**Java 8 inclusion**

References : <http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

1. What is functional interface?

Ans: the interface which has only one abstract method. It can have more than one static and default method

1. Example of functional interface

Ans : Runnable which has only one abstract method Runnable.

There are some generic functional interfaces like

Comparator<T> method int compare(T o1, T o2);

Function<T,R> method R apply(T t)

Consumer<T> void accept(T t)

Predicate<T> Boolean test(T t)

1. Why java 8 introduced default and static method when we have already abstract class?

Ans :

* If we extends an abstact class we cannot extend any other class, but for interface we can implement multiple interfaces from a single class.
* The default methods were introduced to provide backward compatibility so that existing intefaces can use the lambda expressions without implementing the methods in the implementation class.
* Static methods are introduced if we need any utility method for an interface

1. Can a default method is static?

Ans: Non abstract method of interface can be either static or default. Default and static for same method is not applicable

1. How to access default and static methods?

**interface** Interviewer {

**static** **void** bookConferenceNumber(){

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

}

**default** **void** defultMethod(){

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

}

**void** normalMethod();

}

**class** Manager **implements** Interviewer{

@Override

**public** **void** normalMethod() {

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

}

@Override

**public** **void** defultMethod(){

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

}

}

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

Interviewer inv = **new** Manager();

inv.defultMethod(); // print child method

Interviewer.*bookConferenceNumber*();

inv.normalMethod();

}

Static method cannot be accessed by the interface reference unlike normal static methods.

To access the default method of the interface we need to initialize the Interviewer reference.

If we don’t override the method in Manager class then it will print the parent method.

Interviewer inv;

inv. defultMethod(); // nullpointer exception as inv is not initialized

We don’t need to initialize any object for accessing static method

Interviewer.*bookConferenceNumber*();

1. Can we override the static method in implementing class or sub interface?

Ans: Like other static methods static method override is not possible.

If we write

Class Manager implements Interviewer {

**static** **void** bookConferenceNumber(){

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

}

}

Then it is just a another static method of Manager class.

Suppose if we declare the bookConferenceNumber() of Interviewer interface as final

Then we cannot write bookConferenceNumber() method in Manager class. Its look like Manager class is trying to override the static method but due to final keyword not able to do that. Actually what happened , if a method with the same signature and with same instance and final is registered in compiler, compiler doesn’t allow any other class or interface which implements or extends the same interface to declare the same method

1. Rules to call default method ?

Ans :

* Default method cannot be called from a static method of an interface
* Default method cannot be called from a static method of the class which implements the interface.
* Default method can be called from another default method of the interface or the normal method of a class which implements the interface without using any object reference. We can use this , but not required.
* We cannot call a default method from a static method of any class directly. But if we create a reference variable inside a static method then we can access

**interface** Interviewer {

**default void defultMethod(){**

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

**}**

**}**

**class** Manager **implements** Interviewer{

**public** **static** **void** staticClassMethod() {

**Interviewer interviewer = new** Manager();

**Manager manager = new Manager();**

**interviewer. defultMethod();**

**manager. defultMethod();**

**}**

**}**

1. Rules to call static method ?

Ans :

* A static method can be called without any refrence only form the same interface, not from outside.
* We need to call static method using class reference from outside the interface.

Interviewer.staticMethod();

1. **interface** BaseInt1{

**int** ***a***=10;

**}**

**interface** BaseInt2{

**int** ***a***=20;

**}**

**interface** MyInt **extends** BaseInt1,BaseInt2 {

default void method(){

System.out.println(a); // ambiguity error

}

**}**

How to solve this problem?

Ans : This line can be written as

System.out.println(BaseInt1.a); OR System.out.println(BaseInt2.a);

OR

Declare one more variable with the same name a in MyInt

Because variables are not overridden , so it will be treated as a new variable in MyInt interface

Like

**interface** MyInt **extends** BaseInt1,BaseInt2 {

**int a =30;**

default void method(){

System.out.println(a);

}

**}**

Output 30

1. If BaseInt1 and BaseInt2 has same default method with same signature and same return type ?

Ans : interface MyInt or the class which extends both the interface BaseInt1 and BaseInt2

Has to override the default method to resolve ambiguity problem

1. If BaseInt1 and BaseInt2 has same default method with same signature and different return type ?

Ans: No chance of solving ambiguity problem. Even override cannot solve. Compiler error



**interface** BaseInt1{

**default** **void** method(){

System.***out***.println("method of BaseInt1");

}

}

i**nterface** BaseInt2{

**default** **void** method(){

System.***out***.println("method of BaseInt2");

}

}

interface MyInt{

**Static**  **void** statMethod(){

System.***out***.println("static method of BaseInt1");

}

default void method(){

System.out.println(a); // ambiguity error

}

}

Class TestClass implements MyInt{

Public void method2(){

Line1: method(); //

Line2 : BaseInt1 .super. method();

Line3 : MyInt.this. method();

Line4 : MyInt.this. statMethod();

}

}

Ans :

Line 1 : call method() of MyInt interface , it will call most recent level method

Line2 : compiler error. This line can only be written if TestClass implements BaseInt1 directly or

From MyInt interface as it extends BaseInt1 interface, super works for immediate parent

Line3 : Compiler error, as this keyword can only be used from inside MyInt interface , not from outside

Line4 : cannot call any static method from this or super instance

1. Default and static method without any body is possible ?

Ans : No default or static means it is not abstract method so we have to declare the body otherwise compiler error

1. Abstract default or static method is possible ?

Abstract default method is not possible

Before java 8 static method was not possible , now after java 8 static method is possible and abstract is also possible but abstract static is not possible.

