

Java Functional Interface

05506004

<https://www.scaler.com/topics/functional-interface-in-java/>

Functional Interface in Java

- Is also called **Single Abstract Method (SAM)** interface.
 - A functional interface can contain **only one abstract method** and it **can contain** any number of static and default (non-abstract) methods.
- Enables users to implement functional programming in Java. In functional programming, the function is an **independent entity**.
 - No function is independently present on its own in java. They are part of classes or interfaces. And to use them we require either the class or the object of the respective class to call that function.

```
// interface
@FunctionalInterface // annotation
interface interfaceName{
    // abstract method
    abstract returnType methodName( /* parameters */);

    // default or static methods

    int method1(){
        // ....
    }
    String method2(int x, float y){
        // ....
    }
}

// public class
public class className{
    // main method
    public static void main(String[] args){
        interfaceName temp = (/*parameters*/) -> {
            // perform operations
        };

        temp.methodName(); // call abstract method of the interface
    }
}
```

Ways to Use Functional Interfaces in a Class

- concrete class
- Anonymous class \leftrightarrow lambda expression

```
import java.util.*;
@FunctionalInterface
interface PersonalGreet{
    String greeting(String name);
}
public class MyClass implements PersonalGreet{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("May I please know your Name?");
        String name = sc.next();
        MyClass obj = new MyClass();
        System.out.println(obj.greeting(name));
    }
    @Override
    public String greeting(String name){
        return "Hello! "+name;
    }
}
```

```
import java.util.*;
@FunctionalInterface
interface PersonalGreet{
    String greeting(String name);
}
public class MyClass {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("May I please know your Name?");
        String name = sc.next();
        PersonalGreet hello = (String temp) -> "Hello! "+temp;
        System.out.println(hello.greeting(name));
    }
}
```

More on Functional Interface

- A Functional Interface Can Have Methods of Object Class

```
@FunctionalInterface
interface ObjectClassMethods{
    // abstract method
    int abstractMethod(int val);
    // methods of the object class
    int hashCode();
    String toString();
    boolean equals(Object obj);
}
```

- Functional Interface Having Multiple Default and Static Classes
- Java has pre-defined or built-in functional interfaces for commonly occurring cases.
 - A few of these interfaces are Runnable, Comparable, ActionListener, Callable

```
// interface implementation
@FunctionalInterface
interface StaticandDefaultMethods{
    // abstract method
    int square(int x);
    // default methods
    default int add(int a, int b){
        return a+b;
    }
    default int sub(int a, int b){
        return a-b;
    }
    // static methods
    static int multiply(int a, int b){
        return a*b;
    }
    static int divide(int a, int b){
        return a/b;
    }
}

// public class
public class Test implements StaticandDefaultMethods{
    public static void main(String[] args){
```

Built-in Java Functional Interfaces

- Functional interfaces in Java are mainly of four types:
 - Consumer
 - Predicate
 - Function
 - Supplier

Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

Consumer

- The Consumer functional interface in Java accepts a **single** gentrified **argument** and **doesn't return any value**.
- `void accept(T t);`


```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
List<String> names = Arrays.asList("John", "Jane", "Bob");

Consumer<String> printName = name -> System.out.println(name);

names.forEach(printName);
```

```
static void demo1() {
    List<String> names = Arrays.asList(...a: "aba", "abi", "abo", "abe");
    names.forE
```

 `forEach(Consumer<? super String> action)` `void`

Predicate

- The predicate functional interface in Java takes a **single argument** and **returns a boolean value**.
- `boolean test(T t);`

```
public interface Predicate<T>{  
    boolean test(T t);  
}
```

```
Predicate<Integer> isEven = num -> num % 2 == 0;  
  
boolean result1 = isEven.test(4); // returns true  
boolean result2 = isEven.test(7); // returns false  
  
System.out.println("Result1: " + result1);  
System.out.println("Result2: " + result2);
```

```
Predicate<String> isLongerThan5 = str -> str.length() > 5;  
  
boolean result3 = isLongerThan5.test("hello"); // returns false  
boolean result4 = isLongerThan5.test("goodbye"); // returns true  
  
System.out.println("Result3: " + result3);  
System.out.println("Result4: " + result4);
```

