**Lambda Expression**

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface **typically represents a single action**. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run( )**. Therefore, **run( )** defines the action of **Runnable**.

**Lambda Expression Fundamentals**

The lambda expression introduced a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is **−>**. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body,* which specifies the actions of the lambda expression. The **−>** can be verbalized as “becomes” or “goes to.”

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. *We will begin with lambdas that define a single expression*. Example,

It evaluates to a constant value and is shown here:

() -> 123.45

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

double myMeth() { return 123.45; }

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

() -> Math.random() \* 100

This lambda expression obtains a pseudo-random value from **Math.random( )**, multiplies it by 100, and returns the result.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

(n) -> (n % 2)==0

Functional Interface:

Here is an example of a functional interface:

interface MyNumber

{

double getValue();

}

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let’s work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

// Create a reference to a MyNumber instance.

MyNumber myNum;

Next, a lambda expression is assigned to that interface reference:

// Use a lambda in an assignment context.

myNum = () -> 123.45;

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the **getValue( )** method. As a result, the following displays the value 123.45:

// Call getValue(), which is implemented by the previously assigned lambda expression. System.out.println(myNum.getValue());

package com.skg.lambda.demo;

//Demonstrate a simple lambda expression.

//A functional interface.

interface MyNumber {

double getValue();

}

class LambdaDemo {

public static void main(String[] args) {

MyNumber myNum; // declare an interface reference

//Here, the lambda expression is simply a constant expression.

//When it is assigned to myNum, a class instance is

//constructed in which the lambda expression implements

//the getValue() method in MyNumber.

myNum = () -> 123.45;

//Call getValue(), which is provided by the previously assigned

//lambda expression.

System.out.println("A fixed value: " + myNum.getValue());

//Here, a more complex expression is used.

myNum = () -> Math.random() \* 100;

//These call the lambda expression in the previous line.

System.out.println("A random value: " + myNum.getValue());

System.out.println("Another random value: " + myNum.getValue());

//A lambda expression must be compatible with the method

//defined by the functional interface. Therefore, this won't work:

//myNum = () -> "123.03"; // Error!

}

}

The lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue( )**.

//Demonstrate a lambda expression that takes a parameter.

//Another functional interface.

interface NumericTest {

boolean test(int n);

}

class LambdaDemo2 {

public static void main(String[] args) {

//A lambda expression that tests if a number is even.

NumericTest isEven = (n) -> (n % 2) == 0;

if (isEven.test(10))

System.out.println("10 is even");

if (!isEven.test(9))

System.out.println("9 is not even");

//Now, use a lambda expression that tests if a number

//is non-negative.

NumericTest isNonNeg = (n) -> n >= 0;

if (isNonNeg.test(1))

System.out.println("1 is non-negative");

if (!isNonNeg.test(-1))

System.out.println("-1 is negative");

}

}

The output from this program is shown here:

10 is even

9 is not even

1 is non-negative

-1 is negative

Block Lambda Expression:

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies,* and lambdas that have expression bodies are sometimes called *expression lambdas.* In an expression body, the code on the right side of the lambda operator must consist of a single expression. While expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body.* Lambdas that have block bodies are sometimes referred to as *block lambdas*.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

//A block lambda that computes the factorial of an int value.

interface NumericFunc {

int func(int n);

}

class BlockLambdaDemo {

public static void main(String[] args) {

//This block lambda computes the factorial of an int value.

NumericFunc factorial = (n) -> {

int result = 1;

for (int i = 1; i <= n; i++)

result = i \* result;

return result;

};

System.out.println("The factoral of 3 is " + factorial.func(3));

System.out.println("The factoral of 5 is " + factorial.func(5));

}

}

The output is shown here:

The factorial of 3 is 6 The factorial of 5 is 120

//A block lambda that reverses the characters in a string.

interface StringFunc {

String func(String n);

}

class BlockLambdaDemo2 {

public static void main(String[] args) {

//This block lambda reverses the characters in a string.

StringFunc reverse = (str) -> {

String result = "";

int i;

for (i = str.length() - 1; i >= 0; i--)

result += str.charAt(i);

return result;

};

System.out.println("Lambda reversed is " + reverse.func("Lambda"));

System.out.println("Expression reversed is " + reverse.func("Expression"));

}

}

The output

is shown here:

Lambda reversed

is adbmaL

Expression reversed

is noisserpxE

Generic Functional Interface

However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

//Use a generic functional interface with lambda expressions.

//A generic functional interface.

interface SomeFunc<T> {

T func(T t);

}

class GenericFunctionalInterfaceDemo {

public static void main(String[] args) {

//Use a String-based version of SomeFunc.

SomeFunc<String> reverse = (str) -> {

String result = "";

int i;

for (i = str.length() - 1; i >= 0; i--)

result += str.charAt(i);

return result;

};

System.out.println("Lambda reversed is " + reverse.func("Lambda"));

System.out.println("Expression reversed is " + reverse.func("Expression"));

//Now, use an Integer-based version of SomeFunc.

SomeFunc<Integer> factorial = (n) -> {

int result = 1;

for (int i = 1; i <= n; i++)

result = i \* result;

return result;

};

System.out.println("The factoral of 3 is " + factorial.func(3));

System.out.println("The factoral of 5 is " + factorial.func(5));

}

}

The output is shown here:

Lambda reversed is adbmaL

Expression reversed is noisserpxE

The factoral of 3 is 6

The factoral of 5 is 120

Here, **T** specifies both the return type and the parameter type of **func( )**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

Passing Lambda Expression as Argument

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda.

//Use lambda expressions as an argument to a method.

