count = getCountEmptyStringUsingJava7(strings);

System.out.println("Empty Strings: " + count);

count = getCountLength3UsingJava7(strings);

System.out.println("Strings of length 3: " + count);

//Eliminate empty string

List<String> filtered = deleteEmptyStringsUsingJava7(strings);

System.out.println("Filtered List: " + filtered);

//Eliminate empty string and join using comma.

String mergedString = getMergedStringUsingJava7(strings,", ");

System.out.println("Merged String: " + mergedString);

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of square of distinct numbers

List<Integer> squaresList = getSquares(numbers);

System.out.println("Squares List: " + squaresList);

List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);

System.out.println("List: " +integers);

System.out.println("Highest number in List : " + getMax(integers));

System.out.println("Lowest number in List : " + getMin(integers));

System.out.println("Sum of all numbers : " + getSum(integers));

System.out.println("Average of all numbers : " + getAverage(integers));

System.out.println("Random Numbers: ");

//print ten random numbers

Random random = new Random();

for(int i = 0; i < 10; i++) {

System.out.println(random.nextInt());

}

System.out.println("Using Java 8: ");

System.out.println("List: " +strings);

count = strings.stream().filter(string->string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

count = strings.stream().filter(string -> string.length() == 3).count();

System.out.println("Strings of length 3: " + count);

filtered = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

mergedString = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

squaresList = numbers.stream().map( i ->i\*i).distinct().collect(Collectors.toList());

System.out.println("Squares List: " + squaresList);

System.out.println("List: " +integers);

IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

System.out.println("Random Numbers: ");

random.ints().limit(10).sorted().forEach(System.out::println);

//parallel processing

count = strings.parallelStream().filter(string -> string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

}

private static int getCountEmptyStringUsingJava7(List<String> strings) {

int count = 0;

for(String string: strings) {

if(string.isEmpty()) {

count++;

}

}

return count;

}

private static int getCountLength3UsingJava7(List<String> strings) {

int count = 0;

for(String string: strings) {

if(string.length() == 3) {

count++;

}

}

return count;

}

private static List<String> deleteEmptyStringsUsingJava7(List<String> strings) {

List<String> filteredList = new ArrayList<String>();

for(String string: strings) {

if(!string.isEmpty()) {

filteredList.add(string);

}

}

return filteredList;

}

private static String getMergedStringUsingJava7(List<String> strings, String separator) {

StringBuilder stringBuilder = new StringBuilder();

for(String string: strings) {

if(!string.isEmpty()) {

stringBuilder.append(string);

stringBuilder.append(separator);

}

}

String mergedString = stringBuilder.toString();

return mergedString.substring(0, mergedString.length()-2);

}

private static List<Integer> getSquares(List<Integer> numbers) {

List<Integer> squaresList = new ArrayList<Integer>();

for(Integer number: numbers) {

Integer square = new Integer(number.intValue() \* number.intValue());

if(!squaresList.contains(square)) {

squaresList.add(square);

}

}

return squaresList;

}

private static int getMax(List<Integer> numbers) {

int max = numbers.get(0);

for(int i = 1;i < numbers.size();i++) {

Integer number = numbers.get(i);

if(number.intValue() > max) {

max = number.intValue();

}

}

return max;

}

private static int getMin(List<Integer> numbers) {

int min = numbers.get(0);

for(int i= 1;i < numbers.size();i++) {

Integer number = numbers.get(i);

if(number.intValue() < min) {

min = number.intValue();

}

}

return min;

}

private static int getSum(List numbers) {

int sum = (int)(numbers.get(0));

for(int i = 1;i < numbers.size();i++) {

sum += (int)numbers.get(i);

}

return sum;

}

private static int getAverage(List<Integer> numbers) {

return getSum(numbers) / numbers.size();

}

}

Output:

Using Java 7:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9

Random Numbers:

-1279735475

903418352

-1133928044

-1571118911

628530462

18407523

-881538250

-718932165

270259229

421676854

Using Java 8:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9.444444444444445