Function

- The function type functional interface receives a **single argument**, processes it, and **returns a value**.
- R `apply(T t);`
- (Remark
BiFunction, BinaryOperator)
 - Next slide

```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T t);
}
```

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

int length = stringLength.apply("hello");

System.out.println("Length of string: " + length);
```

```
Function<String, Integer> stringToInt = str -> Integer.parseInt(str);

int num = stringToInt.apply("42");

System.out.println("Number: " + num);
```


(More) Function

- BiFunction, BinaryOperator

```
import java.util.function.BiFunction;

public class Main{
    public static void main(String[] args) {
        // This implementation concatenates the argument strings passed as parameters
        BiFunction<String, String, String> concatStrings = (s, s2) -> s.concat(s2);
        String s1 = "hello";
        String s2 = "-educative";

        // calling apply method of the BiFunction
        System.out.println(concatStrings.apply(s1, s2));
    }
}
```

```
import java.util.function.Function;
import java.util.function.BinaryOperator;

public class Main{
    public static void main(String args[]){
        BinaryOperator<Integer> and = (a,b) -> a & b;
        System.out.println(and.apply(12, 4));
    }
}
```

Supplier

- It **doesn't take any arguments**. On calling the supplier it simply **returns a value**.
 - Supplier is a generic interface thus, it takes the type of value in <> (Angular brackets) while implementing to be returned by the get() method.
- T **get()**;

```
@FunctionalInterface
public interface Supplier<T>{
    T get();
}
```

```
Supplier<Integer> randomNumberSupplier = () -> (int) (Math.random() * 100);

int randomNumber = randomNumberSupplier.get();

System.out.println("Random number: " + randomNumber);
```

More Examples

```
class Product {  
    private double price = 0.0;  
  
    public void setPrice(double price) {  
        this.price = price;  
    }  
  
    public void printPrice() {  
        System.out.println(price);  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Consumer<Product> updatePrice = p -> p.setPrice(5.9);  
        Product p = new Product();  
        updatePrice.accept(p);  
        p.printPrice();  
    }  
}
```

```
Predicate<String> isALongWord = t -> t.length() > 10;  
String s = "successfully"  
boolean result = isALongWord.test(s);
```

```
Predicate<String> isALongWord = new Predicate<String>() {  
    @Override  
    public boolean test(String t) {  
        return t.length() > 10;  
    }  
};  
String s = "successfully"  
boolean result = isALongWord.test(s);
```

More Examples

```
public class Test {  
    public static void main(String[] args) {  
        int n = 5;  
        modifyTheValue(n, val-> val + 10);  
        modifyTheValue(n, val-> val * 100);  
    }  
  
    static void modifyValue(int v, Function<Integer, Integer> function){  
        int result = function.apply(v);  
        System.out.println(newValue);  
    }  
}
```

```
public class Program {  
    public static void main(String[] args) {  
        int n = 3;  
        display(() -> n + 10);  
        display(() -> n + 100);  
    }  
  
