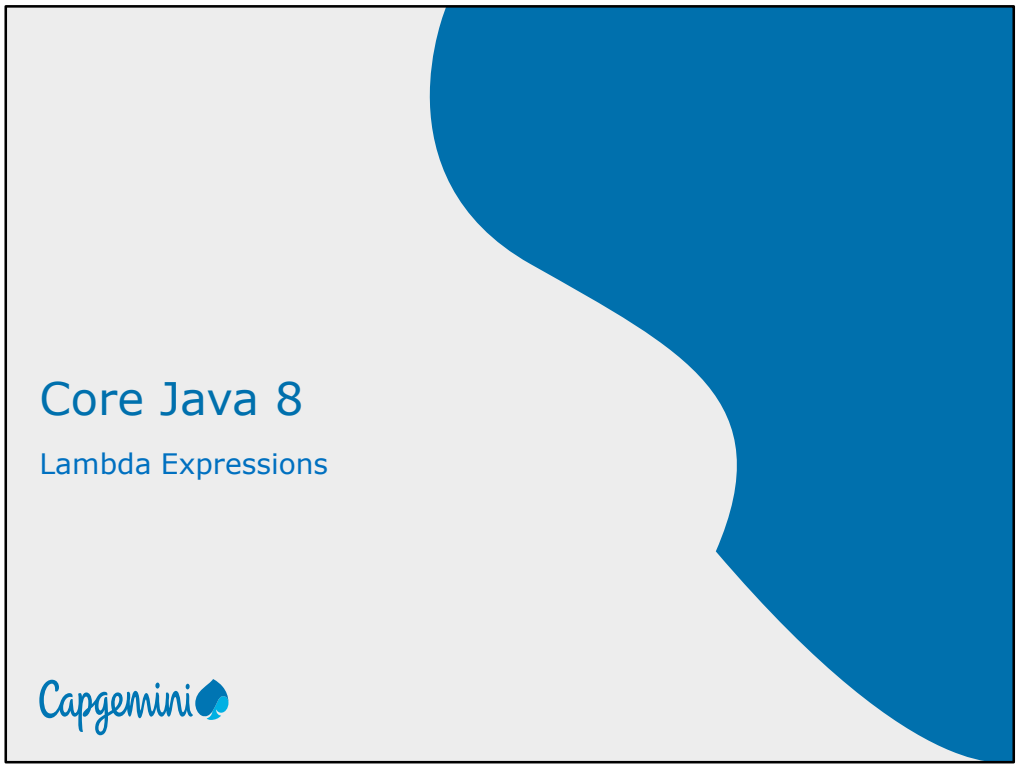
Core Java 8
Lambda Expressions
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand the concept of Lambda expressions
- Work with lambda expressions
- Use method references and functional interfaces

This lesson covers new feature in Java 8, lambda expressions. It also covers concepts of functional interfaces and method references.

Lesson outline:

Introduction to Functional Interface

Functional Interface

Functional Interface is an interface having exactly one abstract method
Such interfaces are marked with optional @FunctionalInterface annotation

```
@FunctionalInterface
interface xyz {
        //single abstract method
}
```

Before we get into the discussion of Lambda expressions, lets first have a look at functional interfaces. When an interface is declared with only one abstract method, then it is referred as Functional Interface. The method in functional interface is called as functional method. Along with the functional method, you can also add default and static method to functional interface. Optionally such interfaces can be annotated with @FunctionalInterface annotation. This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.

Note: An empty interface is called as “Marker Interface”.

Introduction to Functional Interface

Functional Interface : Example

```
@FunctionalInterface
public interface MaxFinder {
    //single abstract method to find max between two
numbers
    public int maximum(int num1,int num2);
}
```

How to implement this interface?

As shown in the slide example, a functional interface is annotated with @FunctionalInterface so as it instructs compiler to rectify any rules violation.

Introduction to Functional Interface

Functional Interface : Implementation

Class Implementation:

```
public class MaxFinderImpl implements MaxFinder {
        @Override
        public int maximum(int num1, int num2) {
                return num1>num2?num1:num2;
        }
}
```

```
MaxFinder finder = new MaxFinderImpl();
int result = finder.maximum(10, 20);
```

Want to know more concise way for implementation?

The slides shows one way to implement the functional interfaces. Is it worthy to create separate class for single method implementation?

Introduction to Functional Interface

Functional Interface : Implementation

Lambda Expression:

```
public class MaxFinderImpl implements MaxFinder {
      @Override
      public int maximum(int num1, int num2) {
                  return num1>num2?num1:num2;
      }
}
```

```
MaxFinder finder = (num1,num2) ->
num1>num2?num1:num2;
int result = finder.maximum(10, 20);
```

Return type of "λE" is Functional Interface!

