Lambda Expression in Java 8

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

**Lambda Expressions**

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

<http://viralpatel.net/blogs/lambda-expressions-java-tutorial/>

* A lambda expression can have zero, one or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
* Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
* Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a
* The body of the lambda expressions can contain zero, one or more statements.
* If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Each lambda expression can be implicitly assigned to one of the interface called Functional interface. For example we can create Runnable interface’s reference from lambda expression like below:

Runnable r = () -> System.out.println("hello world");

This type of conversion is automatically taken care by compiler when we dont specify the functional interface. For example:

new Thread(

() -> System.out.println("hello world")

).start();

So in above code, compiler automatically deduced that lambda expression can be casted to Runnable interface from Thread class’s constructor signature public Thread(Runnable r) { }.

## **Examples of Lambda Expression**

Best way of learning about Lambda expressions is by examples. Following are few examples:

Thread can be initialized like following:

//Old way:

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Hello from thread");

}

}).start();

//New way:

new Thread(

() -> System.out.println("Hello from thread")

).start();

The event handling can be done with Java 8 using lambda expression. Following code we show both old and new way of adding ActionListener to a UI component.

//Old way:

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("The button was clicked using old fashion code!");

}

});

//New way:

button.addActionListener( (e) -> {

System.out.println("The button was clicked. From lambda expressions !");

});

Simple code to print all elements of given array. Note there is one more way of using lambda expression. In below example we use the usual way of creating lambda expression using arrow syntax and also we used a brand new double colon (::) operator that Java 8 has to convert a normal method into lambda expression.

//Old way:

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

for(Integer n: list) {

System.out.println(n);

}

//New way:

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

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

//or we can use :: double colon operator in Java 8

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

In this example we use Predicate functional interface to create a test and print the elements that pass the test. This way you can provide the logic using lambda expression and do something based on it.

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class Main {

public static void main(String [] a) {

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

System.out.println("Print all numbers:");

evaluate(list, (n)->true);

System.out.println("Print no numbers:");

evaluate(list, (n)->false);

System.out.println("Print even numbers:");

evaluate(list, (n)-> n%2 == 0 );

System.out.println("Print odd numbers:");

evaluate(list, (n)-> n%2 == 1 );

System.out.println("Print numbers greater than 5:");

evaluate(list, (n)-> n > 5 );

}

public static void evaluate(List<Integer> list, Predicate<Integer> predicate) {

for(Integer n: list) {

if(predicate.test(n)) {

System.out.println(n + " ");

}

}

}

}

**Output:**

Print all numbers: 1 2 3 4 5 6 7

Print no numbers:

Print even numbers: 2 4 6

Print odd numbers: 1 3 5 7

Print numbers greater than 5: 6 7

**Another example on Lambda expression**

**Predicate Lambda, ArrayList.** The term predicate is used in computer science to mean a boolean-returning method. A Predicate object receives one value and returns true or false.

**RemoveIf:**This method on ArrayList receives a Predicate. Here, we remove all elements starting with the letter "c."

**Java program that uses removeIf, Predicate lambda**

import java.util.ArrayList;

public class Program {

public static void main(String[] args) {

// Create ArrayList and add four String elements.

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

list.add("cat");

list.add("dog");

list.add("cheetah");

list.add("deer");

// Remove elements that start with c.

list.**removeIf**(element -> element.startsWith("c"));

System.out.println(list.toString());

}

}

**Output**

[dog, deer]

<http://www.tutorialspoint.com/java8/java8_lambda_expressions.htm>

## **Syntax**

A lambda expression is characterized by the following syntax −

parameter -> expression body

Following are the important characteristics of a lambda expression −

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

# Some working examples

**Interface1**

**package** com.ddlab.rnd.core;  
**public interface** Interface1 {  
 **void** test();  
}

**TestLambda1**

**package** com.ddlab.rnd.core;  
**public class** TestLambda1 **implements** Interface1 {  
 **public void** test() {  
 System.***out***.println(**"Hello World to Lambda Expression ..."**);  
 }  
  
 **public static void** checkLambda(Interface1 interface1) {  
 interface1.test();  
 }  
  
 **public static void** main(String[] args) {  
  
 Interface1 interface1 = () -> {  
 System.***out***.println(**"Welcome to the world of Lambda"**);  
 };  
 *checkLambda*(interface1);  
 }  
}

