JAVA 8 Features

**Lambda Expression:**

Generally when we are implementing an Interface in our Class we need to create Object of that interface and then use that method. But by using Lambda Expression we can do it directly

Java lambda expression contains three components

Argument-list: It can be empty or non-empty as well

Arrow broken – It is used to link arguments list and Body of expression

Body – It contains expressions and statements for lambda expression

No Parameter syntax-

()->{

}

One parameter syntax-

(p1}->{

}

Multiple parameter syntax-

(p1,p2}->{

}

Without using lambda Expression-

**public** **interface** Drawable {

**public** **void** draw();

}

**public** **class** lambdaexpample {

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

// **TODO** Auto-generated method stub

**int** width=10;

Drawable d2=**new** Drawable() {

@Override

**public** **void** draw() {

System.***out***.println("Drawing "+width);

}

};

}

}

Using the Lambda Expression –

**public** **interface** Drawable {

**public** **void** draw();

}

**public** **class** lambdaexpample {

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

// **TODO** Auto-generated method stub

**int** width=10;

Drawable d2=()->{

System.***out***.println("Drawing "+width);

};

d2.draw();

}

}

No parameter Lambda example -

**public** **interface** Drawable {

**public** String draw();

}

**public** **class** lambdaexpample {

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

// **TODO** Auto-generated method stub

Drawable d2=()->{

**return** "i have nothing to draw";

};

System.***out***.println(d2.draw());

}

}

One parameter Lambda example -

**public** **interface** Drawable {

**public** String draw(String name);

}

**public** **class** lambdaexpample {

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

// **TODO** Auto-generated method stub

Drawable d2=(name)->{

**return** "Drawing " +name;

};

System.***out***.println(d2.draw("picaso"));

}

}

Two parameter Lambda example-

**public** **interface** Drawable {

**public** String draw(String name1,String name2);

}

**public** **class** lambdaexpample {

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

// **TODO** Auto-generated method stub

Drawable d2=(name1,name2)->{

**return** "Drawing " +name1+ "and " +name2;

};

System.***out***.println(d2.draw("picaso","dog"));

}

}

Lambda expression can be used in collection framework. It provides efficient and concise way to iterate, filter and fetch data

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.stream.Stream;

public class lambdaexpample {

int id;

String name;

int price;

public lambdaexpample(int id, String name, int price) {

this.id=id;

this.name=name;

this.price=price;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

List<lambdaexpample> list=new ArrayList<lambdaexpample>();

list.add(new lambdaexpample(1, "HP laptop", 100));

list.add(new lambdaexpample(2, "Keyboard", 200));

list.add(new lambdaexpample(3, "mouse", 300));

//sorting on basis of name

Collections.sort(list,(p1,p2)->{

return p1.name.compareTo(p2.name);

});

for(lambdaexpample p:list) {

System.out.println(p.id+" "+p.name+" "+p.price);

}

//using lambda to filter data

Stream<lambdaexpample> filter\_data=list.stream().filter(p->p.price>200);

//using lambda to iterate through collection

filter\_data.forEach(

lambdaexpample->System.out.println(lambdaexpample.id+" "+lambdaexpample.name)

);

}

}

**Java Method References:**

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface.

Types:

1. Reference to static method
2. Reference to an instance method
3. Reference to a constructor

Reference to a static method Example:

Syntax:

Containing clas::static method name

**public** **interface** Drawable {

**public** **void** draw();

}

**public** **class** Example1 {

**public** **static** **void** drawSomething() {

System.***out***.println("This is static method");

}

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

//Referring static method

Drawable dw= Example1::*drawSomething*;

//calling interface method

dw.draw();

}

}

Reference to an instance method Example:

Syntax: containing object::instance method name

**public** **interface** Drawable {

**public** **void** draw();

}

**public** **class** Example1 {

**public** **void** drawSomething() {

System.***out***.println("This is non-static method");

}

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

Example1 exp=**new** Example1();

