**Java 8**

## 

## **Interface default and static methods.**

* Prior to java 8, [interface in java](https://beginnersbook.com/2013/05/java-interface/) can only have abstract methods. (declaration)
* All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

## **Why default method?**

* For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface.
* This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.
* We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

**interface** Java8Interface{

/\* This is a default method so we need not to implement this method in the implementation classes \*/

**default** **void** newMethod(){

System.***out***.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\*/

**static** **void** anotherNewMethod(){

System.***out***.println("Newly added static method");

}

/\* Already existing public and abstract method. We must need to implement this method in implementation classes.

\*/

**void** existingMethod(String str);

}

**public** **class** DefaultInterfaceMethodDemo **implements** Java8Interface {

// implementing abstract method

**public** **void** existingMethod(String str){

System.***out***.println("String is: "+str);

}

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

DefaultInterfaceMethodDemo obj = **new** DefaultInterfaceMethodDemo ();

//calling the default method of interface

obj.newMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 features");

//calling the static method of interface

Java8Interface.*anotherNewMethod*();

}

}

**An Example of Diamond Problem with Default Methods**Java 8 is also supporting [**multiple inheritances**](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html#axzz55Zuelntp) of classes?  
  
If you look closely, you will find that multiple inheritances of classes are **not supported** in Java 8, instead, the **compiler will do additional checks** to avoid ambiguity in calling default methods and Diamond problem, which could come if a class implements two interfaces that contain the default methods with the same name.

Class D extends A implements B, C {

}

As shown below, this code will not compile in Java 8, because of ambiguity in calling the default method write() from a class, which extends both Poet and Writer interface.  
  
interface Poet

{

default void write()

{

System.out.println("Poet's default method");

}

}

**interface** Writer

{

**default** **void** write() {

System.***out***.println("Writer's default method");

}

}

**public** **class** Multitalented **implements** Poet, Writer

{

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

Multitalented john = **new** Multitalented();

john.write(); // which write method to call, Poet or Writer

}

}

Output: Compile Time Error : class Multitalented inherits unrelated defaults for write() from types Poet and Writer You can see that both Poet and Writer interface have a default method write() and when we have created a class Multitalented, which implements both Poet and Writer it gets the write() method from both super interface Poet and Writer.  
  
Now, the problem is not that we have implemented two interfaces with default methods of the same name. The Actual problems come when we created an object of a Multitalented class and called the write() method on its object

## 

## **How to avoid Diamond Problem With Default Methods in Java 8**

In order to solve this error, you need to override the write() method in your implementation class i.e. class *Multitalented* here, this will remove the ambiguity, making the compiler happy enough to compile this class.

**public** **class** Multitalented **implements** Poet, Writer

{

@Override

**public** **void** write(){

System.***out***.println("Writing stories now days");

}

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

Multitalented john = **new** Multitalented();

john.write();

}

}

# **Lambda Expression in Java 8**

Lambda expressions are similar to methods, it has arguments, a body and return type. They can also be called as anonymous methods. A method without name



One of the main features of lambda expressions is it enables passing a method as argument to another method. I see that because of this feature it is going to unravel umpteen programming possibilities.

## **Functional Interfaces**

* An interface with no method is a marker interface. E.g Cloneable, Serializable
* An interface with only one abstract method is a functional interface. ActionListener class is an example of functional interface.
* We use anonymous class to implement ActionListener and in this kind of scenario, instead of using anonymous inner classes to implement, lambda expressions can be used. It will be simple and better compared to anonymous inner classes.

btn.addActionListener( **new** ActionListener() {

**public** **void** actionPerformed( ActionEvent event ) {

Toolkit.getDefaultToolkit().beep();

}

});

## **Structure of a Lambda Expression**

(Argument List) Arrow Token {Body }

## Example Lambda Expressions

() -> { System.out.printlns("Hello World!");}

(**int** a, **int** b) -> a + b

() -> { **return** 1; }

(String name) -> { System.out.println("Hello "+name); }

n -> n % 2 != 0

(**double** radius) -> Math.PI \* radius \* radius

radius -> { **return** Math.PI \* radius \* radius; }

radius -> { System.out.println(radius); **return** Math.PI \* radius \* radius; }

* The first example's expression-based lambda body doesn't have to be placed between braces.
* The second example converts the expression-based body to a statement-based body, in which return must be specified to return the expression's value.
* The final example demonstrates multiple statements and cannot be expressed without the braces.