**interface2**

**package** com.ddlab.rnd.core;  
**public interface** Interface2 {  
  
 **int** sum(**int** a,**int** b);  
}

**TestLamda2**

**package** com.ddlab.rnd.core;  
**public class** TestLamda2 {  
  
 **public static void** check(Interface2 interface2, **int** a, **int** b) {  
 System.***out***.println(interface2.sum(a, b));  
 }  
  
 **public static void** main(String[] args) {  
 Interface2 interface2 = (a, b) -> a + b;  
 *check*(interface2, 10, 15);  
  
 *//OLD way of writing* interface2 = **new** Interface2() {  
 @Override  
 **public int** sum(**int** a, **int** b) {  
 **return** a+b;  
 }  
 };  
 *check*(interface2, 20, 25);  
 }  
}

**TestListComparator**

**package** com.ddlab.rnd.core;  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.Comparator;  
**import** java.util.List;  
  
**public class** TestListComparator {  
  
 **public static void** main(String[] args) {  
 List<String> stringList = **new** ArrayList<>();  
  
 stringList.add(**"John Abraham"**);  
 stringList.add(**"Vidya Balan"**);  
 stringList.add(**"Nana Patekar"**);  
 stringList.add(**"Johny Depp"**);  
 stringList.add(**"Kate Winslet"**);  
 System.***out***.println(stringList);  
 Comparator<String> comparator = (s1,s2) -> {  
*// return s2.compareTo(s1);//Descending order* **return** s1.compareTo(s2); *//Ascending order* };  
  
 Collections.*sort*(stringList,comparator);  
 System.***out***.println(stringList);  
 }  
}

**Thread1**

**package** com.ddlab.rnd.core;  
**public class** Thread1 {  
  
 **public static void** main(String[] args) {  
 Runnable runnable = () -> {  
 **for** (**int** i = 0; i < 10; i++) {  
 System.***out***.println(**"Thread is running"**);  
 }  
 };  
 Thread th1 = **new** Thread(runnable);  
 th1.start();  
 }  
}

**TestListComparator1.java**

**package** com.ddlab.rnd.core;  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.List;  
  
**public class** TestListComparator1 {  
  
 **public static void** main(String[] args) {  
 List<String> stringList = **new** ArrayList<>();  
 stringList.add(**"John Abraham"**);  
 stringList.add(**"Vidya Balan"**);  
 stringList.add(**"Nana Patekar"**);  
 stringList.add(**"Johny Depp"**);  
 stringList.add(**"Kate Winslet"**);  
 System.***out***.println(stringList);  
 *//Sor in descending order* Collections.*sort*(stringList, (String a, String b) -> b.compareTo(a));  
 System.***out***.println(stringList);  
 *//Another way of writing the lambda expression* Collections.*sort*(stringList, (a, b) -> b.compareTo(a));  
 System.***out***.println(stringList);  
 }  
}

Functional Interfaces in Java 8

Many new functional interfaces are being defined in the Java 8, among the most popular, those found in the new java.util.function package. The following are some examples of new functional interfaces in Java:

public interface Predicate<T> {

boolean test(T t);

}

public interface Function<T,R> {

R apply(T t);

}

public interface BinaryOperator<T> {

T apply(T left, T right);

}

public interface Consumer<T> {

void accept(T t);

}

public interface Supplier<T> {

T get();

}

<http://www.tutorialspoint.com/java8/java8_functional_interfaces.htm>

Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions. Following is the list of functional interfaces defined in java.util.Function package.