1. Example of simple labda expression

( type1 var1, type2 var2, ……. typen varn) -> { expression1 ; expression2; ……. return val; };

**class** Emp{

**private** String name;

**private** **long** salary;

**private** **int** rating;

// getter setter and constructor

**}**

@FunctionalInterface

**interface** Validate{

**boolean** check(Emp e,**int** chekingParam);

}

Validate validate = (emp,rating)->{**return** (emp.getRating() >=rating);};

Emp e = **new** Emp("X", 30000, 3);

**if**(validate.check(e,3)) {

System.***out***.println(e.getName());

}

Output : X;

Lambda expression always returns the instance of the anonymous class which implements the functional interface here in this example Validate

1. What is effective final variable in java 8 ?

Ans : A non-final local variable or method parameter whose value is never changed after initialization is known as effectively final. It's very useful in the context of the lambda expression. If you remember, prior to Java 8, we cannot use a non-final local variable in an anonymous class. The variable should be declared as final. Now in java 8 we don’t need to declare the variable as final , but we need to make sure that the variable should not be re assigned once again. If we modify the code and do not use the local variable inside anonymous class, it will remain as normal variable.

class Demo {

public static void main(String args[]){

int a = 2; ----------------------------------------------- 1

a =3;

MyClass myClass = new MyClass(){

public void method(){

System.out.println(a);

}

};

a =3; ---------------------------------------------------------- 2

}

}

class MyClass{

public void method(){}

}

A is a normal variable, but it is used inside the anonymous class , so we cannot modify the variable before (line 1) and also after(line 2 ) the anonymous class. Prior to java 8 we need to declare a as a final. But in java 8 it is not required.

Lambda expression is nothing but an anonymous class for the functional interface.

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

new Thread(() -> {

System.out.println("i = " + i); // Does not compile!

}).start();

}

It will throw an error as I is keep changing.

1. Example of mapper in String list and user defined list

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

List<String> myList = Arrays.*asList*("a1", "a2", "b1", "c2", "c1");

Employee e1 = **new** Employee(1, 10000, "A");

Employee e2 = **new** Employee(2, 20000, "B");

Employee e3 = **new** Employee(3, 30000, "C");

Employee e4 = **new** Employee(4, 40000, "D");

List<Employee> empList = **new** ArrayList<Employee>();

empList.add(e1);

empList.add(e2);

empList.add(e3);

empList.add(e4);

/\*\* string mapper \*/

Stream<String> stringStream = myList.stream().filter((x)->x.equals("c1"));

Stream<Employee> empStream = empList.stream().filter((e)->e.getId()==1);

Function<String,Integer> mapperString = String :: hashCode;

**int** hashCode = stringStream.map(mapperString).findAny().orElse(0);

String value = stringStream.map(String :: toUpperCase).findAny().orElse("");

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

String name = empStream.map(mapperEmp).findAny().orElse("default");

String name2 = empStream.map(Employee :: getName).findAny().orElse("default");

}

Output :

hashCode = hashcode for “c1”

value = IllegalStateException, saying “stream is closed”.

We can use any of the two execution from one stream instance.

name = “A”

name2 = IllegalStateException, saying “stream is closed”.

We can use any of the two execution from one stream instance.

1. Can we solve these IllegalStateException ?

Ans : We can use Supplier<Stream<T>> in place of Stream<T>

Supplier<Stream<String>> streamSupplier = () -> myList.stream();

**int** hashCode = streamSupplier.get().map(mapperString).findAny().orElse(0);

String value = streamSupplier.get().map(String :: toUpperCase).findAny().orElse("");

streamSupplier.get() always return a new stream instance, so no exception will occur

1. Inside map() method Function<T,R> instance used , how is this “ :: ” working

Before the :: the instance type and after the :: method name

1. What is Stream object ?

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example  **filter, map and sorted** are intermediate operations whereas **forEach** is a terminal operation as its return type is void.