//Referring non-static method using reference

Drawable dw=exp::drawSomething;

//calling interface method

dw.draw();

//Referring non-static method using anonymous object

Drawable dw2=**new** Example1()::drawSomething;

//calling interface method

dw2.draw();

}

}

Reference to a Constructor example:

Class name::new

**public** **interface** Drawable {

Message getMessage(String msg);

}

**public** **class** Message {

**public** Message(String msg) {

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

}

}

**public** **class** Example1 {

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

Drawable dw= Message::**new**;

dw.getMessage("Hello");

}

}

**Functional Interface:**

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

@FunctionalInterface

**public** **interface** Drawable {

//abstract method

**public** **void** draw(String msg);

//It can contain any number of object class methods

String toString();

**boolean** equals(Object obj);

}

**public** **class** Example1 **implements** Drawable {

**public** **void** draw(String msg) {

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

}

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

Example1 exp=**new** Example1();

exp.draw("I am drawing apple");

}

}

**Java 8 Stream:**

Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.

Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

You can use stream to filter, collect, print, and convert from one data structure to other etc.

Example: Filtering collection without using Stream

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> list=**new** ArrayList<lambdaexpample>();

list.add(**new** lambdaexpample(1, "HP laptop", 100));

list.add(**new** lambdaexpample(2, "Keyboard", 200));

list.add(**new** lambdaexpample(3, "mouse", 300));

List<Integer> productpricelist=**new** ArrayList<Integer>();

**for**(lambdaexpample lb:list) {

//filtering data of list

**if**(lb.price<300) {

//adding price to productpricelist

productpricelist.add(lb.price);

}

}

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

}

}

Example: Filtering collection by using Stream

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

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

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

List<Integer> productpricelist=(List<Integer>) prdlist.stream()

.filter(p->p.price<200) //filtering data

.map(p->p.price) //fetching price

.collect(Collectors.*toList*()); //collecting as List

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

}

}

Example: Stream iterating

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

**import** java.util.stream.Stream;

**public** **class** Example1 {

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

Stream.*iterate*(1, i->i+1)

.filter(i->i%5==0)

.limit(5)

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

}

}

Example: Stream example for Filtering and Iteration

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

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

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

prdlist.stream()

.filter(p->p.price==200)

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

}

}

Example: reduce() method in stream

This method takes a sequence of input elements and combines them into a single summary result by repeated operation. For example, finding the sum of numbers, or accumulating elements into a list.

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

//compact approach for filtering data

**int** toalprice=prdlist.stream()

.map(p->p.price)

.reduce(0,(sum,price)->sum+price);

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

}

}

Example: Below is to count how many are less than 200 price

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

//compact approach for filtering data

**long** numprice=prdlist.stream()

.filter(p->p.price<=200)

.count();

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

}

}

Java Default methods:

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Example:

**public** **interface** Drawable {

//default method

**default** **void** draw() {

System.***out***.println("this is default method");

}

//abstract method

**void** drawMore(String msg);

//static abstract method

**static** **void** drawMore2(String msg) {

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

}

}

**public** **class** Example1 **implements** Drawable {

**public** **void** drawMore(String msg) {

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

}

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

Example1 exp=**new** Example1();

exp.draw();

exp.drawMore("I am drawing in abstract method");

Drawable.*drawMore2*("drawing in static method");

}

}

Java forEach loop:

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

Example:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Example1{

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

List<String> lst=**new** ArrayList<String>();

lst.add("cricket");

lst.add("football");

lst.add("tennis");

lst.add("hockey");

lst.forEach(i->System.***out***.println(i));

}

}

Example:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Example1{

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

List<String> lst=**new** ArrayList<String>();

lst.add("cricket");

lst.add("football");

lst.add("tennis");

lst.add("hockey");

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

}

}

Example: Stream forEachOrdered() example

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Example1{

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

List<String> lst=**new** ArrayList<String>();

lst.add("cricket");

lst.add("football");

lst.add("tennis");

lst.add("hockey");