|  |  |
| --- | --- |
| **S. No.** | **Interface & Description** |
| 1 | **BiConsumer<T,U>**  Represents an operation that accepts two input arguments, and returns no result. |
| 2 | **BiFunction<T,U,R>**  Represents a function that accepts two arguments and produces a result. |
| 3 | **BinaryOperator<T>**  Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| 4 | **BiPredicate<T,U>**  Represents a predicate (Boolean-valued function) of two arguments. |
| 5 | **BooleanSupplier**  Represents a supplier of Boolean-valued results. |
| 6 | **Consumer<T>**  Represents an operation that accepts a single input argument and returns no result. |
| 7 | **DoubleBinaryOperator**  Represents an operation upon two double-valued operands and producing a double-valued result. |
| 8 | **DoubleConsumer**  Represents an operation that accepts a single double-valued argument and returns no result. |
| 9 | **DoubleFunction<R>**  Represents a function that accepts a double-valued argument and produces a result. |
| 10 | **DoublePredicate**  Represents a predicate (Boolean-valued function) of one double-valued argument. |
| 11 | **DoubleSupplier**  Represents a supplier of double-valued results. |
| 12 | **DoubleToIntFunction**  Represents a function that accepts a double-valued argument and produces an int-valued result. |
| 13 | **DoubleToLongFunction**  Represents a function that accepts a double-valued argument and produces a long-valued result. |
| 14 | **DoubleUnaryOperator**  Represents an operation on a single double-valued operand that produces a double-valued result. |
| 15 | **Function<T,R>**  Represents a function that accepts one argument and produces a result. |
| 16 | **IntBinaryOperator**  Represents an operation upon two int-valued operands and produces an int-valued result. |
| 17 | **IntConsumer**  Represents an operation that accepts a single int-valued argument and returns no result. |
| 18 | **IntFunction<R>**  Represents a function that accepts an int-valued argument and produces a result. |
| 19 | **IntPredicate**  Represents a predicate (Boolean-valued function) of one int-valued argument. |
| 20 | **IntSupplier**  Represents a supplier of int-valued results. |
| 21 | **IntToDoubleFunction**  Represents a function that accepts an int-valued argument and produces a double-valued result. |
| 22 | **IntToLongFunction**  Represents a function that accepts an int-valued argument and produces a long-valued result. |
| 23 | **IntUnaryOperator**  Represents an operation on a single int-valued operand that produces an int-valued result. |
| 24 | **LongBinaryOperator**  Represents an operation upon two long-valued operands and produces a long-valued result. |
| 25 | **LongConsumer**  Represents an operation that accepts a single long-valued argument and returns no result. |
| 26 | **LongFunction<R>**  Represents a function that accepts a long-valued argument and produces a result. |
| 27 | **LongPredicate**  Represents a predicate (Boolean-valued function) of one long-valued argument. |
| 28 | **LongSupplier**  Represents a supplier of long-valued results. |
| 29 | **LongToDoubleFunction**  Represents a function that accepts a long-valued argument and produces a double-valued result. |
| 30 | **LongToIntFunction**  Represents a function that accepts a long-valued argument and produces an int-valued result. |
| 31 | **LongUnaryOperator**  Represents an operation on a single long-valued operand that produces a long-valued result. |
| 32 | **ObjDoubleConsumer<T>**  Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| 33 | **ObjIntConsumer<T>**  Represents an operation that accepts an object-valued and an int-valued argument, and returns no result. |
| 34 | **ObjLongConsumer<T>**  Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| 35 | **Predicate<T>**  Represents a predicate (Boolean-valued function) of one argument. |
| 36 | **Supplier<T>**  Represents a supplier of results. |
| 37 | **ToDoubleBiFunction<T,U>**  Represents a function that accepts two arguments and produces a double-valued result. |
| 38 | **ToDoubleFunction<T>**  Represents a function that produces a double-valued result. |
| 39 | **ToIntBiFunction<T,U>**  Represents a function that accepts two arguments and produces an int-valued result. |
| 40 | **ToIntFunction<T>**  Represents a function that produces an int-valued result. |
| 41 | **ToLongBiFunction<T,U>**  Represents a function that accepts two arguments and produces a long-valued result. |
| 42 | **ToLongFunction<T>**  Represents a function that produces a long-valued result. |
| 43 | **UnaryOperator<T>**  Represents an operation on a single operand that produces a result of the same type as its operand. |

## **What is functional interface**

Functional interfaces are new additions in [**java 8**](http://howtodoinjava.com/category/java-8/) which **permit exactly one abstract method inside them**. These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**. These can be represented using Lambda expressions, Method reference and constructor references as well. Java 8 introduces an annotation i.e. **@FunctionalInterface** too, which can be used for compiler level [errors](http://howtodoinjava.com/best-practices/java-exception-handling-best-practices/) when the interface you have annotated violates the contracts of Functional Interface.

Let’s build our first functional interface:

|  |
| --- |
| package functionalInterfaceExample;    @FunctionalInterface  public interface MyFirstFunctionalInterface {  public void firstWork();  } |

Below is list of things which are allowed and which are not in a functional interface.

**A)** As discussed above, ***only one abstract method is allowed*** in any functional interface. Second abstract method is not not permitted in a functional interface. If we remove **@FunctionInterface** annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface.