## **Lambda Expression – Key Notes**

* One important difference between anonymous inner class and lambda expression is, if we use ‘this’ it resolves to anonymous class but in the case of lambda expression, it resolves to the enclosing class.
* Lambda expression can have zero to any number of parameters.
* If there are no parameters to be passed, then an empty parentheses is given.
* Type of the passed parameter can be explicitly declared or can be taken from context.
* If the parameter’s type is inferred from the context then the parentheses need not be used.
* Lambda expression body can have zero to any number of statements.
* Body of expression should be enclosed in curly braces, if there is only one statement curly brace is not needed.

### **1. Lambda Expression Hello World**

**public** **class** LambdaHelloWorld {

**interface** HelloWorld {

String hello(String name);

}

**class** Abc **implements** HelloWorld(){

String hello(String name){ syso(“Hello”+name);}

}

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

HelloWorld helloWorld = **new** Abc();

helloWorld.hello(“Joe”);

HelloWorld helloWorld = (String name) -> { **return** "Hello " + name; };

System.out.println(helloWorld.hello("Joe"));

}}

**Sample code of Swing event handling.**

**public** **class** AnonymousListener {

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

JButton anonBtn = **new** JButton("Java Button");

//actionlistener using anonymous class

anonBtn.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent ae) {

System.***out***.println("Anonymous Click!");

}

});

//actionlistener using lambda expression

anonBtn.addActionListener(e -> { System.***out***.println("Lambda Click!");

});

JFrame frame = **new** JFrame("Lambda Expression Sample");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.add(anonBtn, BorderLayout.***CENTER***);

frame.pack();

frame.setVisible(**true**);

}

}

### **2. Lambda Expression with Runnable**

Lambdas simplify the use of functional interfaces, which are annotated interfaces that each declare exactly one abstract method (although they can also declare any combination of default, static, and private methods). For example, the standard class library provides a java.lang.Runnable interface with a single abstract void run() method. This functional interface's declaration appears below:

**@FunctionalInterface**

**public interface Runnable{**

**public abstract void run();**

**}**

The class library annotates Runnable with @FunctionalInterface, which is an instance of the java.lang.FunctionalInterface annotation type. FunctionalInterface is used to annotate those interfaces that are to be used in lambda contexts.

A lambda doesn't have an explicit interface type. Instead, the compiler uses the surrounding context to infer which functional interface to instantiate when a lambda is specified--the lambda is *bound* to that interface. For example, suppose I specified the following code fragment, which passes the previous lambda as an argument to the java.lang.Thread class's Thread(Runnable target) constructor:

**new Thread(() -> System.out.println("Hello"));**

The compiler determines that the lambda is being passed to Thread(Runnable r) because this is the only constructor that satisfies the lambda: Runnable is a functional interface, the lambda's empty formal parameter list () matches run()'s empty parameter list, and the return types (void) also agree. The lambda is bound to Runnable.

Above lambda statement will be same as below code

public static void main(String[] args)

{

Runnable r = new Runnable()

{

@Override

public void run()

{

System.out.println("Hello");

}

};

new Thread(r).start();

}

Complete program of implementation of thread using anonymous class and lambda

**public** **class** RunnableLambdaExample {

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

RunnableLambdaExample obj=**new** RunnableLambdaExample();

// Main thread name

System.***out***.println("Main thread:"+Thread.*currentThread*().getName());

// Implementing Runnable using anonymous class implementation

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

@Override

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

System.***out***.println("Thread with anonymous class :"+Thread.*currentThread*().getName());

}

};

Thread thread1 = **new** Thread(r1);

thread1.start(); //it will call the run method

// Pass Runnable in thread creation itself using anonymous class implementation.