Random Numbers:

-1009474951

-551240647

-2484714

181614550

933444268

1227850416

1579250773

1627454872

1683033687

1798939493

Empty Strings: 2

**Performance program :**

|  |
| --- |
| package Java8New; |
|  |  |
|  | import java.util.ArrayList; |
|  | import java.util.Iterator; |
|  | import java.util.List; |
|  | import java.util.ListIterator; |
|  |  |
|  | public class Performance\_Implications { |
|  | public static void main(String[] args) { |
|  |  |
|  | long t1,t2; |
|  | List<Empl> eList = new ArrayList<Empl>(); |
|  | for(int i=0;i<10000;i++){ |
|  | eList.add(new Empl("A",20000)); |
|  | eList.add(new Empl("B",3000)); |
|  | eList.add(new Empl("C",15030)); |
|  | eList.add(new Empl("D",25030)); |
|  | eList.add(new Empl("E",5030)); |
|  | } |
|  |  |
|  | int count = 0; |
|  | /\*\*\*\*\* Here We Are Iterating the List using Universal Iterator & Checking The Result \*\*\*\*\*/ |
|  | t1=System.currentTimeMillis(); |
|  | Iterator<Empl> itr = (Iterator<Empl>) eList.iterator(); |
|  | while(itr.hasNext()){ |
|  | Empl e =itr.next(); |
|  | if(e.getSalary()>15000){ |
|  | count++; |
|  | } |
|  | } |
|  | t2= System.currentTimeMillis(); |
|  | System.out.println("Sequential iterator Time taken: "+(t2-t1)); |
|  |  |
|  | /\*\*\*\*\* Here We Are Creating A 'Sequential Stream' & Displaying The Result \*\*\*\*\*/ |
|  | t1=System.currentTimeMillis(); |
|  | System.out.println("Sequential stream count: "+eList.stream().filter(e->e.getSalary()>15000).count()); |
|  | t2= System.currentTimeMillis(); |
|  | System.out.println("Sequential stream Time taken: "+(t2-t1)); |
|  |  |
|  | /\*\*\*\*\* Here We Are Creating A 'Parallel Stream' & Displaying The Result \*\*\*\*\*/ |
|  | t1=System.currentTimeMillis(); |
|  | System.out.println("Parallel Stream Count: = "+eList.parallelStream().filter(e->e.getSalary()>15000).count()); |
|  | t2 = System.currentTimeMillis(); |
|  | System.out.println("Parallel Stream Time Taken: " + (t2-t1)); |
|  |  |
|  |  |
|  |  |
|  | } |
|  | } |
|  |  |
|  | class Empl{ |
|  | String name; |
|  | int salary; |
|  |  |
|  | public Empl(String name,int salary) { |
|  | this.name=name; |
|  | this.salary=salary; |
|  | } |
|  |  |
|  | public String getName() { |
|  | return name; |
|  | } |
|  | public void setName(String name) { |
|  | this.name = name; |
|  | } |
|  | public int getSalary() { |
|  | return salary; |
|  | } |
|  | public void setSalary(int salary) { |
|  | this.salary = salary; |
|  | } |
|  | @Override |
|  | public String toString() { |
|  | // TODO Auto-generated method stub |
|  | return name+" "+salary; |
|  | } |
|  |  |
|  | } |

package com.mkyong.java8;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class TestJava8 {

public static void main(String[] args) {

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

//Before Java8

List<String> alphaUpper = new ArrayList<>();

for (String s : alpha) {

alphaUpper.add(s.toUpperCase());

}

System.out.println(alpha); //[a, b, c, d]

System.out.println(alphaUpper); //[A, B, C, D]

// Java 8

List<String> collect = alpha.stream().map(String::toUpperCase).collect(Collectors.toList());

System.out.println(collect); //[A, B, C, D]

// Extra, streams apply to any data type.

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

List<Integer> collect1 = num.stream().map(n -> n \* 2).collect(Collectors.toList());

System.out.println(collect1); //[2, 4, 6, 8, 10]

}

}