OCP Java SE 8
Programmer II
Exam Guide

Chapter 8

Lambda Expressions and Functional interfaces

Business
Training

1

1Z0-809 Chapter 8

Content

- Using Built-in Functional Interfaces
- Primitive Versions of Predicate Interface
- Binary Versions of Functional Interfaces
- The UnaryOperator Interface

M. Romdhani, 2020

2

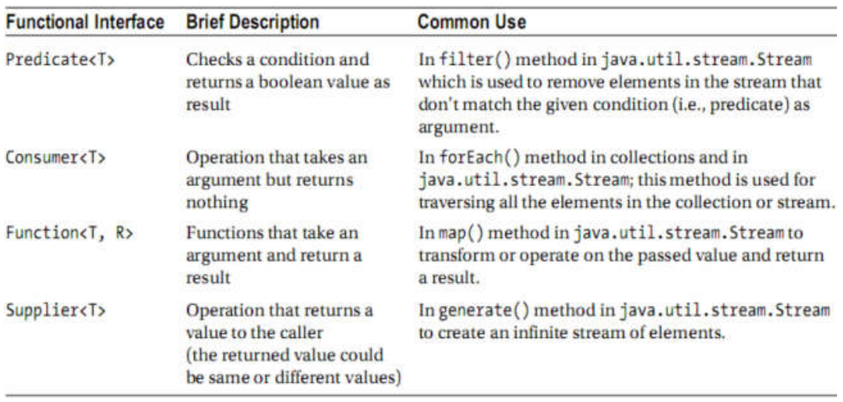
2

Using Built-in Functional Interfaces

3

Key Functional Interfaces in java.util.function Package

- There are four important built-in interfaces included in the java.util.function package: Predicate, Consumer, Function, and Supplier.



Functional Interface	Brief Description	Common Use
`Predicate<T>`	Checks a condition and returns a boolean value as result	In `filter()` method in `java.util.stream.Stream` which is used to remove elements in the stream that don't match the given condition (i.e., predicate) as argument.
`Consumer<T>`	Operation that takes an argument but returns nothing	In `forEach()` method in collections and in `java.util.stream.Stream`; this method is used for traversing all the elements in the collection or stream.
`Function<T, R>`	Functions that take an argument and return a result	In `map()` method in `java.util.stream.Stream` to transform or operate on the passed value and return a result.
`Supplier<T>`	Operation that returns a value to the caller (the returned value could be same or different values)	In `generate()` method in `java.util.stream.Stream` to create an infinite stream of elements.

4

Key Functional Interfaces in java.util.function Package

- **Abstract method declarations in key functional interfaces in java.util.function package**

Predicate<T>	• boolean test(T t)
Consumer<T>	• void accept(T t)
Function<T, R>	• R apply(T t)
Supplier<T>	• T get()

The Predicate Interface

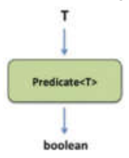
- **Consider the following code segment:**

```
Stream.of("hello", "world")
    .filter(str -> str.startsWith("h"))
    .forEach(System.out::println);
```

  - In this code, the filter() method takes a Predicate as an argument.

- **Here is the Predicate functional interface:**

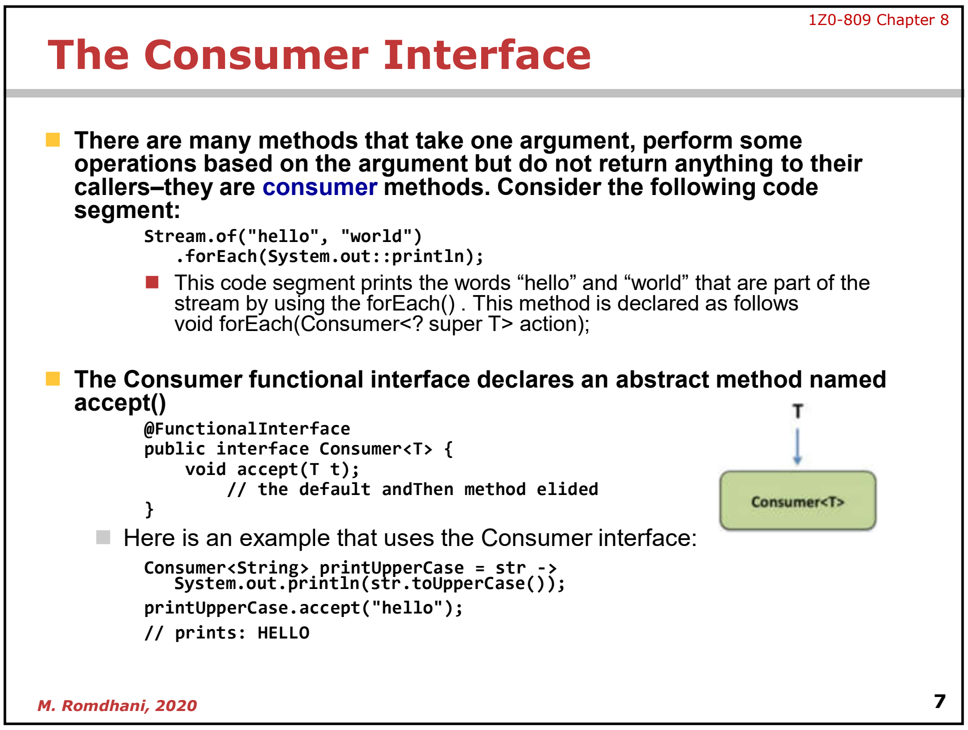
```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // other methods elided
}
```