Thread thread2 = **new** Thread(**new** Runnable() {

@Override

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

System.***out***.println("Thread with anonymous class :"+Thread.*currentThread*().getName());

}

});

thread2.start();

// Lambda expression with Runnable interface

Runnable r3 = () -> {

System.***out***.println("Thread with lambda:"+Thread.*currentThread*().getName());

System.***out***.println(Thread.*currentThread*().getPriority());

};

//new Thread(t2).start();

Thread thread3=**new** Thread(r3);

thread3.start();

}

}

**3. Accessing Local and Class Variables in Lambda Expression**

**public** **class** LambdaVariableAccess {

**public** String wildAnimal = "Lion";

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

**new** LambdaVariableAccess().lambdaExpression();

}

**public** **void** lambdaExpression(){

String domesticAnimal = "Dog";

**new** Thread (() -> {

System.***out***.println("Class Level: " + **this**.wildAnimal);

System.***out***.println("Method Level: " + domesticAnimal);

}).start();

}

}

**4. Redeclaring Local Variables in Lambda Expression**

**public** **class** JavaLocalVariableWithLambda

{

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

{

**int** limit = 10;

Runnable r = () -> {

**int** limit = 5;//variable can’t be re-declare in the scope.

**for** (**int** i = 0; i < limit; i++)

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

};

}

}

Because limit is already present in the enclosing scope (the main() method), the lambda body's redefinition of limit (int limit = 5;) causes the compiler to report the error.

### **5. Modify Local Variables in the Lambda Expression**

**public** **class** LambdaLocalVariableModification

{

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

{

**int** limit = 10;

Runnable r = () -> {

limit = 5; //variable defined in an enclosing scope main must be final or effectively final

**for** (**int** i = 0; i < limit; i++)

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

};

}

}

A local variable or parameter that's defined outside a lambda body and referenced from the body must be marked final or considered effectively final (the variable cannot be assigned to after initialization). Attempting to modify an effectively final variable causes the compiler to report an error.

limit is effectively final. The lambda body's attempt to modify this variable causes the compiler to report an error. It does so because a final/effectively final variable will need to hang around until the lambda executes, which may not happen until long after the code in which the variable was defined returns. Non-final/non-effectively final variables no longer exist

### **6. Lambdas and the 'this' keyword**

**public** **class** LambdaAndThis

{

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

{

LambdaAndThis obj = **new** LambdaAndThis();

System.out.printf("this = %s%n", **obj**);

obj.doWork();

}

**public** **void** doWork()

{

System.out.printf("this = %s%n", **this**);

Runnable r = **new** Runnable(){

@Override

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

{

System.out.printf("this = %s%n", **this**);

}

};

**new** Thread(r).start();

**new** Thread(() -> System.out.printf("this = %s%n", **this**)).start();

}

}

this = LambdaAndThis@1db9742

this = LambdaAndThis@1db9742

this = LambdaAndThis$1@119bba2

this = LambdaAndThis@1db9742

* The first line & second line shows LambdaAndThis’s this reference, the third line shows a different this reference in the new Runnable scope, and the fourth output line shows the this reference in a lambda context. The fourth and first lines match because the lambda's scope is nested inside the doWork() method; this has the same meaning throughout this method.

### **7. Lambdas and exceptions**

A lambda body is not allowed to throw more exceptions than those are specified in the throws clause of the functional interface method. If a lambda body throws an exception, the functional interface method's throws clause must declare the same exception type or its supertype.

