­­­­­­­­Java 8 new features

1. Lambda Expressions
2. Functional Interface
3. Default methods and static methods
4. Predefined Functional Interfaces

Predicate

Function

Consumer

Supplier

Etc

1. Double colon Operatory (::)

Method Reference

Constructor reference

1. Streams
2. Date and Time API
3. Optional class
4. Nashron JavaScript Engine etc

Purpose of Java 8

1. To simplify programming
2. To Utilize functional programming benefits in java
3. To enable parallel programming

**Lambda Expression**

It is an anonymous function without name, without return type, without modifier.

LISP is the first programming that use Lambda expression

Purpose of lambda expression

1. To enable functional programming in java
2. Write more readable maintainable and concise code
3. To use APIs very easily and effectively
4. To enable parallel processing

Note: Without curly braces we can not use return keyword. Compiler will consider returned value automatically. Within curly braces if we want to return some value compulsory we should use return statement.

**Anonymous Inner Class:**

Anonymous inner class can extends a normal class

Anonymous inner class can extends abstract class

Anonymous inner class can implement an interface which contain any number of abstract class.

|  |  |
| --- | --- |
| **Anonymous Inner Class** | **Lambda Expression** |
| It is class without name | It is function with out name(anonymous function) |
| It can extends abstract class and inner class | It can not extend abstract and concrete classes |
| It can implements an interface that contains any number of abstract methods | It can implement an interface with contain single abstract method(Functional Interface) |
| Inside anonymous inner class we can declare instance variable | Inside lambda expression we can’t declare instance variable what ever variables declare are considered as local variable |
| Anonymous inner class can be instantiated | Lambda expressions can not be instantiated |
| Inside anonymous inner class this always refers current anonymous inner class object but not outer class object | Inside lambda expression this always refers current outer class object i.e enclosing class object |
| Anonymous inner class is best choice if we want to handle multiple methods | Lambda expression is the best choice if we want to handle interface with single abstract method (Functional interface) |
| For the anonymous inner class, at the time of compilation, a separate class file will be generated | For the lambda express, at the time of compilation no separate .class file will be generated |
| Memory will be allocated on demand whenever we creating object | Lambda expression will reside in permanent memory of sum(method Area) |

**Functional Interface:**

Functional interface is an interface which contains single abstract method(SAM) ie.

Runnable =====> run()

Collable =====> call()

Comparable ===> compareTo()

ActionListener ===> actionPerformed()

It can contain any number of default method and static method but rule is should contain only one abstract method.

**Default Methods:**

Until 1.7 version every method present inside interface is always: public and abstract

Void m1();

public void m1();

abstract void m1();

public abstract void m1();

Related Methods:

1.8 V:default method +static methods

1.9 V: private methods

Variables:

Public static final

**Default Method:**

Without effecting implementation classes if we want to add new method to the interface == default method.

Interface interf{

default void m1(){

System.out.println(“Default method”);

}

Class Test implements Interf

{

Public void m1()

{

System.out.println(“overriding version of default method”)

}

Public static void main(String args[])

{

Test t=new Test();

t.m1();

}

}

}

Object class methods are not allowed to take as default method.

**Predefined Functional Interfaces:**

**Predicate:** Take some input and perform some conditional check and return Boolean value.

**Function:** Take some input and perform some operation and return the result which is need not be Boolean type.

**Consumer:** Accept some input and perform required operation and not required to return anything.

Interface Consumer<T>{

Public void accept();

}

**Supplier:** Just supply my required objects and it won’t take any input

Interface Supplier<R>

{

Public R get();

}

Here R is return type.it always return a value.it never take any input

**Two argument Predefined functional interfaces**

**BiPredicate:**

Normal Predicate can take only one input argument and perform some conditional check. Sometimes our programming requirement is we have to take 2 input arguments and perform some condtional check, for this requirement we should go for BiPredicate.

BiPredicate is exactly same as Predicate except that it will take 2 input arguments.

Interface BiPredicate<T1,T2>

{

Public Boolean test(T1 t1, T2 t2);

//remaining default methos: .and(), .or(), .negate()

}

**BiFunction:**

Interface BiFunction<T,U,R>

{

Public R apply(T t, U u);

// default method andThen()

}

BiConsumer

**Primitive Functional interfaces**

IntPredicate

IntFunction

IntConsumer

**Primitive Predicate:**

IntPredicate

DoublePredicate

LongPredicate

**Note:** if predicate needs to take 2 argument then we have to go for normal Bipredicate.

**Primitive Function Type:**

**DoubleFunction:c**an take input type as double return type can be any type.

I**ntFunction:** can take int type as input

**LongFunction:** can take long type as input

**DoubleToIntFunction:** input type: double

Return type: int

Method: int applyAsInt(double value)

**DoubleToLongFunction:** input type: double

Return type :long

applyAsLong(double)

**IntToDoubleFunction** -> applyAsDouble()

**IntToLongFunction** -> applyAsLong()

**LongToIntFunction** -> applyAsInt()

**LongToDoubleFunction** --> applyAsDouble()

ToIntFunction -> return type is int

Input can be any thing

Int applyasInt()

ToLongFunction

ToDoubleFunction

ToIntBiFunction-> return must be int type but inputs can be anytype

Int applyAsInt(T t,U u)

ToLongBiFunction

Long applyAsLong(T t,U u)

Applies this function to the given arguments.

ToDoubleBiFunction:

Double applyAsDouble(T t,U u)

Applies this function to the given arguments.

**Primitive type consumer:**

IntConsumer:

Void accept(int value)

Performs this operation on the given argument.

LongConsumer

DoubleConsumer

**ObjDoubleConsumer<T>**

void accept(T t,double value)

ObjIntConsumer<T>

ObjLongConsumer<T>

**PRIMITIVE TYPE OF SUPPLIER**

BooleanSupplier

Boolean getAsBoolean()

IntSupplier

Int getAsInt()

LongSupplier

Long getAsLong()

DoubleSupplier

Double getAsDouble()

**THE PRIMITIVE TYPE OF UNARY OPERATOR:**

UnaryOperator<T>:

If input and out are always same type then we should go for Unary operator

It is child of Function<T,T>

Primitive type of BinaryOperator:

IntBinaryOperator

LongBinaryOperator

DoubleBinaryOperator

**Method and Constructor Reference:**

Classname::method name

Object reference :: method name

In the case of method reference different return types are allowed.

**Collections and Stream:**

If we want to represent a group of object as a single entity then we should go for collection.

If we want to process objects from collection then we should go for stream.

Filter:

Input :10 element

Output 0 to 10 or <=10

Map:

Input: 10 element

Output: 10

Methods:

Stream()

Filter()

Map()

Collect()

Count()

Sorted()

Sorted(Comparator)

Min()

Max()

Stream.forEach(Function)

List.stream.forEach(System.out::Println)

toArray()

forEach()

Stream.of()

­­