  - This functional interface also defines default methods named and() and or() that take a Predicate and return a Predicate.

```
public class PredicateTest {
    public static void main(String []args) {
        Predicate<String> nullCheck = arg -> arg != null;
        Predicate<String> emptyCheck = arg -> arg.length() > 0;
        Predicate<String> nullAndEmptyCheck = nullCheck.and(emptyCheck);
        String helloStr = "hello";
        System.out.println(nullAndEmptyCheck.test(helloStr));

        String nullStr = null;
        System.out.println(nullAndEmptyCheck.test(nullStr));
    }
}
```



The Consumer Interface

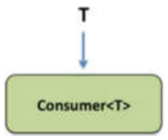
- **There are many methods that take one argument, perform some operations based on the argument but do not return anything to their callers–they are consumer methods. Consider the following code segment:**

```
Stream.of("hello", "world")
   .forEach(System.out::println);
```

  - This code segment prints the words “hello” and “world” that are part of the stream by using the forEach() . This method is declared as follows
    void forEach(Consumer<? super T> action);

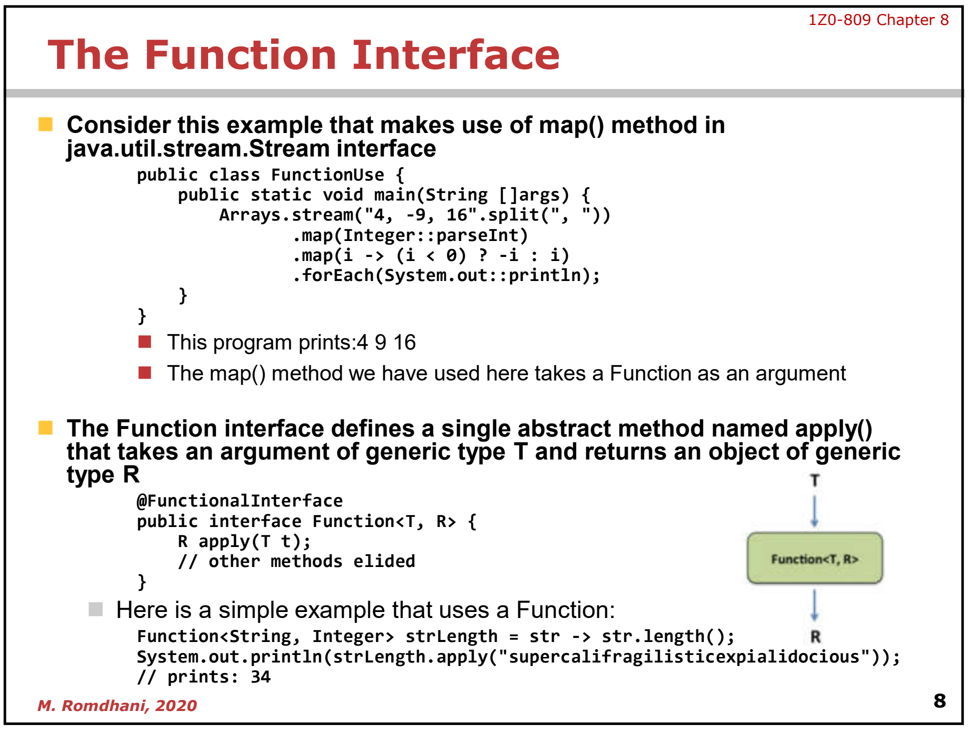
- **The Consumer functional interface declares an abstract method named accept()**