**import** java.awt.AWTException;

**import** java.io.IOException;

@FunctionalInterface

**interface** Work

{

**void** doSomething() **throws** IOException;

}

**public** **class** LambdaAndException

{

**public** **static** **void** main(String[] args) **throws** AWTException, IOException

{

Work work = () -> { **throw** **new** IOException(); };

work.doSomething();

work = () -> { **throw** **new** AWTException(""); };

}

}

A Work functional interface whose doSomething() method is declared to throw java.io.IOException. The main() method assigns a lambda that throws IOException to work, which is okay because IOException is listed in doSomething()'s throws clause.

### **8) Predefined functional interfaces**

* You might find yourself repeatedly creating similar functional interfaces. For example, you might create a CheckConnection functional interface with a boolean isConnected(Connection c) method and a CheckAccount functional interface with a boolean isPositiveBalance(Account acct) method. This is wasteful.
* The previous examples expose the abstract concept of a *predicate* (a Boolean-valued function). Anticipating such patterns, Oracle provides the java.util.function package of commonly-used functional interfaces. For example, this package's Predicate<T> functional interface can be used in place of CheckConnection and CheckAccount.
* Predicate<T> provides a boolean test(T t) method that evaluates this predicate on its argument (t), returning true when t matches the predicate, and returning false otherwise. Notice that test() provides the same kind of parameter list as isConnected() and isPositiveBalance(). Also, notice that they all have the same return type (boolean).

import java.util.ArrayList;

import java.util.List;

import java.util.function.Predicate;

class Account

{

private int id, balance;

Account(int id, int balance)

{

this.balance = balance;

this.id = id;

}

int getBalance()

{

return balance;

}

int getID()

{

return id;

}

void print()

{

System.out.printf("Account: [%d], Balance: [%d]%n", id, balance);

}

}

public class LambdaPredicates

{

static List<Account> accounts;

public static void main(String[] args)

{

accounts = new ArrayList<>();

accounts.add(new Account(1000, 200));

accounts.add(new Account(2000, -500));

accounts.add(new Account(3000, 0));

accounts.add(new Account(4000, -80));

accounts.add(new Account(5000, 1000));

// Print all accounts

System.***out***.println("Print all accounts");

*printAccounts*(account -> **true**);

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

// Print all accounts with negative balances.

System.***out***.println("Print all accounts with negative balance");

*printAccounts*(account -> account.getBalance() < 0);

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

// Print all accounts whose id is greater than 2000 and less than 5000.

System.***out***.println("Print all accounts whose id is greater than 2000 and less than 5000");

*printAccounts*(account -> account.getID() > 2000 && account.getID() < 5000);

}

static void printAccounts(Predicate<Account> tester)

{

for (Account account: accounts)

if (tester.test(account))

account.print();

}

}

**Output:**

Print all accounts

Account: [1000], Balance: [200]

Account: [2000], Balance: [-500]

Account: [3000], Balance: [0]

Account: [4000], Balance: [-80]

Account: [5000], Balance: [1000]

Print all accounts with negative balance

Account: [2000], Balance: [-500]

Account: [4000], Balance: [-80]

Print all accounts whose id is greater than 2000 and less than 5000

Account: [3000], Balance: [0]

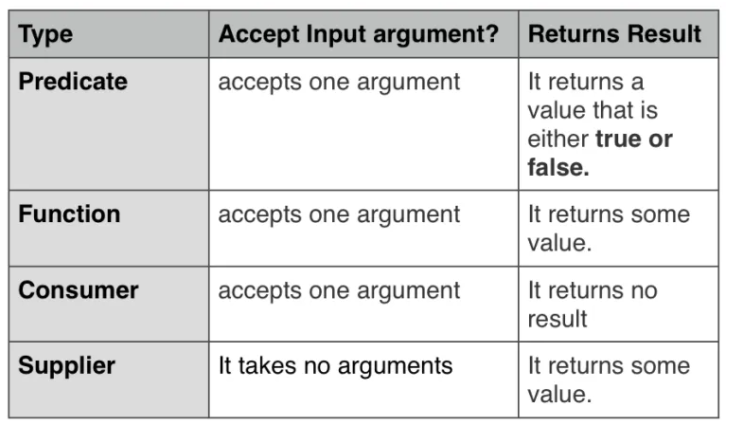
Account: [4000], Balance: [-80]