**B)** A functional interface is ***valid even if the @FunctionalInterface annotation would be omitted***. It is only for informing the compiler to enforce single [abstract method](http://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface.

**C)** Conceptually, a functional interface has exactly one abstract method. Since [**default methods**](http://howtodoinjava.com/java-8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re ***free to add default methods to your functional interface as many as you like***.  
Below is valid functional interface:

|  |
| --- |
| package functionalInterfaceExample;    @FunctionalInterface  public interface MyFirstFunctionalInterface {  public void firstWork();  default void doSomeMoreWork1(){  //Method body  }  default void doSomeMoreWork2(){  //Method body  }  } |

**D)** If an interface declares an ***abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count*** since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. e.g. [**Comparator**](http://howtodoinjava.com/search-sort/when-to-use-comparable-and-comparator-interfaces-in-java/) is a functional interface even though it declared two abstract methods. Why? Because one of these abstract methods  “equals()” which has signature equal to public method in Object class.

e.g. Below interface is a valid functional interface.

|  |
| --- |
| package functionalInterfaceExample;  @FunctionalInterface  public interface MyFirstFunctionalInterface {  public void firstWork();  @Override  public String toString();                //Overridden from Object class  @Override  public boolean equals(Object obj);        //Overridden from Object class  }  Note: You can not declare FunctionalInterface without any method |

**@FunctionalInterface // <- error here**

**public interface FunctionalInterfaceExample {**

**}**

The compiler will raise an error as there is no method. It says that “Example is not a functional interface” as “no abstract method was found”. It’ll also error if we try and add a second method.

## **Extension**

What about the case of an interfaces that extends another interfaces?

Let’s create a new functional interface called A and another called B. B extends A. B is still “functional”. It inherits the parents apply method as you’d expect.

|  |
| --- |
| @FunctionalInterface  interface A {  abstract void apply();  }  interface B extends A {  } |

If you wanted to make this clearer, you can also override the functional method from the parent.

|  |
| --- |
| @FunctionalInterface  interface A {  abstract void apply();  }  interface B extends A {  @Override  abstract void apply();  } |

We can verify it works as a functional interface if we use it as a lambda. So I’ll implement a little method to show that a lambda can be assigned to a type of A and a type of B. The implementation will just print out “A” or “B”.

|  |
| --- |
| @FunctionalInterface  public interface A {  void apply();  }  public interface B extends A {  @Override  void apply();  }  public static void main(String... args) {  A a = () -> System.out.println("A");  B b = () -> System.out.println("B");  } |

You can’t add a new abstract method to the extending interface though, as the resulting type would have two abstract methods and the compiler will error.

|  |
| --- |
| @FunctionalInterface  public interface A {  void apply();  }  public interface B extends A {  void illegal();  }  public static void main(String... args) {  A a = () -> System.out.println("A");  B b = () -> System.out.println("B"); // <- error  } |

In both cases, you can override methods from Object without causing problems. You can also add default methods (also new to Java 8). As you’d probably expect, it doesn’t make sense to try and mark an abstract class as a functional interface.

# Working Examples on @Functional Interface

**package** com.ddlab.rnd.funtionalinterface;  
  
@FunctionalInterface  
**public interface** Validator {  
 **boolean** isValid( String s );  
  
 *//You can declare methods from Object class* String toString();  
 **boolean** equals(Object object);  
}

**package** com.ddlab.rnd.funtionalinterface;  
**public class** ValidationImpl {  
  
 **public static void** check(String s, Validator validator) {  
 System.***out***.println(**"Validity : "** + validator.isValid(s));  
 }  
  
 **public static void** main(String[] args) {  
 Validator validator = (s) -> {  
 **if** (s == **null**)  
 **return false**;  
 **else  
 return true**;  
 };  
 String s = **null**;  
 *check*(s, validator);  
 s = **"abcd"**;  
 *check*(s,validator);  
 }  
}

**package** com.ddlab.rnd.funtionalinterface;  
  
@FunctionalInterface  
**interface** Converter<F, T> {  
 T convert(F from);  
}

**package** com.ddlab.rnd.funtionalinterface;  
**public class** TestConverter {  
 **public static void** main(String[] args) {  
 Converter<String, Integer> converter = (from) -> Integer.*valueOf*(from);  
 Integer converted = converter.convert(**"123"**);  
 System.***out***.println(converted); *// 123* }  
}