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
        // the default andThen method elided
}
```



  - Here is an example that uses the Consumer interface:

```
Consumer<String> printUpperCase = str ->
   System.out.println(str.toUpperCase());
printUpperCase.accept("hello");
// prints: HELLO
```



The Function Interface

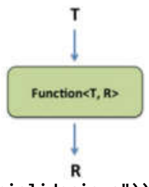
- **Consider this example that makes use of map() method in java.util.stream.Stream interface**

```
public class FunctionUse {
    public static void main(String []args) {
        Arrays.stream("4, -9, 16".split(", "))
               .map(Integer::parseInt)
               .map(i -> (i < 0) ? -i : i)
               .forEach(System.out::println);
    }
}
```

  - This program prints:4 9 16
  - The map() method we have used here takes a Function as an argument

- **The Function interface defines a single abstract method named apply() that takes an argument of generic type T and returns an object of generic type R**

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



  - Here is a simple example that uses a Function:

```
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("supercalifragilisticexpialidocious"));
// prints: 34
```

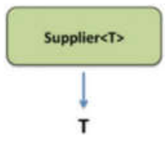
The Supplier Interface

- **In programs, often we need to use a method that does not take any input but returns some output. Consider the following program that generates Boolean values**

```
class GenerateBooleans {
    public static void main(String []args) {
        Random random = new Random();
         Stream.generate(random::nextBoolean)
                  .limit(2)
                  .forEach(System.out::println);
    }
}
```

  - This program randomly prints two boolean values, for example, “true” and “false”. The generate() method in Stream interface is a static member that takes a Supplier as the argument: **static <T> Stream<T> generate(Supplier<T> s)**
  - You can pass the method reference for nextBoolean to Stream’s generate() method because it matches the abstract method in the Supplier interface, i.e., T get()

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
    // no other methods in this interface
}
```



  - Here is a simple example that returns a value without taking anything as input:

```
Supplier<String> currentDateTime = () -> LocalDateTime.now().toString();
System.out.println(currentDateTime.get());
```

Primitive Versions of Predicate Interface

Primitive Versions of Predicate Interface

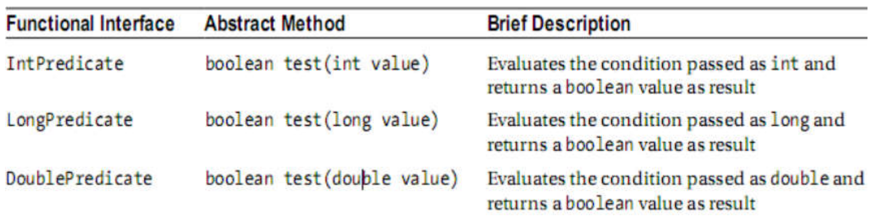
- **Consider this example:**

```
IntStream.range(1, 10).filter(i ->
                    (i % 2) == 0).forEach(System.out::println);
```

  - Here the filter() method takes an **IntPredicate** as the argument since the underlying stream is an **IntStream**. Here is the equivalent code that explicitly uses an IntPredicate:

```
IntPredicate evenNums = i -> (i % 2) == 0;
IntStream.range(1, 10).filter(evenNums).forEach(System.out::println);
```

- **Primitive Versions of Predicate Interface**



Functional Interface	Abstract Method	Brief Description
IntPredicate	boolean test(int value)	Evaluates the condition passed as int and returns a boolean value as result
LongPredicate	boolean test(long value)	Evaluates the condition passed as long and returns a boolean value as result
DoublePredicate	boolean test(double value)	Evaluates the condition passed as double and returns a boolean value as result

Primitive Versions of Function Interface

- **Here is an example that uses a Stream with the primitive type integers:**

```
AtomicInteger ints = new AtomicInteger(0);
Stream.generate(ints::incrementAndGet).limit(10).forEach(System.out::println);
```

  - // prints integers from 1 to 10 on the console
  - This code calls the **int incrementAndGet()** method defined in the class java.util.concurrent.atomic.AtomicInteger. Note that this method returns an int and not an Integer.
    - Still, we can use it with Stream because of implicit autoboxing and unboxing to and from int's wrapper type Integer. This boxing and unboxing is simply unnecessary. Instead you can use the IntStream interface; its generator() method takes an IntSupplier as an argument. With this change, here is the equivalent code:

```
AtomicInteger ints = new AtomicInteger(0);
IntStream.generate(ints::incrementAndGet).limit(10).forEach(
   System.out::println);
// prints integers from 1 to 10 on the console
```

