**JAVA-8 Features**

**Functional Interface (SAM/Single Abstract Method Interface)**

* **An interface that contains exactly one abstract method is known as Functional Interface.**
* Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.
* It can have any number of default, static methods but can contain only one abstract method.
* It can also declare methods of object class.(Example-2)
* It helps to achieve Functional Programming Approach.
* A functional interface can extends to other interface only when that does not have any abstract method. (Example-3)
* If a Functional Interface extends another interface having an abstract method, then compiler will throw a compile time error. (Example-4)

**Example1**

**package** FunctionalInterface;

@FunctionalInterface

**interface** message{

**void** alert (String msg);

}

**public** **class** FunctionalInterfaceTest1 **implements** message{

**public** **void** alert(String message){

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

}

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

FunctionalInterfaceTest1 fit = **new** FunctionalInterfaceTest1();

fit.alert("Hello World");

}

}

**Example2**

**package** FunctionalInterface;

@FunctionalInterface

**interface** sayable{

**void** say(String msg); // abstract method

// It can contain any number of methods of Object class.

**int** hashCode();

String toString();

**boolean** equals(Object obj);

}

**public** **class** FunctionalInterfacewithObjectClassMethods **implements** sayable{

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

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

}

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

FunctionalInterfacewithObjectClassMethods fie = **new** FunctionalInterfacewithObjectClassMethods();

fie.say ("Hello World");

}

}

**Example3**

**package** FunctionalInterface;

//Non-Functional Interface

**interface** doable{

**default** **void** doIt(){

System.***out***.println("Do it now");

}

}

@FunctionalInterface

**interface** sayable1 **extends** doable{

**void** say(String msg); // abstract method

}

**public** **class** FunctionalInterfaceExtendingNonFunctionalInterface **implements** sayable1{

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

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

}

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

FunctionalInterfaceExtendingNonFunctionalInterface fie = **new** FunctionalInterfaceExtendingNonFunctionalInterface();

fie.say("Hello World");

fie.doIt();

}

}

**Example4**

**package** FunctionalInterface;

**interface** sayable2{

**void** say(String msg); // abstract method

}

@FunctionalInterface

**interface** doable1 **extends** sayable{

// Invalid '@FunctionalInterface' annotation; doable1 is not a functional interface

**void** doIt();

}

**public** **class** FunctionalInterfaceImplementingNonFuctionaInterface {

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

}

}

**Output: Compiletime Error**

**Predefined Functional Interface**

* Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.
* We can also define our own custom functional interface.
* Following is the list of functional interface which are placed in java.util.function package.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument and returns boolean value. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

**Consumer Interface** **(**[**Consumer<T>**](https://www.javatpoint.com/java-consumer-interface)**)**

* The Consumer Interface accepts a single argument and does not return any result.
* It is a functional interface defined in java.util.function package.
* It contains an abstract accept() and a default andThen() method.
* It can be used as the assignment target for a lambda expression or method reference.

**Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void accept(T t) | It performs this operation on the given argument. |
| default Consumer<T> andThen(Consumer<? super T> after) | It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed. |

Any input type

Consumer<T>

**Example1**

**package** Consumer;

**import** java.util.function.Consumer;

**public** **class** ConsumerTest {

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

Consumer<String> c=s->System.***out***.println(s);

Consumer<Integer> c1=s->System.***out***.println(s);

c.accept("Hello");

c.accept("World");

c1.accept(10);

}

}

**Output:** Hello

World

10

**Example2**

**package** Consumer;

**import** java.util.function.Consumer;

**public** **class** ConsumerInterfaceTest2 {

**static** **void** printMessage(String name)

{

System.***out***.println("Hello "+name);

}

**static** **void** printValue(**int** val)

{

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

}

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

// Referring method to String type Consumer interface

Consumer<String> consumer1 = ConsumerInterfaceTest2::*printMessage*;

consumer1.accept("Smruti"); // Calling Consumer method

// Referring method to Integer type Consumer interface

Consumer<Integer> consumer2 = ConsumerInterfaceTest2::*printValue*;

consumer2.accept(12); // Calling Consumer method

}

}

**Output:** Hello Smruti

12

**BiConsumer Interface** **(**[**BiConsumer<T,U>**](https://www.javatpoint.com/java-biconsumer-interface)**)**

* It represents an operation that accepts two input arguments and does not return any result.
* This is the two-arity specialization of Consumer interface.
* It provides a functional method accept (Object, Object) to perform custom operations.