# Java 8 Predicates and Functions

## **Predicate**

Predicates represent single argument functions that return a boolean value:

**Simple predicate**

Predicate<Integer> greaterThanTen = (i) -> i > 10;

// Will print true

greaterThanTen.test(14);

Predicates may also be chained together by the means of **and**, **or** and **negate**. Following next is a simple example but one may write complex evaluation rules by chaining predicates:

**Predicate chaining**

Predicate<Integer> greaterThanTen = (i) -> i > 10;

Predicate<Integer> lowerThanTwenty = (i) -> i < 20;

// Will print true

greaterThanTen.and(lowerThanTwenty).test(15);

// Will print false

greaterThanTen.and(lowerThanTwenty).negate().test(15)

Predicates may also be passed into functions:

**Passing predicates into functions**

// Will print "Number 10 was accepted!"

process(10, (i) -> i > 7);

void process(int number, Predicate<Integer> predicate) {

if (predicate.test(number)) {

System.out.println("Number " + number + " was accepted!");

}

}

Another example:

**Filtering list elements with a predicate**

List<User> users = new ArrayList<>();

users.add(new User("John", "admin"));

users.add(new User("Peter", "member"));

List<User> admins = process(users, (u) -> u.getRole().equals("admin"));

List<User> process(List<User> users, Predicate<User> predicate) {

List<User> result = new ArrayList<>();

for (User user : users) {

if (predicate.test(user)) {

result.add(user);

}

}

return result;

}

# Working example is given below

**package** com.ddlab.rnd.predicate;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.function.Predicate;  
  
**public class** TestPredicate1 {  
  
 **public static void** showSinglePredicate() {  
 *//Single predicate* Predicate<Integer> greaterThanTen = (i) -> i > 10;  
 System.***out***.println(**"Predicate Value : "**+greaterThanTen.test(14));*//True* }  
  
 **public static void** showPredicateChaining() {  
 *//Predicate chaining* Predicate<Integer> greaterThanTen = (i) -> i > 10;  
 Predicate<Integer> lowerThanTwenty = (i) -> i < 20;  
  
 **boolean** flag = greaterThanTen.and(lowerThanTwenty).test(15);  
 System.***out***.println(flag);  
 }  
  
 *//Passing predicates into functions* **public static void** process(**int** number, Predicate<Integer> predicate) {  
 **if** (predicate.test(number)) {  
 System.***out***.println(**"Number "** + number + **" was accepted!"**);  
 }  
 }  
  
 **static class** User {  
 String **name**;  
 String **role**;  
  
 **public** User(String name, String role) {  
 **this**.**name** = name;  
 **this**.**role** = role;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public** String getRole() {  
 **return role**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "User{"** +  
 **"name='"** + **name** + **'\''** +  
 **", role='"** + **role** + **'\''** +  
 **'}'**;  
 }  
 }

*// Filtering list elements with a predicate* **public static** List<User> process(List<User> users, Predicate<User> predicate) {  
 List<User> result = **new** ArrayList<>();  
 **for** (User user : users) {  
 **if** (predicate.test(user)) {  
 result.add(user);  
 }  
 }  
 **return** result;  
 }  
  
 **public static void** main(String[] args) {  
*// showPredicateChaining();  
  
 // Will print "Number 10 was accepted!"  
// process(10, (i) -> i > 7);* List<User> users = **new** ArrayList<>();  
 users.add(**new** User(**"John"**, **"admin"**));  
 users.add(**new** User(**"Peter"**, **"member"**));  
 List<User> admins = *process*(users, (u) -> u.getRole().equals(**"admin"**));  
 **for**( User u : admins )  
 System.***out***.println(u);  
 }  
}

## **Function**

Functions also represent a single argument function but they return a result of an arbitrary type:

**Simple function**

Function<String, Integer> stringLength = (s) -> s.length();

// Will print 11

stringLength.apply("Hello world");

Functions may also be chained:

**Function chaining**

Function<String, Integer> stringLength = (s) -> s.length();

Function<Integer, Boolean> greaterThanFive = (i) -> i > 5;

// Will print true

stringLength.andThen(greaterThanFive).apply("Hello world");

Another function chaining example:

**Another function chaining example**

Function<String, Integer> stringLength = (s) -> s.length();

Function<Integer, Boolean> lowerThanTen = (i) -> i < 10;

Function<String, Boolean> function = stringLength.andThen(lowerThanTen);