Primitive Versions of Consumer Interface

- **Depending on the kind of arguments, there are numerous versions of primitive types for Consumer interface available**

Functional Interface	Abstract Method	Brief Description
`IntConsumer`	`void accept(int value)`	Operates on the given `int` argument and returns nothing
`LongConsumer`	`void accept(long value)`	Operates on the given `long` argument and returns nothing
`DoubleConsumer`	`void accept(double value)`	Operates on the given `double` argument and returns nothing
`ObjIntConsumer<T>`	`void accept(T t, int value)`	Operates on the given generic type argument `T` and `int` arguments and returns nothing
`ObjLongConsumer<T>`	`void accept(T t, long value)`	Operates on the given generic type argument `T` and `long` arguments and returns nothing
`ObjDoubleConsumer<T>`	`void accept(T t, double value)`	Operates on the given generic type argument `T` and `double` arguments and returns nothing

Primitive Versions of Supplier Interface

- **The primitive versions of Supplier are BooleanSupplier, IntSupplier, LongSupplier, and DoubleSupplier that return boolean, int, long, and double respectively**

Functional Interface	Abstract Method	Brief Description
`BooleanSupplier`	`boolean getAsBoolean()`	Takes no arguments and returns a `boolean` value
`IntSupplier`	`int getAsInt()`	Takes no arguments and returns an `int` value
`LongSupplier`	`long getAsLong()`	Takes no arguments and returns a `long` value
`DoubleSupplier`	`double getAsDouble()`	Takes no arguments and returns a `double` value

Binary Versions of Functional Interfaces

15

Binary Versions of Functional Interfaces

- **Here is the binary version of the Function interface:**

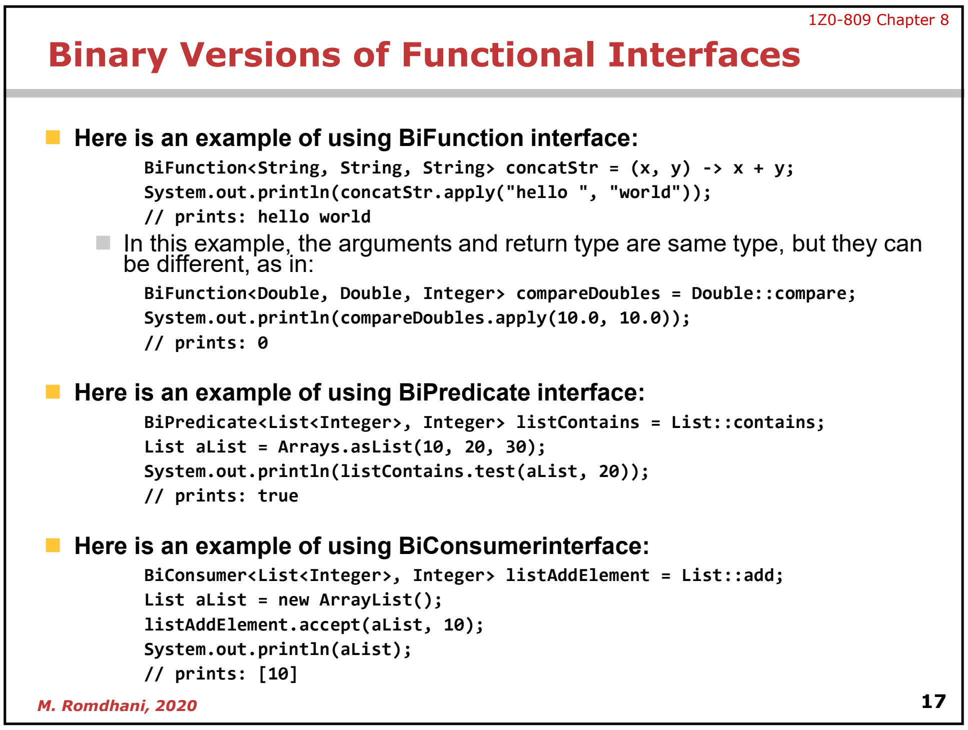
```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    // other methods elided
}
```

  - a **BiFunction** is similar to Function, but the difference is that it takes two arguments: it takes arguments of generic types T and U and returns an object of generic type R. You can call apply() method on a **BiFunction** object.

