JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. **Functional Programming**

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Functional Programming

- The most of programming languages use the *imperative programming style*: by defining a series of statements, you are telling the computer ***what*** *to do* to accomplish a particular task with a sequence of *statements (if, for, ...)*.
- JDK 8 brought *functional programming* that uses a *declarative programming style*: you describe ***how*** *your program should work (filter(x -> x > 0), map((a, b) -> a + 2 * b, ...)*.
- Functional programming has advantages:

1) allows you to write shorter and more clear code;
2) avoid a major source of bugs in code and increase speed by memoization;
3) ease of organizing its execution by multiple threads;
4) easy testing and debugging.

- This is achieved through the use of *first-class high-order pure functions*

Functional Programming Languages

- In *functional programming languages* (Lisp, Haskell, Elm), functions come to the fore. It is possible to assign them to variables and pass them through arguments to other functions.
- Popular programming languages such as JavaScript, Python, Java and others have *functional programming tools* to support *functional-style programming*. In Java, they are based on *functional interfaces* and *lambda expressions*.

Pure functions

Pure functions is the functions that 1) *have no I/O or memory side effects* and 2) they depend only on their parameters and only return their own result - the same input will always create the same output.

```
public int square(int x) {
    return x * x;
}
```

Pure function

```
public int div(int a, int b) {
    return a/b;
}
```

"Side-effect" function - it can throw exception

```
List<Integer> list = new ArrayList<>();
public void append(int x) {
    list.add(x);
}
```

"Side-effect" function - list can be changed outside

```
public List<Integer> append(List<Integer> list, int x) {
    List<Integer> aList = new ArrayList<>(list);
    aList.add(x);
    return aList;
}
```

Pure function

Referential Transparency

- Code that doesn't mutate or depend on the external world is said to be *referentially transparent*.
- If the same input will always create the same output, repeated calls can be replaced by the initial result (*memoization*), making the call *referentially transparent* (The Java compiler doesn't support automatic memorization, but some frameworks do, e.g. @Cacheable in Spring).
- Pure functions are referentially transparent. If function uses uncertain data - random numbers, current date or throws supposed to handle Exception - it impure.

Replacing Evaluated Expressions

Abstract Function

$$f(x) = x * x$$

$$\begin{aligned} result &= f(5) + f(5) \\ &= 25 + f(5) \\ &= f(5) + f(5) \\ &= 25 + 25 \end{aligned}$$

function doesn't *do* anything. It only has a value, which is only dependent on its argument

First-class functions

First-class function is function *able to be declared as a variable*. This allows the function to be manipulated as a data type value and executed at the same time.

```
public interface Appendable<E> {
    List<E> append(List<E> list, int x);
}
```

single abstract method only

```
List<Integer> list = new ArrayList<>();
public static void main(String[] args) {
    Appendable<Integer> appendFunc = new Appendable<>() {
        @Override
        public List<Integer> append(List<Integer> list, int x) {
            List<Integer> aList = new ArrayList<>(list);
            aList.add(x);
            return aList;
        }   }; }
```

1st class function

High-order functions

Higher-order function is a function that uses first-class functions:

- *takes one or more functions as arguments*,
- *returns a function as its result*.

```
List<Integer> list = new ArrayList<>();
Comparator<Integer> compareFunc =
        new Comparator<>() {
            @Override
            public int compare(Integer a, Integer b) {
                return a.compareTo(b);
            }
        };
Collections.sort(list, compareFunc);
```

1st class function

high-order function

Why Functional Programming Matters

- "Functional programs contain no assignment statements, so variables, once given a value, never change.
- More generally, functional programs contain no side effects at all. A function call can have no effect other than to compute its result.
- This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side effect can change an expression's value, it can be evaluated at any time.
- This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa—that is, programs are "referentially transparent."
- This freedom helps make functional programs more tractable mathematically than their conventional counterparts".

John Hughes, "Why Functional Programming Matters," from D. Turner, ed., Research Topics in Functional Programming (Addison-Wesley, 1990), 17–42, www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

Lambda Calculus

- Functional programming relies on a mathematical system called *lambda calculus*. This system has two main principles:

1) All functions can be *anonymous* since the only meaningful part of a function header is the argument list;
2) When called, all functions go through a *currying process* - converting a function taking multiple arguments into a sequence of functions that each take only a single argument.