**Lambda Expression**

* It provides a clear & concise way to represent one method interface using an epression.
* It is very useful in collection library.
* It helps to iterate, filter & extract data from collection.
* It is a replacement of java inner anonymous class.
* Java lambda expression is treated as a function , so compiler does not create .class file.
* Lambda expression provides implementation of functional interface.
* **An interface which has only one abstract method is called functional interface**.
* Java provides an annotation **@FunctionalInterface**, which is used to declare an interface as functional Interface.
* Less Coding.

**Syntax for Lambda Expression**

**(argument-list)->{body};**

* Java lambda expression consists of 3 components :-

1. **Argument-list**: It can be empty or non-empty as well.
2. **Arrow-token:** Used to link argument list & body of expression.
3. **Body**: It consists expressions & statements of lambda expression.

**WithoutLambdaExample1**

**package** lambdaExpression;

**interface** Drawable1{

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

}

**public** **class** WithoutLambdaExample1 {

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

**int** width=30;

Drawable1 d=**new** Drawable1() {

@Override

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

System.***out***.println("Width is:"+width);

}

};

d.draw();

}

}

**LambdaExpressionExample1**

**package** lambdaExpression;

@FunctionalInterface

**interface** Drawable{

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

}

**public** **class** LambdaExpressionExample1 {

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

**int** width=23;

Drawable d=()->{

System.***out***.println("Width is:"+width);

};

d.draw();

}

}

**LambdaExpressionSingleParameter**

**package** lambdaExpression;

@FunctionalInterface

**interface** Sayable{

**public** String say(String name);

}

**public** **class** LambdaExpressionSingleParameter {

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

Sayable s=(name)->{

**return** "Hello "+name;

};

System.***out***.println(s.say("smruti"));

}

}

**LambdaExpessionMultiParameter**

**package** lambdaExpression;

**interface** Addable{

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

}

**public** **class** LambdaExpessionMultiParameter {

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

Addable add1=(a,b)->(a+b);

System.***out***.println(add1.add(19, 23));

Addable add2=(**int** a, **int** b)->(a+b);

System.***out***.println(add2.add(4, 6));

}

}

**LambdaExpressionForEach**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** LambdaExpressionForEach {

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

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

list.add("Smruti");

list.add("Ranjan");

list.add("Gouda");

list.forEach((n)->System.***out***.println(n));

}

}

**LambdaExpressionThread**

**package** lambdaExpression;

**public** **class** LambdaExpressionThread {

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

//Without lambda

Runnable r1=**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Thread withuot lambda is running........");

}

};

Thread t1=**new** Thread(r1);

t1.start();

//With Lambda

Runnable r2=()->{

System.***out***.println("Thread with lambda is running........");

};

Thread t2=**new** Thread(r2);

t2.start();

}

}

**LambdaExpressionforComparator**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Product{

**int** id;

String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

}

**public** **class** LambdaExpressionforComparator{

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

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

//Adding Products

list.add(**new** Product(1,"HP Laptop",25000f));

list.add(**new** Product(3,"Keyboard",300f));

list.add(**new** Product(2,"Dell Mouse",150f));

System.***out***.println("Sorting on the basis of name...");

// implementing lambda expression

Collections.*sort*(list,(p1,p2)->{ **return** p1.name.compareTo(p2.name);

});

**for**(Product p:list){

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

}

}

}

**LambdaExpressionforCollection**

**package** lambdaExpression;

**import** java.util.ArrayList;

**import** java.util.List;

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

**class** Product1{

**int** id;

String name;

**float** price;

**public** Product1(**int** id, String name, **float** price) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

}

**public** **class** LambdaExpressionCollection{

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

List<Product1> list=**new** ArrayList<Product1>();

list.add(**new** Product1(1,"Samsung A5",17000f));

list.add(**new** Product1(3,"Iphone 6S",65000f));

list.add(**new** Product1(2,"Sony Xperia",25000f));

list.add(**new** Product1(4,"Nokia Lumia",15000f));

list.add(**new** Product1(5,"Redmi4 ",26000f));

list.add(**new** Product1(6,"Lenevo Vibe",19000f));

// using lambda to filter data

Stream<Product1> filtered\_data = list.stream().filter(p -> p.price > 20000);

// using lambda to iterate through collection

filtered\_data.forEach(

product1 -> System.***out***.println(product1.name+": "+product1.price)

);

}

}