- **Binary Versions of Functional Interfaces**

Functional Interface	Abstract Method	Brief Description
`BiPredicate<T, U>`	`boolean test(T t, U u)`	Checks if the arguments match the condition and returns a boolean value as result
`BiConsumer<T, U>`	`void accept(T t, U u)`	Operation that consumes two arguments but returns nothing
`BiFunction<T, U, R>`	`R apply(T t, U u)`	Function that takes two argument and returns a result

16



Binary Versions of Functional Interfaces

- **Here is an example of using BiFunction interface:**

```
BiFunction<String, String, String> concatStr = (x, y) -> x + y;
System.out.println(concatStr.apply("hello ", "world"));
// prints: hello world
```

  - In this example, the arguments and return type are same type, but they can be different, as in:

```
BiFunction<Double, Double, Integer> compareDoubles = Double::compare;
System.out.println(compareDoubles.apply(10.0, 10.0));
// prints: 0
```

- **Here is an example of using BiPredicate interface:**

```
BiPredicate<List<Integer>, Integer> listContains = List::contains;
List aList = Arrays.asList(10, 20, 30);
System.out.println(listContains.test(aList, 20));
// prints: true
```

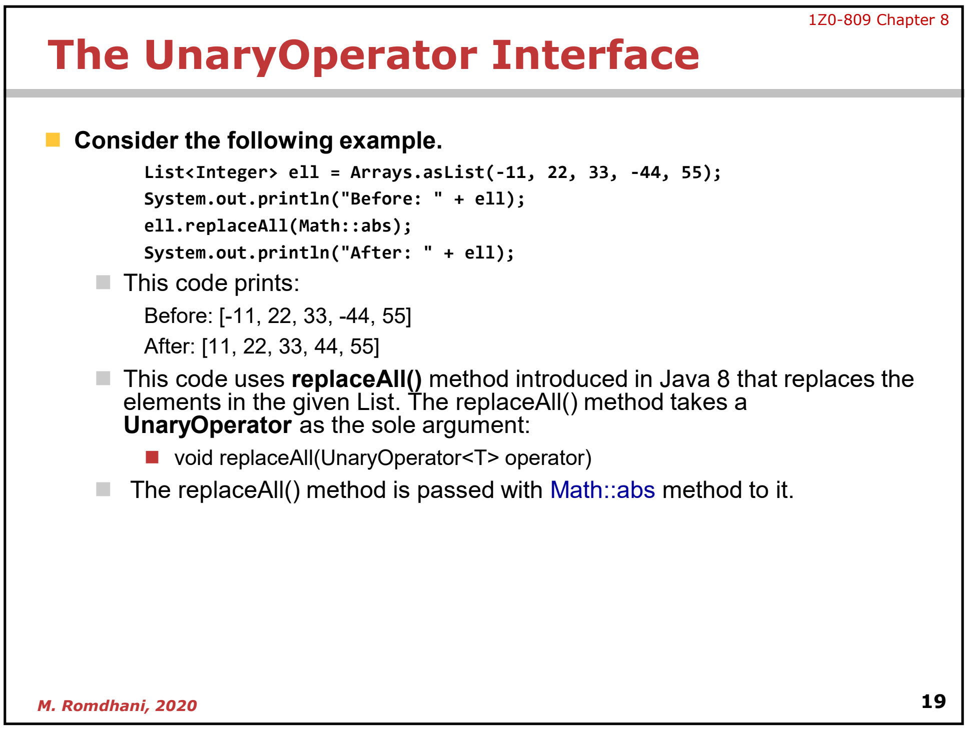
- **Here is an example of using BiConsumerinterface:**

```
BiConsumer<List<Integer>, Integer> listAddElement = List::add;
List aList = new ArrayList();
listAddElement.accept(aList, 10);
System.out.println(aList);
// prints: [10]
```

17

The UnaryOperator Interface

18



1Z0-809 Chapter 8

The UnaryOperator Interface

- **Consider the following example.**

```
List<Integer> ell = Arrays.asList(-11, 22, 33, -44, 55);
System.out.println("Before: " + ell);
ell.replaceAll(Math::abs);
System.out.println("After: " + ell);
```

  - This code prints:

```
Before: [-11, 22, 33, -44, 55]
After: [11, 22, 33, 44, 55]
```

  - This code uses **replaceAll()** method introduced in Java 8 that replaces the elements in the given List. The replaceAll() method takes a **UnaryOperator** as the sole argument:
    - void replaceAll(UnaryOperator<T> operator)
  - The replaceAll() method is passed with Math::abs method to it.

M. Romdhani, 2020 19

19