Such a chain of stream operations as seen in the example above is also known as *operation pipeline.*

1. How many way you can create a stream object?

**From arrays**

**i.>**

String[] arr = { "program", "creek", "program", "creek", "java", "web",

"program" };

stream = Stream.of(arr);

ii>

stream = Stream.of("program", "creek", "program", "creek", "java",

"web", "program");

iii>

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

Stream<String> stream = Arrays.stream(stringArr);

iv>

From above example for user defined objects

Stream<Employee> empStream = Stream.*of*(e1,e2,e3,e4);

**From collections**

stream = list.stream();

**Stream.generate**

Stream<String> stream = Stream.*generate*(() -> "test").limit(10);

String[] strArr = stream.toArray(String[]::**new**);

System.***out***.println(Arrays.*toString*(strArr));

Output : test test test…… 10 times

**Stream.iterate**

Stream<BigInteger> bigIntStream = Stream.*iterate*(BigInteger.***ZERO***, n -> n.add(BigInteger.***ONE***)).limit(10);

BigInteger[] bigIntArr = bigIntStream.toArray(BigInteger[]::**new**);

System.***out***.println(Arrays.*toString*(bigIntArr));

Output : 0 1 2 3 … …. 9

1. How forEach(Consumer<T> consumer) works

Stream<Employee> streamEmp = empList.stream().filter((e)->e.getName().startsWith("A"));

Stream<String> streamString = myList.stream().filter((x)->x.endsWith("1"));

Consumer<String> stringConsumer = (x)->System.***out***.println(x);

Consumer<String> stringConsumer2 = System.***out*** :: println;

Consumer<Employee> empConsumer = (emp)->System.***out***.println(emp.getId());

Consumer<Employee> empConsumer2 = Employee :: printName;

streamString.forEach(stringConsumer);

streamEmp.forEach(empConsumer2);

Output :

a1

b1

c1

A

Abc

void printName() is a new method introduced in Employee class to print employee name

1. Can we do streaming for primitive types?

Ans : Yes.

Besides regular object streams Java 8 ships with special kinds of streams for working with the primitive data types int, long and double. As you might have guessed it's IntStream, LongStream and DoubleStream

IntStream.range(1, 4) .forEach(System.out::println);

Output:

1

2

3

All those primitive streams work just like regular object streams with the following differences: Primitive streams use specialized lambda expressions, e.g. **IntFunction** instead of **Function** or **IntPredicate** instead of Predicate. And primitive streams support the additional terminal aggregate operations **sum()** and **average()**:

Arrays.stream(new int[] {1, 2, 3})

.map(n -> 2 \* n + 1)

.average()

.ifPresent(System.out::println); // 5.0

1. How to convert a regular object stream to a primitive stream or vice versa. ?

For that purpose object streams support the special mapping operations mapToInt(), mapToLong() and mapToDouble:

Stream.of("a1", "a2", "a3")

.map(s -> s.substring(1))

.mapToInt(Integer::parseInt)

.max()

.ifPresent(System.out::println); // 3

Primitive streams can be transformed to object streams via mapToObj():

IntStream.range(1, 4)

.mapToObj(i -> "a" + i)

.forEach(System.out::println);

// a1

// a2

// a3

Here's a combined example: the stream of doubles is first mapped to an int stream and than mapped to an object stream of strings:

Stream.of(1.0, 2.0, 3.0)

.mapToInt(Double::intValue)

.mapToObj(i -> "a" + i)

.forEach(System.out::println);

// a1

// a2

// a3

1. What is collect ?

Ans: Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map

Example

List<Emp> result = empList.stream().filter(e-> e.getRating()>4).collect(Collectors.*toList*());

1. Lets build a table for mostly used functional methods methods of Stream<T>

|  |  |  |  |
| --- | --- | --- | --- |
| Method name | Input argument | Functional interface name | Functional method |
| forEach : void | Consumer<? **super** T> action | Consumer<T> | **void** accept(T t); |
| map : Stream | Function<? **super** T, ? **extends** R> mapper | Function<T,R> | R apply(T t); |
| filter : Stream | Predicate<? **super** T> predicate | Predicate<T> | **boolean** test(T t); |
| mapToInt : IntStream | ToIntFunction<? **super** T> mapper | ToIntFunction<T> | **int** applyAsInt(T value); |
| Sorted : Stream | Comparator<? **super** T> comparator | Comparator<T> | **int** compare(T o1, T o2); |
| Collect : <R, A> R | Collector<? **super** T, A, R> collector | Collector<T, A, R> (this is not a functional interface) |  |

1. Sort a map by keys

Ans :

Map<Integer,String> sortedMapWithKey = map1.entrySet().stream().sorted(Map.Entry.*comparingByKey*()).collect(Collectors.*toMap*(Map.Entry::getKey, Map.Entry::getValue,

(oldValue, newValue) -> oldValue, LinkedHashMap::**new**));

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

1. Sort a map by values ?

Ans :

Map<String, Integer> sortedmapWithVal = map2.entrySet().stream()

.sorted(Map.Entry.*comparingByValue*())

.collect(Collectors.*toMap*(Map.Entry::getKey, Map.Entry::getValue,

(oldValue, newValue) -> oldValue, LinkedHashMap::**new**));

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