    static void display(Supplier<Integer> arg) {  
        System.out.println(arg.get());  
    }  
}
```

Method References in Java 8

05506004

Outlines

- Introduction
- Four types of method references
 1. Method reference to a static method of a class
 2. Method reference to an instance method of an object
 3. Method reference to an instance method of an arbitrary object of a particular type
 4. Method reference to a constructor

Nr	Method Reference Type	Method Reference	Lambda expression
1	Static method	<code>String::valueOf</code>	<code>(int i) -> String.valueOf(i)</code>
2	Instance method of a particular object	<code>s::substring</code>	<code>(int beg, int end) -> s.substring(beg, end)</code>
3	Instance method of an arbitrary object	<code>String::equals</code>	<code>(String s1, String s2) -> s1.equals(s2)</code>
		<code>JLabel::getIcon</code>	<code>(JLabel lb) -> lb.getIcon()</code>
4	Constructor	<code>String::new</code>	<code>() -> new String()</code>

<https://stackoverflow.com/questions/23023618/how-to-invoke-parameterized-method-with-method-reference>

There are 4 kinds of method references in Java:

1. `ContainingClass::staticMethodName` - reference to a static method
2. `containingObject::instanceMethodName` - reference to an instance method of a particular object
3. `ContainingType::methodName` - reference to an instance method of an arbitrary object of a particular type
4. `ClassName::new` - reference to a constructor

<https://stackoverflow.com/questions/66930573/method-reference-for-static-and-instance-methods>

Recap anonymous class and lambda expression on Functional Interface

```
interface DoubleMe {  
    public void timesTwo(int n);  
}
```

```
static void main() {  
    // anonymous class  
    DoubleMe obj1 = new DoubleMe() {  
        public void timesTwo(int n) {  
            System.out.println(n * 2);  
        }  
    };  
    System.out.println( obj1.timesTwo(6) );  
  
    // lambda expression  
    DoubleMe obj2 = n -> System.out.println(n * 2);  
    System.out.println( obj2.timesTwo(6) );  
}
```

Introduction

- **method reference** can be defined as some special form of the lambda expression.
 - It helps reduce the code of the program and aids in supporting the newly developed functionality of the programming language. It is used for **referring to a method** that serves to be functionally important.
- The following points can be marked while understanding the concept:
 - The method is referred to through the use of “::”.
 - The arguments for the method type are inferred upon by JRE at the runtime **through the context** it is defined.

If your lambda expression is like this:

```
str -> System.out.println(str)
```

then you can replace it with a method reference like this:

```
System.out::println
```


Person.java, WaysToReferenceClassMethod.java

- Sending the context as a parameter to Class's static method (Method reference to **a static method of a class**)
- `ContainingClass::staticMethodName` - reference to a static method

```
import java.util.Arrays;
import java.util.function.Consumer;
public class MRDemo
{
    public static void main(String[] args)
    {
        int[] array = { 10, 2, 19, 5, 17 };
        Consumer<int[]> consumer = Arrays::sort;
        consumer.accept(array);
        for (int i = 0; i < array.length; i++)
            System.out.println(array[i]);
        System.out.println();
        int[] array2 = { 19, 5, 14, 3, 21, 4 };
        Consumer<int[]> consumer2 = (a) -> Arrays.sort(a);
        consumer2.accept(array2);
        for (int i = 0; i < array2.length; i++)
            System.out.println(array2[i]);
    }
}
```

<https://www.infoworld.com/article/3453296/get-started-with-method-references-in-java.html>

Person.java, WaysToReferenceClassMethod.java

- variable `s` invokes the `print()` class method with functionality to obtain this string's length as this method's argument. (Method reference to **an instance method of an object**)
- `ContainingObject::instanceMethodName` - reference to an instance method of a particular object

```
import java.util.function.Supplier;
public class MRDemo
{
    public static void main(String[] args)
    {
        String s = "The quick brown fox jumped over the lazy dog";
        print(s::length);
        print(() -> s.length());
        print(new Supplier<Integer>()
        {
            @Override
            public Integer get()
            {
                return s.length(); // closes over s
            }
        });
    }
    public static void print(Supplier<Integer> supplier)
    {
        System.out.println(supplier.get());
    }
}
```

<https://www.infoworld.com/article/3453296/get-started-with-method-references-in-java.html>

Person.java, WaysToReferenceClassMethod.java

- the print() method with functionality to convert a string to lowercase and the string to be converted as the method's arguments. Method reference to **an instance method of an arbitrary object of a particular type**)
- `ContainingType::methodName` - reference to an instance method of an arbitrary object of a particular type