//iterating by lambda expression

lst.stream().forEachOrdered(i->System.***out***.println(i));

//iterating by passing method interface

lst.stream().forEachOrdered(System.***out***::println);

}

}

**Java 8 Collectors:**

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Example:

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.Set;

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

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

//fetching price and collecting to as List

List<Integer> pricelist=prdlist.stream()

.map(p->p.price)

.collect(Collectors.*toList*());

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

//fetching price and collecting to as Set

Set<Integer> pricelist2=prdlist.stream()

.map(p->p.price)

.collect(Collectors.*toSet*());

System.***out***.println(pricelist2);

}

}

Example: below where collectors used for summing, average and count of number of items

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.Set;

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

**import** java.util.stream.Stream;

**public** **class** lambdaexpample {

**int** id;

String name;

**int** price;

**public** lambdaexpample(**int** id, String name, **int** price) {

**this**.id=id;

**this**.name=name;

**this**.price=price;

}

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

// **TODO** Auto-generated method stub

List<lambdaexpample> prdlist=**new** ArrayList<lambdaexpample>();

prdlist.add(**new** lambdaexpample(1, "HP laptop", 100));

prdlist.add(**new** lambdaexpample(2, "Keyboard", 200));

prdlist.add(**new** lambdaexpample(3, "mouse", 300));

//summing of prices

**int** Sumprice=prdlist.stream()

.collect(Collectors.*summingInt*(p->p.price));

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

//average of prices

Double avgprice=prdlist.stream()

.collect(Collectors.*averagingInt*(p->p.price));

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

//count of prices

**long** countprice=prdlist.stream()

.collect(Collectors.*counting*());

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

}

}

StringJoiner Java 8:

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

Example:

**import** java.util.StringJoiner;

**public** **class** Example1{

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

//delimiter as comma

StringJoiner sj=**new** StringJoiner(",");

sj.add("ramu");

sj.add("raju");

sj.add("ravi");

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

}

}

Example: Merging two String joiners using new String joiner

**import** java.util.StringJoiner;

**public** **class** Example1{

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

//delimiter as comma

StringJoiner sj=**new** StringJoiner(",");

sj.add("ramu");

sj.add("raju");

sj.add("ravi");

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

StringJoiner sj2=**new** StringJoiner(":");

sj2.add("pradip");

sj2.add("abhi");

sj2.add("nishnat");

System.***out***.println(sj2);

StringJoiner sj3=sj.merge(sj2);

System.***out***.println(sj3);

}

}

**Java Option Class:**

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Example: Java program without using Optional

The below program terminates abnormally with Null pointer exception

**import** java.util.StringJoiner;

**public** **class** Example1{

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

String[] str = **new** String[10];

//since we are not passing any value to string and asking to lowecase it will throw error since there is nothing

String lowercaseString = str[5].toLowerCase();

System.***out***.print(lowercaseString);

}

}

Example: Now using Optional class

**import** java.util.Optional;

**import** java.util.StringJoiner;

**public** **class** Example1{

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

String[] str = **new** String[10];

//since we are not passing any value to string and asking to lowercase it will throw error since there is nothing

//now optional will handle it

Optional<String> op=Optional.*ofNullable*(str[5]);

//it will check whether any String or value present

**if**(op.isPresent()) {

String lowercaseString = str[5].toLowerCase();

System.***out***.print(lowercaseString);

}**else**

System.***out***.println("string value is not present");

}

}

Java 8 Parallel Soting:

An additional feature in Array class which is used to sort array elements parallel

Example:

**import** java.util.Arrays;

**import** java.util.Optional;

**import** java.util.StringJoiner;

**public** **class** Example1{

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

// Creating an integer array

**int**[] arr = {5,8,1,0,6,9};

// Iterating array elements

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

System.***out***.print(i+" ");

}

// Sorting array elements parallel

Arrays.*parallelSort*(arr);

System.***out***.println("\nArray elements after sorting");

// Iterating array elements

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

System.***out***.print(i+" ");

}

}

}

Java Type Inference:

Type inference is a feature of Java which provides ability to compiler to look at each method invocation and corresponding declaration to determine the type of arguments.