Initial function x = f (a, b, c)

Curried functions: h = g(a)

i = h(b)

x = i(c)

Since the functions are pure, this works.

Java uses lambda through interface anonymous realization

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Functional interfaces

- A **functional interface** is an interface with a single abstract method, called its **functional method**.

```
@FunctionalInterface
public interface StringProcessor {
    String process(String s);
}
```

- If StringProcessor contained more than one abstract method, the @FunctionalInterface annotation would cause a compilation error to be generated.
- Functional interfaces are ideal for defining a single problem or operation. In JDK 8 the API was enhanced to utilize functional interfaces.
- Many of the functional interfaces can contain static and default methods, making them extendable.

See funcifaces\NamedStringProcessor

Lambda Expressions

- The basic form of a **lambda expression** is the following:

  lambda_parameters_list -> lambda_body

- Lambda expressions are used to represent functional interfaces and have functional interfaces type.
- The code specified in **lambda_body** provides the implementation of the functional method.
- The parameters to the functional method are specified in **lambda__parameters_list**.
- Arguments specified when use lambda.

See funcifaces\StringProcessor & NamedStringProcessor

See funcifaces\FiVoid & FiNoParam & Main

See funcifaces\TwoArgsProcessor & TestTwoArgsProcessor

Lambda Body in Block Form

- The lambda can be in expression form or in the block form.
- In block form the lambda body may consist of multiple statements, each ending with a semicolon.
- Lambda expression may contain if statements and loops.
- Return statement provided for lambda expression that returns value.
- Lambdas can “capture” constants and variables from the scope in which the lambda is defined.
- Lambda body's local variables can not shadow variables in enclosing scope.
- Lambda body's local variables can not be redefined (should be final or effectively final).
- Lambda expression may contain exception handling.

See funcifaces\BlockFormLambdaTest

Valid & invalid lambda expressions

1. No parameter

```
() -> true          //valid
-> 1                //invalid, missing variable declaration part
```

2. One parameter

```
a -> a*a            //valid
(a) -> a*a          //valid
(int a) -> a*a      //valid
int a -> a*a        //invalid, if you use the parameter type for the
                    //parameter name, you will need to use parentheses
```

3. More than one parameters

```
(a, b, c) -> a + b + c                  //valid
a, b -> a + b                           //invalid, parameters must be within ( )
(int a, int b, int c) -> a + b + c      //valid
(var a) -> a*a                          //valid
(int a, int b, c) -> a + b + c          //invalid, does not specify the type of c
```

If you apply annotations on the parameter then parameter type is required.

Valid & invalid lambda expressions

4. Expression with or without a return value

```
a -> a + 2                              //valid
a - > return a + 2                      //invalid, must not have return keyword
(a, b) -> System.out.println(a+b)   //method call is a valid expression
```

5. Block of code with or without a return value - the same as writing a method body with or without a return value

```
 a  -> {return a.startsWith("test"); }    //valid
() -> {
          int x = 2;
          int y = 3;
          System.out.println(x+y);
      }             //valid
```

Unlike lambdas expression, the statements within the block end with a semi-colon.

6. Lambdas don't allow overriding default methods of functional interface, non-abstract methods of the functional interface implemented by a lambda expression are not accessible in the lambda body.

The Scope of a Lambda Expression

- Lambdas can “capture” constants and variables from the scope in which the lambda is defined (add some "impurity" for flexibility).
- But these constants and variables must be final or effectively final.

Variable type	Rule
Instance variable	Allowed
Static variable	Allowed
Local variable	Allowed if effectively final
Method parameter	Allowed if effectively final
Lambda parameter	Allowed

See funcifaces\LambdaScopeTest

Basic Models for java.util.function Interfaces

Model	**Functional method**	Has arguments	Returns a value	Description
Predicate	test	yes	boolean	Tests argument and returns true or false.
Function	apply	yes	yes	Maps one type to another.
Consumer	accept	yes	no	Consumes input (returns nothing).
Supplier	get	no	yes	Generates output (using no input).

