*Java 8*

*Lambda expression*

*Function Interface*

*Default methods in interfaces*

*Static methods*

*Function*

*Method Reference and construct reference “(: : )" double coloun operator*

*Stream API*

*Date and Time API*

*Supplier<R> (Predifined functioanal interface)*

*Consumer*

*Java script (Nashorn)*

*To simplify Programming*

*To utilise function Programming/Processing.*

*Lamda expressions*

*To enable function Programming in java*

*Write more readable, maintainable and concise calls.*

*To use API’s very easily and effectively*

*To enable Parallel Processing*

*Landa expression is anonymous function (name less function)*

*Anonyms functions are also called* ***clauses***

*No name*

*No return type*

*No modifres*

*Added symbol ->*

|  |  |
| --- | --- |
| *Pblic void m(){*  *s.o.p();*  *}* | *() -> {S.O.P(‘’);}* |
| *Public void add(int a, int b) {*  *s.o.p(“a+b”);*  *}* | *(int a, int b ) -> s.o.p(“a”);*  *// based on context compiler will guess the type of the arguments.* |
| *Public int squre(int a) {*  *Return n\*n;*  *}* | *n-> n\*n;*  *//if passing one argument parenthesis is optional*  *// if one statement we can remove {}* |

*Conclusion*

*Lambda can have any number of arguments*

*For one argument, lambda expressions parenthesis are optional.*

*Based on context compiler can guss type automatically*

*{}/Curly braces are option for one statement*

***How to invoke LAMBDA expressions***

***Example :*** *lambdaExpressions1.java*

**interface** FuncInterface

{

// An abstract function

**void** abstractFun(**int** x);

// A non-abstract (or default) function

**default** **void** normalFun()

{

System.out.println("Hello");

}

}

**class** Test

{

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

{

// lambda expression to implement above

// functional interface. This interface

// by default implements abstractFun()

FuncInterface fobj = (**int** x)->System.out.println(2\*x);

// This calls above lambda expression and prints 10.

fobj.abstractFun(5);

}

}

***Functional Interace***

*If an interface having only one abstract methods, then it is functional interface.*

*Some interfaces*

*Java.lang.Runnable interface – method is run() functional interface*

*Java.util.concurrent.Callable.Callable interface - method is call() functional interface*

Single abstract method in an functional interface.

Can contain any number of default and static methods in an functional interface.

***@FunctionalInterface***

To indicate explicitly an interface as functional interface we use @functional Interface annotations.

@FunctioanlInterface

Interface interface{

Public abstract m1();

Static void m3(); //abstract

}

@FunctionInterface

Interface a { } ------> invalid functional interface should have only one single abstract method.

@FunctionalInterface

Interface B

{

Public void m1(); -----> valid interface

}

@FunctionalInterface

Interface A extends B { } valid if child interface does not contain any abstract method.

Valid functions interface

@FunctionInterface

Interface A {

Public abstract void m1();

}

**@FunctionalInterface**

Interface B extends A { public void m1(); // method is overridden method.

Invalid Interface

@FunctioanlInterface

Interface A {

Public void m1();

}

@FunctionalInterface

Interface B extends A {

Public void m2();

}

@FunctionalInterface

Interface int {

Public void m1();

}

Class Test {

Public static void main(String args[]) {

Interface f= ()-> {s.op();}

f.m1();

}

}

@FunctionalInterface

Interface f {

Public void add(int a, int b)}

Class Test {

Interface f = (a,b) -> s.o.p(a+b);

f.add(2,3);

Based on context argument type is validated

Interface f = n -> n\*n;

Whenever we are taking curly braces return statement compulsory and a semicolon after curly braces.

Interface i = n-> {n\*n;};

n->n\*n;

N -> return n\*n // invalid because without {} we can’t take a return statement

N -> {return n\*n;};

N-{n\*n}; //invalid with in curly brace return statement be required

}

**Date and Time api’s (JODA API)**

**Date**

**Time**

**Calander**

Developed by Joda.org

Java.time.LocalDate

LocalDate date = LocalDate.now();

S.O.P(date);

LocalTime time = LocalTime.now();

Sop(time);

LocalDateTime dateTime = LocalDateTime.now();

LocalDateTime can represent current or required data object

Period ( quantity of time )

LocalDate birthdate = LocalDate.of(1976,12,14);

Today = LocalDate.now();

Perod p = Peroid.between(birthday,today);

Sop()

Year()

Scanner sc= new Scanner(System.in);

Sop(Enter number);

N = sc.nextInt();

Year y = Year.of(n);

ZoneId;

ZoneId zone = ZoneId.systemDefault();

ZoneI.of(Ameria/Lo angler);

ZoneDateTime time = ZoneDateTime.now();

Default Method

Without effecting implementation calls we can extend functionality.

StaticMehod

Main() in interface

interface{

Public static void main(String args[])

}

***Predicate***

Java.util.function.Predicate (Interface) -- [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

test(T)

***Function*** (Interface) -- [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

Java.util.funcation.Funcation

apply(T)

**Consumer** (Interface) -- [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

Java.util.function.consumer. accept(T)

Supplier<T> (Interface) -- [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)

Java.util.Function.Supplier. get()

**:: Operator**

**To implement method reference or construct reference**.

Reference to a static method className::methodName

Reference to an instance method. Object::methodName

Reference to a constructor. className::new

**PremGen space situation**

This memory space is completely removed.

The PremSize and MaxPermSize JVM arguments are ignored and a warning is issued if present at start-up

**Metaspace Memory allocation model**

Most allocations for the class metadata are now allocated out of native memory.

The Classes that were used to describe class metadata have been removed.

**Type Annotations**

Type Annotations are annotations that can be placed anywhere you use a type. This includes the new operator, type casts, and implements clauses and throws clauses.

Simple type annotations are

@NotNull String str1 = … @Email String str2 = …. ;

Type annotations can also applied to nested types

[Map.@NonNull](mailto:Map.@NonNull) Empty = …;

Constructors with type annotations:

New @Interned Myobject()

New @NonEmpty @Readonly List<String>(myNonEmptyStringSet)

Octal Numbers:

8 10(1+0)

9 11(1+1)

**Hexadecimal Literals:**

|  |  |
| --- | --- |
| 9 | 9 |
| 10 | A |
| 11 | B |
| 12 | C |
| 13 | D |
| 14 | E |
| 15 | F |
| 16 | 10 (1+0) |
| 17 | 11 (1+1) |

Thread

join() method of java.lang.Thread class is used to maintain the order of execution of threads.

ArrayDeques :

**The ArrayDeque class**in Javais introduced from JDK 1.6. It is an implementation of **Deque Interface** which allows insertion of elements at both the ends. It does not have any restrictions on capacity. It expands automatically as we add more elements. The ArrayDeque class extends **AbstractCollection** class and implements **Deque** interface. **Deque** interface extends queue interface. It also implements **Cloneable** and **Serializable** marker interfaces

* ArrayDeque can be used as a **stack** (LIFO) as well as a **queue**(FIFO)
* Performance of ArrayDeque is sometimes considered as the best among the collection framework. It gives performance of **O(1)** for insertion, removal and retrieval operations

As ques

* // arrayDeque.offer("One"); insert an element
* //  arrayDeque.offer("Five"); // remove an element

As stack

* Push and pop() methods