```
import java.util.function.Function;
public class MRDemo

    public static void main(String[] args)
    {
        print(String::toLowerCase, "STRING TO LOWERCASE");
        print(s -> s.toLowerCase(), "STRING TO LOWERCASE");
        print(new Function<String, String>()
        {
            @Override
            public String apply(String s) // receives argument in parameter
            {                               // doesn't need to close over s
                return s.toLowerCase();
            }
        }, "STRING TO LOWERCASE");
    }

    public static void print(Function<String, String> function, String s)
    {
        System.out.println(function.apply(s));
    }
}
```

<https://www.infoworld.com/article/3453296/get-started-with-method-references-in-java.html>

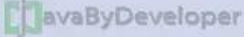

- You can use a method reference to refer to a constructor without instantiating the named class. This kind of method reference is known as a constructor reference. (Method reference to [a constructor](#))
- `ClassName::new` - reference to a constructor

```
//https://www.baeldung.com/java-method-  
references  
//constructor  
public Bicycle(String brand) {  
    this.brand = brand;  
    this.frameSize = 0;  
}  
...  
brands.stream().map(Bicycle::new).toList();
```

```
public class Person {
    ...
    public Person(String n, Integer a,
                  String d, int num) {
        name = n; age = a;
        favoriteDrink = d;
        lottoMotto = num;
    }

    public String getName() { ... }
    public Integer getAge() { ... }
    public String favoriteDrink() {
        return favoriteDrink; }
    public Integer getLotto() {
        return lottoMotto; }
    public void setLotto(Integer num) {
        lottoMotto = num; }

    @Override
    public String toString() {
        return "Person(" + name + ", "
            + age + ", " + favoriteDrink
            + ", " + lottoMotto + ")"; }
}
```

Method Reference Type	Syntax	Example
Reference to static method	<code><target class name>::<code>static method name</code></code>	<code>Integer::parseInt</code> 
Reference to an instance method	<code><target class object>::<code>instance method</code></code>	<code>System.out::println</code>
Reference to an instance method of an arbitrary object of particular type	<code><arbitrary object type>::<code>instance method</code></code>	<code>String::toUpperCase</code>
Constructor reference	<code><Class Name>::<code>new</code></code> 	<code>String::new</code> ; <code>ArrayList::new</code> ;

<https://javabydeveloper.com/method-reference-java-8/>

```
public class WaysToReferenceClassMethod {
    public static String staticMethodCS(String str) {
        return str + " is a CS student.";
    }
    public static String staticMethodKMITL(String str) {
        return str + "is a kmitl student";
    }
    public String capitalizeName(Person p) {
        return p.getName().substring(0, 1)
            .toUpperCase() + p.getName().substring(1);
    }
}
```

Method reference to a static method of a class

```
public static void q1_demo_reference_to_a_static_method() {  
    List<String> tmp = new ArrayList<>();  
    for (Person p : personList)  
        tmp.add(p.getName());  
    List<String> ans1 = tmp  
        .stream().map(WaysToReferenceClassMethod::staticMethodCS)  
        .collect(Collectors.toList());  
    System.out.println(ans1);  
    // [yindee is a CS student., preeda is a CS student., pramote is a CS student.]  
  
    List<String> ans2 = personList.stream()  
        .map(Person::getName)/* refer arbitrary obj method */.toList()  
        .stream().map(WaysToReferenceClassMethod::staticMethodKMITL)  
        .collect(Collectors.toList());  
    System.out.println(ans2);  
    // [yindeeis a kmitl student, preedais a kmitl student, pramoteis a kmitl  
    // student]  
}
```

Method reference to an instance method of an object

```
public static void q2_demo_reference_to_an_instance_method() {  
    WaysToReferenceClassMethod  
        waysMethodRef = new WaysToReferenceClassMethod();  
    List<String> ans = personList.stream()  
        .map(waysMethodRef::capitalizeName).toList();  
    System.out.println(ans);  
    // [Yindee, Preeda, Pramote]  
}
```

Type	Example	Syntax
Reference to a Static Method	ContainingClass::staticMethodName	Class::staticMethodName
Reference to an Instance Method of a Particular Object	containingObject::instanceMethodName	object::instanceMethodName
Reference to an Instance Method of an Arbitrary Object of a Particular Type	ContainingType::methodName	Class::instanceMethodName
Reference to a Constructor	ClassName::new	ClassName::new

Method reference to an instance method of an arbitrary object of a particular type