In JDK 8, the **java.util.function** package was added to the Java API

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Interface java.util.function.Predicate

- In mathematical logic, a **predicate** is commonly understood to be a boolean-valued function:
  X → {true, false} called a predicate on X.

Returns primitive boolean, not Boolean instance!

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    ...
    // some static and default methods
}
```

evaluates a condition on an input variable of a generic type

```
public class TestTest {
    public static void main(String[] args) {
        Predicate<Integer> p1 = x -> x > 7;
        System.out.println(p1.test(9));          //true
        System.out.println(p1.test(3));          //false
    }
}
```

Predicate is often used when filtering or matching

See predicate package

Functional Interfaces Chaining

- Many of the functional interfaces in the **java.util.function** package have **default** and **static** methods that return new functional interface objects, whose methods can, in turn, be called down the **method chain**.
- Using this technique, long chains of functional interfaces can be used to perform series of calculations and to inline the logic of Your program.
- The Predicate chain begin analyzed from the end - from test method argument.

Predicate methods

Modifier and Type	Method and Description
default **Predicate**<**T**>	**and**(**Predicate**<? super **T**> other) Returns a composed predicate that represents a short-circuiting logical AND of this predicate and other predicate.
static <T> **Predicate**<T>	**isEqual**(**Object** targetRef) Returns a predicate that tests if targetRef and **test** method's argument of current predicate are equal according to **Objects.Equals(Object, Object)**.
default **Predicate**<**T**>	**negate**() Returns a predicate that represents the logical negation of the current predicate result.
static <T> **Predicate**<T>	**not**(**Predicate**<? super T> target) Returns a predicate that is the negation of the supplied predicate-argument target result, since JDK 11.
default **Predicate**<**T**>	**or**(**Predicate**<? super **T**> other)Returns a composed predicate that represents a short-circuiting logical OR of this predicate and other predicate.
boolean	**test**(**T** t)Evaluates this predicate on the given argument.

Predicates Chaining

- The Predicate chain begin analyzed from the end - from test() method argument.

See predicate\chaining\PredicateChaining

- In a chain of predicates, the position of negate() and isEqual methods matters.

See predicate\chaining\TestNegate

See predicate\chaining\TestIsEqual

- We can add null-safety to Predicate by abstract and/or default methods override

See predicate\chaining\NullSafePredicate

Specialized Predicates

- The Java API provides the non-generic **IntPredicate**, **LongPredicate**, and **DoublePredicate** interfaces which can be used to test **Integers**, **Longs**, and **Doubles**

```
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
    // other default methods – and, or, negate only
    ...
}
```

- Its using eliminates unnecessary autoboxing-autounboxing into wrapping classes.
- To work with **byte, short**, **char** use IntPredicate, and to work with **float** - DoublePredicate

See predicate\specialized\Main

BiPredicate

- It is often useful to create a single predicate of two different types - BiPredicate<T, U>

```
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
    default BiPredicate<T, U> and(BiPredicate<? super T, ? super U>
                                                    other) {...}
    default BiPredicate<T, U> negate() {...}
    default BiPredicate<T, U> or(BiPredicate<? super T, ? super U>
                                                    other) {...}
}
```

See predicate\bipredicate\Main

Predicate use case

- We can use **default boolean removeIf**(Predicate<? **super** E> filter) method of java.util.Collection to remove collection elements which satisfy the predicate-argument.

See predicate\usecase\Main

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Interface java.util.function.Function

- **Function** is a functional interface with two type parameters **T** and **R**. Its functional method, called **apply**, takes an argument of type **T** and returns an object of type **R**. Functions are ideal for converting an object of type **T** to one of type **R**.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ... // some static and default methods (see later)
}
```

See function\FunctionExample

- You can pass function to method along with an argument of type **T**

See function\Transformer & NumberPaeser

Functions chaining

Modifier and Type	Method and Description
default <V> **Function<T,V>**	**andThen(Function**<? super **R**,? extends **V**> **after)** Returns a composed function that first applies this function to its input, and then applies the **after** function (**andThen** parameter) to the result..
R	**apply**(**T** t) Applies this function to the given argument.
default <V> **Function**<V,**R**>	**compose(Function**<? super **V**,? extends **T**> **before)** Returns a composed function that first applies the **before** function to its input, and then applies this function to the result.
static <T> **Function**<T, T>	**identity()** Returns a function that always returns its input argument.