## **Functional Interfaces: Predicate, Consumer, Function, and Supplier**

* Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called functional method for that functional interface, to which the lambda expression’s parameter and return types are matched or adapted.
* There are around **40+ functional interfaces** under **[java.util.function](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html" \t "_blank)** package.In this blog, we will discuss the important ones: **Predicate, Consumer, Function, and Supplier.**

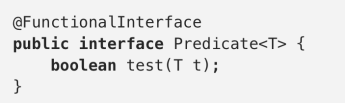
**Complete list of Function interface**

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>



#### **Predicate**

A predicate is a statement that may be true or false depending on the values of its variables. It can be thought of as a function that returns a value that is either true or false.



Predicate<String> predFunc = t -> t.length() > 5;

System.***out***.println("Is Programming String has length > 5 : "+predFunc.test("Programming"));

System.***out***.println("Is length of the ‘Dance’ > 5 :"+predFunc.test("Dance"));

// Predicate use to filter the collection

List<String> list1= **new** ArrayList<>();

list1.add("Pizza");

list1.add("Burger");

list1.add("oats");

list1.add("Almonds");

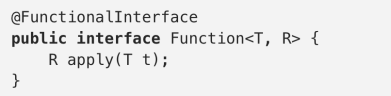
list1.add("Muli");

List<String> filteredString= list1.stream().filter(t->t.length()>=5).collect(Collectors.*toList*());

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

#### **Function**

This functional interface represents a function that accepts one argument and produces a result. One use, for example, it’s to convert or transform from one object to another. Since it’s a functional interface, you can pass a lambda expression wherever a Function is expected.  
The input parameter type and the return type of the method can either be same or different.



Function<Integer,Integer> funcInterface = t -> t\*2;

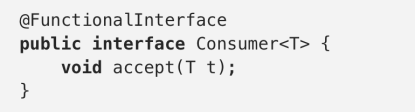
System.***out***.println(funcInterface.apply(5));

Function<Integer,String> funcInterface1 = t -> t.toString();

System.***out***.println(funcInterface1.apply(5));

#### **Consumer**

This functional interface represents an operation that accepts a single input argument and returns no result. The real outcome is the side-effects it produces. Since it’s a functional interface, you can pass a lambda expression wherever a Consumer is expected.



**class** Emp{

**private** String name;

**public** Emp(String s) {

**this**.name=s;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

Emp emp=**new** Emp("Radha");

Consumer<Emp> updateName= s->s.setName("Radhe");

updateName.accept(emp);

System.***out***.println("Consumer update name :"+emp.getName());

#### **Supplier**

This functional interface does the opposite of the Consumer, it takes no arguments but it returns some value. It may return different values when it is being called more than once. Since it’s a functional interface, you can pass a lambda expression wherever a Supplier is expected.



Supplier<Integer> supplierFunc = () -> 20;

System.***out***.println(supplierFunc.get());

Complete program:

**import** java.util.ArrayList;

**import** java.util.List;

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

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**import** java.util.function.Supplier;

**import** java.util.stream.Collectors;

**public** **class** FunctionalInterfacesDemo {

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

Predicate<String> predFunc = t -> t.length() > 5;

System.***out***.println("Is Programming > 5 : "+predFunc.test("Programming"));

System.***out***.println("Is Dance > 5 :"+predFunc.test("Dance"));

// Predicate use to filter the collection

List<String> list1= **new** ArrayList<>();

list1.add("Pizza");

list1.add("Burger");

list1.add("oats");

list1.add("Almonds");

list1.add("Muli");

List<String> filteredString= list1.stream().filter(t->t.length()>=5).collect(Collectors.*toList*());

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

// Function :This functional interface represents a function that accepts one argument

// and produces a result. One use, for example, it’s to convert or transform from one object to another.