The slides shows how to implement the functional interfaces using lambda expression. This way is more concise as we are implementing functional interface without creating additional class.

Lets discover the lambda expression in detail.

Writing Lambda Expressions

Lambda Expression

Lambda expression represents an instance of functional interface
A lambda expression is an anonymous block of code that encapsulates an expression or a block of statements and returns a result
Syntax of Lambda expression:

The arrow operator -> is used to separate list of parameters and body of lambda expression

```
(argument list) -> { implementation }
```

What is Lambda expression?

Lambda expression allows for creation and use of single method anonymous classes instead of creating separate concrete class for functional interface implementation.

A lambda expression is an anonymous block of code that encapsulates an expression or a series of statements and returns a result. They can accept zero or more parameters, any of which can be passed with or without type specification since the type can be automatically derived from the context.

The parameter list for a lambda expression can include zero or more arguments. If there are no arguments, then an empty set of parentheses can be used. No parenthesis is required for single argument.

Why Lambda Expressions?

Lambda expressions are an important addition to Java that greatly improves overall maintainability, readability, and developer productivity. They can be applied in many different contexts, ranging from simple anonymous functions to sorting and filtering Collections. Moreover, lambda expressions can be assigned to variables and then passed into other objects.

Writing Lambda Expressions

Lambda Expression

Sample Lambda Expressions

Functional Method	Lambda Expression
int fun(int arg);	(num) -> num + 10
int fun(int arg0,int arg1);	(num1, num2) -> num1+num2

int fun(int arg0,int arg1);	(num1, num2) -> {
	int min = num1>num2?num2:num1;
	return min; }

String fun();	() -> "Hello World!"
void fun();	() -> { }
int fun(String arg);	(str) -> str.length()
int fun(String arg);	str -> str.length()

Lambda Expressions and Functional Interface

Demo

Execute the :

- CalculatorDemo

Builtin Functional Interfaces

Built-in Functional Interfaces

Java SE 8 provides a rich set of 43 functional interfaces
All these interfaces are included under package java.util.function
This set of interfaces can be utilized to implement lambda expressions
All functional interfaces are categorized into four types:

- Supplier
- Consumer
- Predicate
- Function

Built-in Functional Interfaces

As we have learnt so far, functional interfaces can be implemented by writing lambda expressions. Does it means, we have to write functional interface every time if we want to work with Lambda expressions?

Certainly no. As Java 8 comes with dozens of built-in functional interfaces, all are written and kept in java.util.function package. These interfaces can be useful when implementing lambda expressions.

All of these functional interfaces are written in generics format and hence can be applied in many different contexts. Use of such interfaces can greatly reduce the amount of code.

Functional Interfaces are categories into four types:

Supplier
Consumer
Predicate
Function

Lets discover the each type in detail!

Builtin Functional Interfaces

Supplier

A Supplier<T> represents a function that takes no argument and returns a result of type T.
This is an interface that doesn't takes any object but provides a new one

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

List of predefined Suppliers:

- BooleanSupplier
- IntSupplier
- LongSupplier
- DoubleSupplier etc.

Supplier<T>

As the name suggest, Supplier<T> interface contains only one method get() which accepts no arguments but supplies a new object of type T. There are few predefined suppliers which returns object of specified type. For example, BooleanSupplier supplies Boolean valued results.

Builtin Functional Interfaces

Consumer

A Consumer<T> represents a function that takes an argument and returns no result

A BiConsumer<T,U> takes two objects which can be of different type and returns nothing

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

```
@FunctionalInterface
public interface BiConsumer<T,U>
{
        void accept(T t, U,u);
}
```

List of predefined Consumer:

- IntConsumer
- LongConsumer
- ObjIntConsumer
- ObjLongConsumer etc.

Consumer<T>/BiConsumer<T,U>

Consumers are of two types, a Consumer<T> accepts a single object and returns nothing, while the another BiConsumer<T> accepts two objects of any type and returns nothing. It contains a single method called accept() as shown in the slide.

Builtin Functional Interfaces
Predicate

A Predicate<T> represents a function that takes an argument and returns true or false result
A BiPredicate<T,U> takes two objects which can be of different type and returns result as either true or false

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

```
@FunctionalInterface
public interface
BiPredicate<T,U> {
        boolean test(T t, U,u);
}
```

List of predefined Predicates:
- IntPredicate
- LongPredicate
- DoublePredicate etc.

Predicate<T>/BiPredicate<T,U>

In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument. The other type BiPredicate is a predicate of two arguments which returns a Boolean value.