See function\chaining\Main

Specialized Functions – convert *from* primitive types

The Java API provides the **IntFunction**, **LongFunction**, and **DoubleFunction** interfaces which convert ***from*** **int**, **long**, and **double** primitive types, respectively. These interfaces are generic for a single type parameter which specifies **the type of the object returned** from the apply method.

```
@FunctionalInterface
public interface IntFunction<R> {
   R apply(int value);
}
```

LongFunction<R>,
DoubleFunction<R>
are similar

See function\specialized\Main - 1st part

Specialized Functions – convert *to* primitive types

The Java API provides the **ToIntFunction**, **ToLongFunction**, and **ToDoubleFunction** interfaces which convert ***to*** **int**, **long**, and **double** primitive types, respectively. These interfaces are generic for a single type parameter which specifies the type of the argument to their functional methods.

```
@FunctionalInterface
public interface ToIntFunction<T> {
   int applyAsInt(T value);
}
```

ToLongFunction<T>,
ToDoubleFunction<T>
are similar

See function\specialized\Main - 2nd part

Specialized non-generic primitive types converting Functions

The Java API provides non-generic specializations of the Function interface which convert between **int**, **long**, and **double** primitive types.

```
@FunctionalInterface
public interface IntToLongFunction {
   long applyAsLong(int value);
}
```

public interface IntToDoubleFunction { … similarly…}
public interface LongToIntFunction { … similarly…}
public interface LongToDoubleFunction { … similarly…}
public interface DoubleToIntFunction { … similarly…}
public interface DoubleToLongFunction { … similarly…}

See function\specialized\Main - 3d part

BiFunction

- The BiFunction<T, U, R> specifies two type parameters for input types in addition to the output type parameter

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
   default <V> BiFunction<T, U, V>
                    andThen(Function<? super R, ? extends V> after) {…}
}
```

See function\bifunction\Main

BiFunctions chaining

Modifier and Type	Method and Description
default <V> **BiFunction**<**T**,**U**,**V**>	**andThen(Function**<? super **R**,? extends **V**> **after)** Returns a composed function that first applies this function to its input, and then applies the **after** function (**andThen** parameter) to the result..
R	**apply**(**T** t, **U**, u) Applies this function to the given argument.

See function\bifunction\ChainingMain

Specialized BiFunctions – convert *to* primitive types

The Java API provides the **ToIntBiFunction**, **ToLongBiFunction**, and **ToDoubleBiFunction** interfaces which convert ***to*** **int**, **long**, and **double** primitive types. These interfaces are generic for two type parameters which specify the types of the arguments to their functional method.

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
   int applyAsInt(T t, U u);
}
```

ToLongBiFunction<T , U >,
ToDoubleBiFunction<T , U >
are similar

See function\bifunction\BiFunctionConvert

Custom Functions

- Java provides a built-in interface for functions with one or two parameters. What if you need more?
- Suppose that you want to create a functional interface to determine how fast your quad-copter is going given the power of the four motors. You could create a functional interface such as the following:

```
@FunctionalInterface
interface QuadFunction<T, U, V, W, R> {
   R apply(T t, U u, V v, W w);
}
```

- There are five type parameters here. The first four supply the types of the four motors. Ideally these would be the same type, but you never know. The fifth is the return type in this example.

Function use case

- We can use BiFunction<T, U, R> for computation on Map collection

See function\usecase\Main

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Operator

- When a **Function (BiFunction) parameter(s) input type is the same as the output type**, an **Operator** type interface can be used in place of a **Function**.
- There are **UnaryOperator** and **BinaryOperator** standard functional interfaces.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {…}
```

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {…}
```

UnaryOperator

- **UnaryOperator** is a functional interface with single parameter with type the same as return type.
- Like Function, a lambda expression that represents the **apply** method with a single argument must be provided.
- UnaryOperator supports inherited **andThen**, **compose** and **identity** methods as well.
- UnaryOperator is useful for implementing operations on a single operand.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

See operator\unaryoperator\UnaryOperatorExample

```
@FunctionalInterface
public interface Function<T, T> {
    T apply(T t);
    ... // some static and default methods }
```