Function<Integer,Integer> funcInterface = t -> t\*2;

System.***out***.println("Function interface with return t\*2 in integer :"+funcInterface.apply(5));

Function<Integer,String> funcInterface1 = t -> t.toString();

System.***out***.println("Function interface with return String :"+funcInterface1.apply(5));

/\* Consumer : This functional interface represents an operation that accepts a

\* single input argument and returns no result.

\* Since it’s a functional interface, you can pass a lambda expression wherever a Consumer

\* is expected.

\*/

**class** Emp{

**private** String name;

**public** Emp(String s) {

**this**.name=s;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

Emp emp=**new** Emp("Radha");

Consumer<Emp> updateName= s->s.setName("Radhe");

updateName.accept(emp);

System.***out***.println("Consumer update name :"+emp.getName());

/\*Supplier : This functional interface does the opposite of the Consumer,

\* it takes no arguments but it returns some value.

\* It may return different values when it is being called more than once.

\* Since it’s a functional interface, you can pass a lambda expression wherever a Supplier is expected.

\*/

Supplier<Integer> supplierFunc = () -> 20;

System.***out***.println("Supplier value :"+supplierFunc.get());

}

}

Output :

Is Programming > 5: true

Is Dance > 5: false

[Pizza, Burger, Almonds]

Function interface with return t\*2 in integer :10

Function interface with return String :5

Consumer update name :Radhe

Supplier value :20

# **Java 8 method references, double colon (::) operator**

In Java 8, the double colon (::) operator is called method references.

Anonymous class to print a list.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(new Consumer<String>() { // anonymous class

@Override

public void accept(String str) {

System.out.println(str);

}

});

Anonymous class -> Lambda expressions.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(str -> System.out.println(str)); *// lambda*

Lambda expressions -> Method references.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(System.out::println); *// method references , out is PrintStream class*

There are four kinds of method references:

* Reference to a static method ClassName::staticMethodName
* Reference to an instance method of a particular object Object::instanceMethodName
* Reference to an instance method of an arbitrary object of a particular type ContainingType::methodName–
* Reference to a constructor ClassName::new

## **1. Static method**

Lambda expression.

(args) -> ClassName.staticMethodName(args)

Method Reference.

ClassName::staticMethodName

1.1 This example prints a list of Strings, method reference to a static method SimplePrinter::print.

**import** java.util.Arrays;

**import** java.util.List;

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

**public** **class** MethodReferenceStatic {

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

List<String> list = Arrays.*asList*("A", "B", "C");

// anonymous class

list.forEach(**new** Consumer<String>() {

@Override

**public** **void** accept(String x) {

SimplePrinter.*print*(x);

}

});

// lambda expression

list.forEach(x -> SimplePrinter.*print*(x));

// method reference

list.forEach(SimplePrinter::*print*);

}

}

**class** SimplePrinter {

**public** **static** **void** print(String str) {

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

}

}

1.2 This example converts a list of Strings into a list of Integers, method reference to a static method Integer::parseInt.

**Integer.java**

**public static int parseInt(String s) throws NumberFormatException {**

**return parseInt(s,10);**

**}**

**MethodReferenceStatic1.java**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Function;

**import** java.util.stream.Collectors;

**public** **class** MethodReferenceStatic1 {

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

List<String> list = Arrays.*asList*("1", "2", "3");

// anonymous class

List<Integer> collect1 = list.stream()

.map(**new** Function<String, Integer>() {

@Override

**public** Integer apply(String s) {

**return** Integer.*parseInt*(s);

}

})

.collect(Collectors.*toList*());

// lambda expression

List<Integer> collect2 = list.stream()

.map(s -> Integer.*parseInt*(s))

.collect(Collectors.*toList*());

// method reference

List<Integer> collect3 = list.stream()

.map(Integer::*parseInt*)

.collect(Collectors.*toList*());

System.***out***.println("Output using anonymous class :"+collect1);

System.***out***.println("Output using Lambda :"+collect2);

System.***out***.println("Output using method reference :"+collect3);

}

}

**2. Reference to an instance method of a particular object**

Lambda expression.