**interface** StringFunc {

String func(String n);

}

**class** LambdasAsArgumentsDemo {

//This method has a functional interface as the type of

//its first parameter. Thus, it can be passed a reference to

//any instance of that interface, including the instance created

//by a lambda expression.

//The second parameter specifies the string to operate on.

**static** String stringOp(StringFunc sf, String s) {

**return** sf.func(s);

}

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

String inStr = "Lambdas add power to Java";

String outStr;

System.***out***.println("Here is input string: " + inStr);

// Here, a simple expression lambda that uppercases a string

// is passed to stringOp( ).

outStr = *stringOp*((str) -> str.toUpperCase(), inStr);

System.***out***.println("The string in uppercase: " + outStr);

// This passes a block lambda that removes spaces.

outStr = *stringOp*((str) -> {

String result = "";

**int** i;

**for** (i = 0; i < str.length(); i++)

**if** (str.charAt(i) != ' ')

result += str.charAt(i);

**return** result;

}, inStr);

System.***out***.println("The string with spaces removed: " + outStr);

// Of course, it is also possible to pass a StringFunc instance

// created by an earlier lambda expression. For example,

// after this declaration executes, reverse refers to an

// instance of StringFunc.

StringFunc reverse = (str) -> {

String result = "";

**int** i;

**for** (i = str.length() - 1; i >= 0; i--)

result += str.charAt(i);

**return** result;

};

// Now, reverse can be passed as the first parameter to stringOp()

// since it refers to a StringFunc object.

System.***out***.println("The string reversed: " + *stringOp*(reverse, inStr));

}

}

The output is shown here:

Here is input string: Lambdas add power to Java

The string in uppercase: LAMBDAS ADD POWER TO JAVA

The string with spaces removed: LambdasaddpowertoJava

The string reversed: avaJ ot rewop dda sadbmaL

**Method References**

1. Method References to static Methods

To create a **static** method reference, use this general syntax:

*ClassName*::*methodName*

Notice that the class name is separated from the method name by a double colon. The**::** is a separator that was added to Java by JDK 8 expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a **static** method reference:

//Demonstrate a method reference for a static method.

//A functional interface for string operations.

interface StringFunc {

String func(String n);

}

//This class defines a static method called strReverse().

class MyStringOps {

//A static method that reverses a string.

static String strReverse(String str) {

String result = "";

int i;

for (i = str.length() - 1; i >= 0; i--)

result += str.charAt(i);

return result;

}

}

class MethodRefDemo {

// This method has a functional interface as the type of

// its first parameter. Thus, it can be passed any instance

// of that interface, including a method reference.

static String stringOp(StringFunc sf, String s) {

return sf.func(s);

}

public static void main(String[] args) {

String inStr = "Lambdas add power to Java";

String outStr;

// Here, a method reference to strReverse is passed to stringOp().

outStr = stringOp(MyStringOps::strReverse, inStr);

System.out.println("Original string: " + inStr);

System.out.println("String reversed: " + outStr);

}

}

The output is shown here:

Original string: Lambdas add power to Java

String reversed: avaJ ot rewop dda sadbmaL

In the program, pay special attention to this line:

outStr = stringOp(MyStringOps::strReverse, inStr);

Here, a reference to the **static** method **strReverse( )**, declared inside **MyStringOps**, is passed as the first argument to **stringOp( )**. This works because **strReverse** is compatible with the **StringFunc** functional interface. Thus, the expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func( )** in **StringFunc**.

1. Method References to Instance Method:

To pass a reference to an instance method on a specific object, use this basic syntax:

*objRef*::*methodName*

//Demonstrate a method reference to an instance method

//A functional interface for string operations.

interface StringFunc {

String func(String n);

}

//Now, this class defines an instance method called strReverse().

class MyStringOps {

String strReverse(String str) {

String result = "";

int i;

for (i = str.length() - 1; i >= 0; i--)

result += str.charAt(i);

return result;

}

}

class MethodRefDemo2 {

//This method has a functional interface as the type of

//its first parameter. Thus, it can be passed any instance

//of that interface, including method references.

static String stringOp(StringFunc sf, String s) {

return sf.func(s);

}

public static void main(String[] args) {

String inStr = "Lambdas add power to Java";

String outStr;

//Create a MyStringOps object.

MyStringOps strOps = new MyStringOps();

//Now, a method reference to the instance method strReverse

//is passed to stringOp().

outStr = stringOp(strOps::strReverse, inStr);

System.out.println("Original string: " + inStr);

System.out.println("String reversed: " + outStr);

}

}

1. Method References with Generics.

You can use method references with generic classes and/or generic methods. For example, consider the following program:

//Demonstrate a method reference to a generic method

//declared inside a non-generic class.

//A functional interface that operates on an array

//and a value, and returns an int result.

interface MyFunc<T> {

int func(T[] vals, T v);

}

//This class defines a method called countMatching() that

//returns the number of items in an array that are equal

//to a specified value. Notice that countMatching()

//is generic, but MyArrayOps is not.

class MyArrayOps {

static <T> int countMatching(T[] vals, T v) {

int count = 0;

for (int i = 0; i < vals.length; i++)

if (vals[i] == v)

count++;

return count;S

}

}

class GenericMethodRefDemo {

// This method has the MyFunc functional interface as the

// type of its first parameter. The other two parameters

// receive an array and a value, both of type T.

static <T> int myOp(MyFunc<T> f, T[] vals, T v) {

return f.func(vals, v);

}

public static void main(String[] args) {

Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };

String[] strs = { "One", "Two", "Three", "Two" };

int count;

count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);

System.out.println("vals contains " + count + " 4s");

count = myOp(MyArrayOps::<String>countMatching, strs, "Two");

System.out.println("strs contains " + count + " Twos");

}

}

The output is shown here:

vals contains 3 4s strs contains 2 Twos