Specialized UnaryOperators

- The Java API provides non-generic specializations of the UnaryOperator interface which perform a single operation on an **int**, **long**, or **double** primitive argument, respectively.
- These interfaces include **IntUnaryOperator**, **LongUnaryOperator** and **DoubleUnaryOperator.**

```
@FunctionalInterface
public interface IntUnaryOperator {
   int applyAsInt(int operand);
   default IntUnaryOperator compose(IntUnaryOperator before) {...}
   default IntUnaryOperator andThen(IntUnaryOperator after) {...}
   static IntUnaryOperator identity() {
      return t -> t;
   }
}
```

DoubleUnaryOperator and LongUnaryOperator are similar

See operator\unaryoperator\SpecializedUnaryOperator

Specialized UnaryOperator chaining

Modifier and Type	Method	Description
default **IntUnaryOperator**	**andThen**(**IntUnaryOperator** after)	Returns a composed operator that first applies this operator to its input, and then applies the after operator to the result.
int	**applyAsInt**(int operand)	Applies this operator to the given operand.
default **IntUnaryOperator**	**compose**(**IntUnaryOperator** before)	Returns a composed operator that first applies the before operator to its input, and then applies this operator to the result.
static **IntUnaryOperator**	**identity**()	Returns a unary operator that always returns its input argument.

See operator\unaryoperator\UnaryOperatorsChaining

BinaryOperator

- **BinaryOperator** is a functional interface with two parameters with type the same as return type.
- Like BiFunction, a lambda expression that represents the **apply** method with two arguments must be provided.
- BinaryOperator is useful for implementing operations on two operands.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {
   public static <T> BinaryOperator<T> minBy(Comparator<? super T>
                                             comparator) {…}
   public static <T> BinaryOperator<T> maxBy(Comparator<? super T>
                                             comparator) {… }
}
```

```
@FunctionalInterface
public interface BiFunction<T, T, T> {
   T apply(T t, T u);
   ... // some static and default methods }
```

See operator\binaryoperator\BinaryOperatorExample

Specialized BinaryOperators

- The Java API provides non-generic specializations of the **BinaryOperator** interface which perform a single operation on two **int**, **long**, and **double** primitive arguments, respectively.
- These interfaces include **IntBinaryOperator**, **LongBinaryOperator** and **DoubleBinaryOperator**.

```
@FunctionalInterface
public interface IntBinaryOperator {
   int applyAsInt(int left, int right);
}
```

LongBinaryOperator and DoubleBinaryOperator are similar

See operator\binaryoperator\SpecializedBinaryOperator

BinaryOperators chaining

- There no additional chaining default and static methods in the BinaryOperator, so it can use the default <V> BiFunction<T, U, V> **andThen**(Function<? super R, ? extends V> after) method of BiFunction interface.
- We can not chain the specialized BinaryOperators, because they have not default or static methods and do not extend the BinaryFunction interface.

See operator\binaryoperator\BinaryOperatorsChaining

Operator use case

- We can modify elements in List instance using UnaryOperator.
- We can compare two objects with BinaryOperator.

See operator\usecase

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Consumer

- **Consumer** is a functional interface that is used to process data without returning a processing result. By consume, we mean that we want to do something with the given object.
- Consumer's functional method, called **accept**, takes an argument of type T and has return type **void**.

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> {
            accept(t);
            after.accept(t);
        };
    }
}
```

See consumer\consumer\ConsumerExample

Consumers Chaining

- The Consumer interface's **default** Consumer<T> **andThen**(Consumer<? **super** T> after) method further processes the input argument after the current consumer accept method completes**.**
- Method chains frequently end in a **Terminal Operation Consumer** that displays the result of the processing that occurred along the chain.