// Will print false

function.apply("Hello world");

# Working example is given below

**package** com.ddlab.rnd.predicate;  
**import** java.util.function.Function;  
  
**public class** TestPredicate2 {  
  
 *//Simple Function* **public static void** testFunction1() {  
 Function<String, Integer> stringLength = (s) -> s.length();  
 System.***out***.println(stringLength.apply(**"Hello world"**));*//Will print 11* }  
  
 *//Function chaining* **public static void** functionChaining() {  
 Function<String, Integer> stringLength = (s) -> s.length();  
 Function<Integer, Boolean> greaterThanFive = (i) -> i > 5;  
 *// Will print true* System.***out***.println(stringLength.andThen(greaterThanFive).apply(**"Hello world"**));  
 }  
  
 *//Function chaining* **public static void** functionChaining1() {  
 Function<String, Integer> stringLength = (s) -> s.length();  
 Function<Integer, Boolean> lowerThanTen = (i) -> i < 10;  
 Function<String, Boolean> function = stringLength.andThen(lowerThanTen);  
 *// Will print false* System.***out***.println(function.apply(**"Hello world"**));  
 }  
  
 **public static void** main(String[] args) {  
*// testFunction1();  
// functionChaining();  
 functionChaining1*();  
 }  
}

Default methods in java.util.function.Predicate: There a few default methods also provided in the Predicate functional interface which enable us to do various types of boolean operations such as and, or, not(negate) with different instances of Predicate. These default methods are –

| **Default Method Name** | **Explanation** |
| --- | --- |
| and() | It does logical AND of the predicate on which it is called with another predicate. Example: predicate1.and(predicate2) |
| or() | It does logical OR of the predicate on which it is called with another predicate. Example: predicate1.or(predicate2) |
| negate() | It does boolean negation of the predicate on which it is invoked. Example:predicate1.negate() |

# Java 8 UnaryOperator and BinaryOperator Example

Java 8 has introduced UnaryOperator and BinaryOperator that can be assigned as lambda expression. UnaryOperator and BinaryOperator are functional interface. UnaryOperator extends Function and BinaryOperator extends BiFunction and accordingly they accept argument. UnaryOperator accepts one operand and returns a value of the same type as operand. BinaryOperator accepts two operand of the same type and returns the result of the same type as operand.

### **UnaryOperator**

java.util.function.UnaryOperator is a java 8 functional interface that extends java.util.function.Function. UnaryOperator is used to work on a single operand. It returns the same type as an operand. UnaryOperator can be used as lambda expression to pass as an argument. While defining UnaryOperator, we need to define Function.apply(Object) where Function will be the instance of UnaryOperator. Find the example.   
**UnaryOperatorDemo.java**

package com.concretepage.util.stream;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.function.UnaryOperator;

public class UnaryOperatorDemo {

public static void main(String[] args) {

List<Integer> list = Arrays.asList(10,20,30,40,50);

unaryOperatorFun(i->i\*i,list).forEach(x->System.out.println(x));

}

private static List<Integer> unaryOperatorFun(UnaryOperator<Integer> unaryOpt, List<Integer> list){

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

list.forEach(i->uniList.add(unaryOpt.apply(i)));

return uniList;

}

}

Find the output.

100

400

900

1600

2500

### **BinaryOperator**

java.util.function.BinaryOperator is a functional interface that can be assigned as lambda expression. BinaryOperator extends java.util.function.BiFunction. It accepts two operands of the same type and process it and then returns results of the same type as operands.   
**BinaryOperatorDemo.java**

package com.concretepage.util.stream;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.function.BinaryOperator;

public class BinaryOperatorDemo {

public static void main(String[] args) {

Map<String,String> map = new HashMap<>();

map.put("X", "A");

map.put("Y", "B");

map.put("Z", "C");

binaryOperatorFun((s1,s2)-> s1+"-"+s2,map).forEach(x->System.out.println(x));

}

private static List<String> binaryOperatorFun(BinaryOperator<String> binaryOpt, Map<String,String> map){

List<String> biList = new ArrayList<>();

map.forEach((s1,s2)->biList.add(binaryOpt.apply(s1,s2)));

return biList;

}

}

Find the output.