(args) -> object.instanceMethodName(args)

Method Reference.

object::instanceMethodName

**2.1 This example sorts a list of Employee by salary. We can reference to an instance method compareBySalary of a particular object ComparatorProvider.**

**import** java.math.BigDecimal;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** Java8MethodReference2 {

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

List<Employee> list = Arrays.*asList*(

**new** Employee("Kunal", 38, BigDecimal.*valueOf*(38000)),

**new** Employee("Shyam", 5, BigDecimal.*valueOf*(1000)),

**new** Employee("Riya", 25, BigDecimal.*valueOf*(25000)),

**new** Employee("Saloni", 99, BigDecimal.*valueOf*(99999)));

ComparatorProvider provider = **new** ComparatorProvider();

// anonymous class

/\*list.sort(new Comparator<Employee>() {

@Override

public int compare(Employee o1, Employee o2) {

return provider.compareBySalary(o1, o2);

}

});\*/

// lambda

// list.sort((o1, o2) -> provider.compareBySalary(o1, o2));

// method reference

list.sort(provider::compareBySalary);

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

}

}

**class** ComparatorProvider {

**public** **int** compareByAge(Employee o1, Employee o2) {

**return** o1.getAge().compareTo(o2.getAge());

}

**public** **int** compareByName(Employee o1, Employee o2) {

**return** o1.getName().compareTo(o2.getName());

}

**public** **int** compareBySalary(Employee o1, Employee o2) {

**return** o1.getSalary().compareTo(o2. getSalary ());

}

}

**import** java.math.BigDecimal;

**public** **class** Employee {

String name;

Integer age;

BigDecimal salary;

// getters, setters, constructor, toString

}

## 3. Reference to an instance method of an arbitrary object of a particular type.

Lambda expression.

*// arg0 is the first argument*

(arg0, rest\_of\_args) -> arg0.methodName(rest\_of\_args)

*// example, assume a and b are String*

(a, b) -> a.compareToIgnoreCase(b)

Method Reference.

*// first argument type*

arg0\_Type::methodName

*// arg0 is type of ClassName*

ClassName::methodName

*// example, a is type of String*

String::compareToIgnoreCase

To understand this better lets first learn BiFunction Interface

In Java 8, *[BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)* is a functional interface; it takes two arguments and returns an object.

**BiFunction.java**

**@FunctionalInterface**

**public interface BiFunction<T, U, R> {**

**R apply(T t, U u);**

**}**

* T – Type of the first argument to the function.
* U – Type of the second argument to the function.
* R – Type of the result of the function.

## **BiFunction<T, U, R>**

This example takes two Integers and returns an Integer, Double or List

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.BiFunction;

**public** **class** Java8BiFunction1 {

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

// takes two Integers and return an Integer

BiFunction<Integer, Integer, Integer> func = (x1, x2) -> x1 + x2;

Integer result = func.apply(2, 3);

System.***out***.println(result); // 5

// take two Integers and return an Double

BiFunction<Integer, Integer, Double> func2 = (x1, x2) -> Math.*pow*(x1, x2);

Double result2 = func2.apply(2, 4);

System.***out***.println(result2); // 16.0

// take two Integers and return a List<Integer>

BiFunction<Integer, Integer, List<Integer>> func3 = (x1, x2) -> Arrays.*asList*(x1 + x2);

List<Integer> result3 = func3.apply(2, 3);

System.***out***.println(result3);

}

}

## **Example : Reference to an instance method of an arbitrary object of a particular type.**

**import** java.util.function.BiPredicate;

**import** java.util.function.Function;

**public** **class** Java8MethodReference3a {

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

// lambda

**int** result = *playOneArgument*("Python", x -> x.length()); // 6

// method reference

**int** result2 = *playOneArgument*("Python", String::length); // 6

// lambda

Boolean result3 = *playTwoArgument*("Python ", "y", (a, b) -> a.contains(b)); // true