See consumer\consumer\ConsumerChaining

See consumer\consumer\ComputePolynomial

Specialized Consumers

- The Java API provides **IntConsumer**, **LongConsumer**, and **DoubleConsumer**, which are non-generic specializations of the Consumer interface.
- They process **int**, **long**, and **double** primitive types, respectively.

```
@FunctionalInterface
public interface IntConsumer {
   void accept(int value);
   default IntConsumer andThen(IntConsumer after) {
      Objects.requireNonNull(after);
      return (int t) -> {
         accept(t);
         after.accept(t); };
   }
}
```

DoubleConsumer and LongConsumer are similar

See consumer\consumer\ConsumerSpecialized

BiConsumer

- It is often useful to process inputs of **two different generic types**.
- The **BiConsumer** functional interface specifies type parameters T and U. Its **accept** method takes arguments of types T and U and has return type **void**.

```
@FunctionalInterface
public interface BiConsumer<T, U> {
   void accept(T t, U u);
   default BiConsumer<T, U> andThen(BiConsumer<? super T,
                                                  ? super U> after) {
      Objects.requireNonNull(after);
      return (l, r) -> {
         accept(l, r);
         after.accept(l, r);
      };
   }
}
```

See consumer\biconsumer\BiConsumerExample

Specialized BiConsumers

- The Java API provides the **ObjIntConsumer**, **ObjLongConsumer**, and **ObjDoubleConsumer** interfaces which specialize the second argument to the **accept** method.

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
   void accept(T t, int value);
}
```

ObjLongConsumer and ObjDoubleConsumer are similar

See consumer\biconsumer\BiConsumerSpecialized

BiConsumers chaining

- The BiConsumer interface's
  default BiConsumer<T, U> andThen(BiConsumer<? super T,
  ? super U> after)
  is used to chain BiConsumers.
- We can not chain the specialized BiConsumers, because they have not default or static methods and do not extend the BiConsumer interface.

See consumer\biconsumer\BiConsumerChaining

Consumer use case

- We can use consumer for collection elements printing while iterating.

See consumer\usecase

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Supplier

- **Supplier** is a functional interface that is used to generate or supply data without any input.
  A Supplier object is specified with type parameter T.
- Its functional method, called **get**, takes no arguments and returns an object of type T.

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

Suppliers supplies an object of type T whenever it's get() method invoked

See supplier\supplier\SupplierExample

- Using suppliers to wrap each user prompt can help to simplify the logic of a program.

See supplier\supplier\UserPromptWrapperSupplier

Specialized Suppliers

- The Java API provides **BooleanSupplier**, **IntSupplier**, **LongSupplier**, and **DoubleSupplier**, which are non-generic specializations of the Supplier interface.
- They generate **boolean**, **int**, **long**, and **double** primitive types, respectively.

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

IntSupplier, LongSupplier and DoubleSupplier are similar

See supplier\supplier\SupplierSpecialized

Suppliers chaining

- We can not chain any Suppliers, because they have not default or static methods.

Supplier use case

- We can process user input with Supplier.

See supplier\usecase\PersonAge

Basic of java.util.function Interfaces

Functional Interface	Functional method	Description
Predicate<T>	boolean test(T)	Tests a condition with argument T and returns true or false.
BiPredicate<T, U>	boolean test(T, U)	Tests two conditions with arguments T and U and returns true or false.
Function<T, R>	R apply(T)	Maps type T to type R.
BiFunction<T, U, R>	R apply(T, U)	Maps types T and U to type R.
UnaryOperator<T>	T apply(T)	Performs single operand operation with argument and return value of the same type.
BinaryOperator<T>	T apply(T, T)	Performs two operand operation with arguments and return value of the same type.
Consumer<T>	void accept(T)	Uses an argument without return value.
BiConsumer<T, U>	void accept(T, U)	Uses two arguments without return value.
Supplier<T>	T get()	Generates or supply return value without taking any input.

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Method Reference

- If we only pass parameter(s) to existing method of the class, then instead of the lambda expression defining we can specify **a reference to the called method** by more concise syntactic construction:

class/object_name :: method_name

See methodreference\FiVoid & Main

Method Reference

There are four formats for method references:

1. Static methods
2. Bound non-static methods
3. Unbound non-static methods
4. Constructors

See methodreference\StaticMehodReference

See methodreference\BoundNonStaticMehodReference

See methodreference\UnboundNonStaticMehodReference

See methodreference\ConstructorReference

Method Reference

Type	Before colon	After colon	Example
Static methods	Class name	Method name	`Collections::sort`
Bound non-static methods	Instance variable name	Method name	`str::startsWith`
Unbound non-static methods	Class name	Method name	`String::startWith`
Constructors	Class name	new	`ArrayList::new`