X-A

Y-B

Z-C

### **BinaryOperator.maxBy and BinaryOperator.minBy**

BinaryOperator.maxBy accepts a Compotator and returns BinaryOperator which will return maximum between two elements. In the same way, BinaryOperator.minBy is used to get minimum between two elements.   
**MaxByMinBy.java**

package com.concretepage.util;

import java.util.Arrays;

import java.util.Comparator;

import java.util.List;

import java.util.Map;

import java.util.Optional;

import java.util.function.BinaryOperator;

import java.util.stream.Collectors;

public class MaxByMinBy {

public static void main(String[] args) {

Student s1 = new Student("Shyam", 22,"A");

Student s2 = new Student("Ram",23,"A");

Student s3 = new Student("Mohan",22,"B");

Student s4 = new Student("Ramesh",21,"B");

List<Student> list = Arrays.asList(s1,s2,s3,s4);

Comparator<Student> ageComparator = Comparator.comparing(Student::getAge);

//Using BinaryOperator.maxBy

System.out.println("---BinaryOperator.maxBy---");

Map<String, Optional<Student>> eldestByClass = list.stream().collect(Collectors.groupingBy(Student::getClassName,

Collectors.reducing(BinaryOperator.maxBy(ageComparator))));

eldestByClass.forEach((k,v)->System.out.println("Class:"+k+" Age:"+

((Optional<Student>)v).get().getAge()+" Name:"+((Optional<Student>)v).get().getName()));

//Using BinaryOperator.minBy

System.out.println("---BinaryOperator.minBy---");

Map<String, Optional<Student>> youngestByClass = list.stream().collect(Collectors.groupingBy(Student::getClassName,

Collectors.reducing(BinaryOperator.minBy(ageComparator))));

youngestByClass.forEach((k,v)->System.out.println("Class:"+k+" Age:"+

((Optional<Student>)v).get().getAge()+" Name:"+((Optional<Student>)v).get().getName()));

}

}

Find the Student class used in Example.   
**Student.java**

package com.concretepage.util;

public class Student {

private String name;

private Integer age;

private String className;

public Student(String name,Integer age, String className){

this.name=name;

this.age=age;

this.className = className;

}

public String getName() {

return name;

}

public Integer getAge() {

return age;

}

public String getClassName() {

return className;

}

}

Find the output.

---BinaryOperator.maxBy---

Class:A Age:23 Name:Ram

Class:B Age:22 Name:Mohan

---BinaryOperator.minBy---

Class:A Age:22 Name:Shyam

Class:B Age:21 Name:Ramesh

# Supplier

**A supplier of objects. The result objects are either created during the invocation of get() or by some prior action.**

And the sole method in this interface is **get()** which should return an object of the declared type.

To understand the usage of this interface, lets consider 2 classes- Vehicle and Car defined as:

class Vehicle{

public void drive(){

System.out.println("Driving vehicle ...");

}

}

class Car extends Vehicle{

@Override

public void drive(){

System.out.println("Driving car...");

}

}

And lets consider a static method which invokes the drive() method on the instance of Vehicle or Car:

public class SupplierDemo {

static void driveVehicle(Supplier<? extends Vehicle> supplier){

Vehicle vehicle = supplier.get();

vehicle.drive();

}

}

And to invoke the above **driveVehicle** method, we should provide an implementation of the Supplier interface which we can do by passing in a [lambda expression](http://blog.sanaulla.info/2013/03/11/using-lambda-expression-to-sort-a-list-in-java-8-using-netbeans-lambda-support/). Lets look at the code:

public static void main(String[] args) {

//Using Lambda expression

driveVehicle(()-> new Vehicle());

driveVehicle(()-> new Car());

//Using method expression

driveVehicle(Vehicle::new);

driveVehicle(Car::new);

}

The first two invocations are familiar to the readers of my blog, but the other two are something new and those are called as “**Method Expression**“.

## Consumer in Java 8

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Consumer<T>**

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

An example is given below.

**package** com.ddlab.rnd.predicate;  
**import** java.util.function.Consumer;  
  
**public class** ConsumerTest {  
  
 **public static void** consumeString(Consumer<String> consumer, String x) {  
 consumer.accept(x);  
 }  
  
 **public static void** main(String[] args) {  
 Consumer<String> function = x -> System.***out***.println(x);  
 Consumer<String> function2 = x -> System.***out***.println(x.toLowerCase());  
 *consumeString*(function, **"StringA"**);*// prints StringA  
 consumeString*(function2,**"StringA"**);*// prints stringa* }  
}