Java provides improved version of type inference in Java 8. the following example explains, how we can use type inference in our code:

Here, we are creating arraylist by mentioning integer type explicitly at both side. The following approach is used earlier versions of Java.

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

In the following declaration, we are mentioning type of arraylist at one side. This approach was introduce in Java 7. Here, you can left second side as blank diamond and compiler will infer type of it by type of reference variable.

Example:

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Example1{

**public** **static** **void** showList(List<Integer>list){

**if**(!list.isEmpty()){

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

}**else** System.***out***.println("list is empty");

}

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

// An old approach(prior to Java 7) to create a list

List<Integer> list1 = **new** ArrayList<Integer>();

list1.add(11);

*showList*(list1);

// Java 7

List<Integer> list2 = **new** ArrayList<>(); // You can left it blank, compiler can infer type

list2.add(12);

*showList*(list2);

// Compiler infers type of ArrayList, in Java 8

*showList*(**new** ArrayList<>());

}

}

**Method Parameter Reflection:**

Java provides a new feature in which you can get the names of formal parameters of any method or constructor. The java.lang.reflect package contains all the required classes like Method and Parameter to work with parameter reflection.

Example:

**public** **class** Calculate {

**int** add(**int** a, **int** b){

**return** (a+b);

}

**int** mul(**int** a, **int** b){

**return** (b\*a);

}

}

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Parameter;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Example1{

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

// Creating object of a class

Calculate cal = **new** Calculate();

// instantiating Class class

Class cls = cal.getClass();

// Getting declared methods inside the Calculate class

Method[] method = cls.getDeclaredMethods(); // It returns array of methods

// Iterating method array

**for** (Method method2 : method) {

System.***out***.println(method2.getName()); // getting name of method

// Getting parameters of each method

Parameter parameter[] = method2.getParameters(); // It returns array of parameters

// Iterating parameter array

**for** (Parameter parameter2 : parameter) {

System.***out***.println(""+parameter2.getParameterizedType()); // returns type of parameter

System.***out***.println(""+parameter2.getName()); // returns parameter name

}

}

}

}

Java Type Annotations:

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type.

For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this

@NonNull String str;

Below are some other annotations

@NonNull List<String>

List<@NonNull String> str

Arrays<@NonNegative Integer> sort

@Encrypted File file

@Open Connection connection

**Java Repeating Anotations:**

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

1. Declare the repeatable annotation type
2. Declare the containing annotation type

1 .Declare the annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

**import** java.lang.annotation.Repeatable;

@Repeatable(Students.**class**)

**public** **@interface** Student {

String name();

String address();

}

2. Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** Students {

Student[] value();

}

**public** **class** Example1{

@Student(name="raju", address="pune")

@Student(name="ravi", address="mumbai")

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

System.***out***.println("I cam here");

Student[] st=Example1.**class**.getAnnotationsByType(Student.**class**);

**for**(Student st1:st) {

System.***out***.println("I cam here");

System.***out***.println(st1.name());

System.***out***.println(st1.address());

}

}

}

Before Java 8 there is no retention so we had to like below

**import** java.lang.annotation.Repeatable;

**public** **@interface** Student {

String name();

String address();

}

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** Students {

Student[] value();

}

**public** **class** Example1{

@Students({

@Student(name="raju", address="pune"),

@Student(name="ravi", address="mumbai")

})

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

System.***out***.println("I cam here");

Student[] st=Example1.**class**.getAnnotationsByType(Student.**class**);

**for**(Student st1:st) {

System.***out***.println("I cam here");

System.***out***.println(st1.name());

System.***out***.println(st1.address());

}

}

}