```
public static void q3_1_demo_reference_to_an_arbitrary_obj() {  
    Comparator<Person> byLotto =  
        Comparator.comparing(Person::getLotto);  
    Collections.sort(personList, byLotto);  
    personList.forEach(System.out::println);  
    // Person(pramote, 19, cappuccino, 39)  
    // Person(yindee, 24, Latte, 73)  
    // Person(preeda, 25, Espresso, 82)  
  
    Collections.sort(personList,  
        (p, q) -> p.getAge().compareTo(q.getAge()));  
    // (p, q) -> Integer.compare(p.getAge(), q.getAge())  
    personList.forEach(System.out::println);  
    // Person(pramote, 19, cappuccino, 39)  
    // Person(yindee, 24, Latte, 73)  
    // Person(preeda, 25, Espresso, 82)  
}
```

Method Reference In Java: Java 8 New Feature	
Method Reference vs Lambda Expression	
Method Reference	Lambda Expression
String :: toString	s -> s.toString()
String :: toLowerCase	s -> s.toLowerCase()
String :: length	s -> s.length()
Integer :: compareTo	(i1,i2) -> i1.compareTo(i2)
String :: compareTo	(s1,s2) -> s1.compareTo(s2)

<https://javatechonline.com/method-reference-java-8/>

Method reference to an instance method of an arbitrary object of a particular type

```
public static void q3_2_demo_reference_to_an_arbitrary_obj() {  
    List<String> names = Arrays.asList("YINDEE", "PREEDA", "PRAMOTE");  
    List<String> ans = names.stream()  
        .map(element -> element.toLowerCase()).toList();  
    System.out.println("original " + names);  
    System.out.println(ans);  
  
    ans = names.stream()  
        .map(String::toLowerCase).toList();  
    System.out.println(ans);  
  
    ans = personList.stream()  
        .map(p -> p.getName()).toList();  
    System.out.println(ans);  
  
    ans = personList.stream()  
        .map(Person::getName).toList();  
    System.out.println(ans);  
    System.out.println("original " + names);  
}
```

Type	Description	Usage	Example
Static Method Reference	Use class name to refer the static method of the class	ClassName::StaticMethodName	Math::floor
Instance Method Reference	Use the object to refer the instance method of the class	Object::InstanceMethodName	System.out::println
Arbitrary object - Instance Method Reference	Use class name to refer the instance method of the class	ClassName::InstanceMethodName	ArrayList::size
Constructor Reference	Use class name to refer the constructor of the class	ClassName::new	ArrayList::new

<https://www.javainterviewpoint.com/java-method-reference/>

Method reference to a constructor

```
public static void q4_reference_to_constructor() {  
    List<String> singerNames =  
        Arrays.asList("Aba", "Abi", "Abo", "Abe");  
    List<Singer> singers = singerNames.stream()  
        .map(Singer::new).toList();  
    singers.forEach(Singer::sing);  
}
```

Lambda Expressions	Equivalent Method References
(String s) -> Integer.parseInt(s)	Integer::parseInt
(String s) -> s.toLowerCase()	String::toLowerCase
(int i) -> System.out.println(i)	System.out::println
(Student s) -> s.getName()	Student::getName
() -> s.getName()	s::getName where 's' refers to <i>Student</i> object which already exist.
() -> new Student()	Student::new

<https://javaconceptoftheday.com/java-8-method-references/>

Further Readings

- <https://www.geeksforgeeks.org/method-references-in-java-with-examples/>
- <https://examples.javacodegeeks.com/java-development/core-java/java-8-method-reference-example/>