// method reference

Boolean result4 = *playTwoArgument*("Python", "y", String::contains); // true

// lambda

Boolean result5 = *playTwoArgument*("Python", "y", (a, b) -> a.startsWith(b)); // false

// method reference

Boolean result6 = *playTwoArgument*("Python", "y", String::startsWith); // false

System.***out***.println(result6);

}

**static** <R> R playOneArgument(String s1, Function<String, R> func) {

**return** func.apply(s1);

}

**static** Boolean playTwoArgument(String s1, String s2, BiPredicate<String, String> func) {

**return** func.test(s1, s2);

}

}

## **4. Reference to a constructor.**

Lambda expression.

(args) -> new ClassName(args)

Method Reference.

ClassName::new

**4.1 Reference to a default constructor.**

**import** java.math.BigDecimal;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.function.Supplier;

**public** **class** MethodReferenceConstrcutorDemo {

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

// lambda

Supplier<Map> obj1 = () -> **new** HashMap(); // default HashMap() constructor

Map map1 = obj1.get();

// method reference

Supplier<Map> obj2 = HashMap::**new**;

Map map2 = obj2.get();

// lambda

Supplier<Invoice> obj3 = () -> **new** Invoice(); // default Invoice() constructor

Invoice invoice1 = obj3.get();

// method reference

Supplier<Invoice> obj4 = Invoice::**new**;

Invoice invoice2 = obj4.get();

}

}

**class** Invoice {

String no;

BigDecimal unitPrice;

Integer qty;

**public** Invoice() {

}

**public** String getNo() {

**return** no;

}

**public** **void** setNo(String no) {

**this**.no = no;

}

**public** BigDecimal getUnitPrice() {

**return** unitPrice;

}

**public** **void** setUnitPrice(BigDecimal unitPrice) {

**this**.unitPrice = unitPrice;

}

**public** Integer getQty() {

**return** qty;

}

**public** **void** setQty(Integer qty) {

**this**.qty = qty;

}

}

**4.2 Reference to a constructor which accepts an argument – Invoice(BigDecimal unitPrice)**

**import** java.math.BigDecimal;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Function;

**public** **class** MethodReferenceConstructorDemo2 {

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

List<BigDecimal> list = Arrays.*asList*(

BigDecimal.*valueOf*(9.99),

BigDecimal.*valueOf*(2.99),

BigDecimal.*valueOf*(8.99));

// lambda

// List<Invoice> invoices = fakeInvoice(list, (unit) -> new Invoice1(unit));

// method reference

List<Invoice1> invoices = *fakeInvoice*(list, Invoice1::**new**);

invoices.forEach(System.***out***::println);

}

**static** List<Invoice1> fakeInvoice(List<BigDecimal> list, Function<BigDecimal, Invoice1> func) {

List<Invoice1> result = **new** ArrayList<>();

**for** (BigDecimal unit : list) {

result.add(func.apply(unit));

}

**return** result;

}

}

**class** Invoice1 {

String no;

BigDecimal unitPrice;

Integer qty;

**public** Invoice1() {

}

**public** Invoice1(BigDecimal bd) {

**this**.unitPrice = bd;

}

**public** String getNo() {

**return** no;

}

**public** **void** setNo(String no) {

**this**.no = no;

}

**public** BigDecimal getUnitPrice() {

**return** unitPrice;

}

**public** **void** setUnitPrice(BigDecimal unitPrice) {

**this**.unitPrice = unitPrice;

}

**public** Integer getQty() {

**return** qty;

}

**public** **void** setQty(Integer qty) {

**this**.qty = qty;

}

**public** String toString() {

**return** **this**.getUnitPrice().toString();

}

}

Output

9.99

2.99

8.99