Builtin Functional Interfaces
Function

A Function<T> represents a function that takes an argument and returns another object
A BiFunction<T,U> takes two objects which can be of different type and returns one object

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

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

List of predefined Functions:
- DoubleFunction<R>
- IntFunction<R>
- IntToDoubleFunction
- DoubleToIntFunction
- DoubleToLongFunction etc.

Function<T,R>/BiFunction<T,U,R>

Function<T> represents a function that takes an argument of type T and returns a result of type R. BiFunction<T,U> represents a function that takes two arguments of types T and U, and returns a result of type R.

UnaryOperator<T,T>

Inherits the Function<T,T>, where it accepts and return a result of type T.

BinaryOperator<T>

Inherits from BiFunction<T,T,T>. Represents a function that takes two arguments of the same type and returns a result of the same.

Having discussed the different types of functional interfaces, let see now how to use them to write lambda expressions.

Builtin Functional Interfaces and Lambda Expressions

Lambda Expression for Function Interfaces

Writing Lambda Expressions for Predefined Functional Interfaces

Functional Interface	Functional Method	Lambda Expression
Supplier<String>	String get();	() -> “Hello World”;
BooleanSupplier	Boolean get();	() -> { return true; }
Consumer<String>	void accept(String str);	(msg) -> System.out.println(msg);
IntConsumer	void accept(Integer num);	(num) -> System.out.println(num);
Predicate<Integer>	boolean test(Integer num);	(num) -> num>0;
Function<String,Integer>	Integer apply(String str);	(str) -> str.length;
UnaryOperator<Integer>	Integer apply(Integer num);	(num) -> num +10;
BiFunction<String,String, Boolean>	Boolean apply(String user,String pass);	(user,pass) -> { //functionality to validate user }

Builtin Functional Interfaces and Lambda Expressions

Using Built-in Functional Interfaces

```
Consumer<String> consumer = (String str)-> System.out.println(str);
consumer.accept("Hello LE!");
Supplier<String> supplier = () -> "Hello from Supplier!";
consumer.accept(supplier.get());
//even number test
Predicate<Integer> predicate = num -> num%2==0;
System.out.println(predicate.test(24));
System.out.println(predicate.test(20));
//max test
BiFunction<Integer, Integer, Integer> maxFunction = (x,y)->x>y?x:y;
System.out.println(maxFunction.apply(25, 14));
```

Using Built-in Functional Interfaces

The first example show a consumer of type string which accepts a string and return nothing but print the accepted string value.

The supplier is of type String which supplies an object of String to a consumer which prints it.

The predicate is used to accept integer objects and returns true or false based on even test on given number.

The last example shows, a BiFunction which accepts two integer objects and return an integer which is maximum. The same expression can also be written as:

BinaryOperator<Integer> maxFunction = (x,y) -> x>y?x:y;

The functional interfaces described so far are utilized throughout the JDK, and they can also be utilized in developer applications.

Functional Interface
Demo

Execute the :
- FunctionalInterfaces

Method Reference

Method References

Method reference is a shorthand way to write lambda expressions
It is a new way to refer a method by its name instead of calling it directly
Consider the below lambda expression, which call println method of System.out object:

Such lambda expressions are candidate for method references as it just calling a method for some functionality
The same expression can be written as with method reference:

```
Consumer<String> consumer = (String str)->
System.out.println(str);
```

```
Consumer<String> consumer = System.out :: println;
```

Method Reference

If a lambda expression is written only to invoke a single method by name, then such expression can be shorthanded by using method reference. The syntax of method reference is:

<class or instance name> :: <method name>

The double colon operator specifies the method reference. The method which is referred must reflect pre-defined or custom functional interface.

The slide example shows method println() which accept object to print and returns nothing, suits perfectly with predefined functional interface called Consumer<T> and matches with abstract method void accept(T) signature. Hence lambda expression can be short handed by refereeing it with method reference.

Method reference
Demo

Execute the :
- MethodReference

Method reference
Lab

Lab14

Summary

In this lesson, you have learnt:

- Writing Lambda Expressions
- Functional Interfaces
- Method reference

Review Question

Question 1 : Which of the following Lambda expressions are valid to perform addition of two numbers?

- **Option 1 :** (x, y) -> x +y;
- **Option 2 :** (Integer x,Integer y) - > {return x+y;}
- **Option 3 : (**Integer x, Integer y) -> (x + y);
- **Option 4**: All of above

Question 2 : _____ is a predicate of two arguments.

Question 3 : A method reference is shorthand to create a lambda expression using an existing method.

- True/False