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. Exceptions and Assertions
11. Nested classes
12. Enums
13. Wrapper classes for primitive types
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java Classes
19. Object Oriented Design
20. **Functional Programing in Java**

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Functional Programming

Functional programming advantages:

- allows you to write shorter and more predictable code;
- increased code reliability;
- increased speed of programs (due to memoization of function arguments and results);
- ease of organizing its execution by multiple threads.

This is achieved through the use of *pure functions* that have no I/O and memory side effects (they depend only on their parameters and only return their own result).

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*. In Java, they are based on *functional interfaces* and *lambda expressions*.

Functional interfaces

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

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

- 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 Java 8, the API was enhanced to utilize functional interfaces.
- Many of the functional interfaces can contain static and default methods, making them extendable by the user

Implementing Functional Interfaces with Pre-Java 8 Constructs

```
class NamedStringProcessor implements StringProcessor {
    @Override
    public String process(String s) {
        return s;
    } }
…
public static void main(String[] args) {
    NamedStringProcessor namedSP =
                                        new NamedStringProcessor();
    StringProcessor anonSP = new StringProcessor() {
        @Override
        public String process(String x) {
            return x.toUpperCase();
        }     };
    System.out.println(namedSP.process("hello"));
    System.out.println(anonSP.process("hello"));   }
```

Named class implemented interface

anonymous class

OUTPUT:
Hello
HELLO

Lambda Expressions

- A ***lambda expression*** is used to represent a statement. The basic form of a lambda expression is the following:

lambda_argument_list -> lambda_body

- Lambda expressions are used to represent functional interfaces. The code specified in **lambda_body** provides the implementation of the functional method. The arguments to the functional method are specified in **lambda_argument_list**.

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

Functional interface

```
x -> x
```

Lambda expression

```
public static void main(String[] args) {
    StringProcessor lambdaSP = x -> x;
    System.out.println(lambdaSP.process("Hello"));
}
```

OUTPUT:
Hello

Lambda Expressions

```
public static void main(String[] args) {
    StringProcessor lambdaSP = x -> x;
    System.out.println(lambdaSP.process("Hello"));
}
```

The same code!!!

```
public static void main(String[] args) {
    StringProcessor lambdaSP = new StringProcessor() {
        @Override
        public String process(String x) {
            return x;
        }     };
    System.out.println(lambdaSP.process("Hello"));
}
```

OUTPUT:
Hello

Lambda Expression for Functional interface with void method

```
@FunctionalInterface
interface FIVoid {
    void method1(int i);
}
```

The **lambda_body** must be an expression that results in a void type.

```
FIVoid lambdaVoid = x -> System.out.println(x);
lambdaVoid.method1(5);
```

OUTPUT:
5

Lambda Expression with method reference

If the lambda expression only calls another method (for example, the method System.out.println(x), as in the previous example), then in the body of the lambda expression you can specify a **reference to** the called **method** by syntactic construction.

```
class/object_name :: method_name
```

For the previous example, the lambda expression will look like this:

FIVoid lambdaVoid = **System.out::println**;

If a reference is made to a class constructor, the syntactic construction in the body of the lambda expression will look like:

class_name :: **new**

The Scope of a Lambda Expression

```
private static int myField = 2;
public static void main(String[] args) {
    int myLocal = 7;
    myLocal++;
    FIVoid lambdaVoid = x -> System.out.println(x + myField
                                                + myLocal);
}
```

ERROR: local variables used in lambdas must be final or effectively final

```
private static int myField = 2;
public static void main(String[] args) {
    int myLocal = 7;
    FIVoid lambdaVoid = x -> System.out.println(x + myField
                                                + myLocal);
    lambdaVoid.method1(5);
}
```

OUTPUT:
14

Lambda Argument List Variations

```
class A {
    int i;
}

class Inference {
    static <T> void m2(Z<T> arg) { }
    public static void main(String[] args) {
        m2(t -> t.i);     // ERROR  - variable t of type Object
        m2((A t) -> t.i); // OK
        m2(new Z<A>() {     //anonymous class implementation
            @Override
            public int m(A t) {
                return t.i;
            }        });    //OK
    }
}
```

```
@FunctionalInterface
interface Z<T> {
    int m(T t);
}
```

Lambda Argument List Variations

```
@FunctionalInterface
public interface FI2 {
    String method1(String x, int y);
}
class Main {
    public static void main(String[] args) {
        /*Two-args lambda_argument_list without types*/
        FI2 fi2Lambda1 = (a, b) -> a + Integer.toString(b);
        System.out.println(fi2Lambda1.method1("Hello", 1));
        /*Two-args lambda_argument_list with types*/
        FI2 fi2Lambda2 = (String c, int d) -> c
                                        + Integer.toString(d + 1);
        System.out.println(fi2Lambda2.method1("Hello", 1));
        /*Without-args lambda_argument_list*/
        FI3 fi3Lambda1 = () -> 99;
        System.out.println(fi3Lambda1.method1());  }   }
```

```
@FunctionalInterface
public interface FI3 {
    Integer method1();
}
```

OUTPUT:

Hello1
Hello2
99

Lambda Bodies in Block Form

Expression Form:

```
FIVoid lambda1 = x -> System.out.println(x);
```

Block Form:

```
FIVoid lambda2 = x -> { System.out.println(x); };
```

```
/*In block form the lambda body may consist of multiple statements, each ending in a semicolon.*/
FIVoid lambdaBlock = x -> {
    x++;
    System.out.println(x);
};
lambdaBlock.method1(5);
```

OUTPUT:
6

Lambda Bodies in Block Form

```
/*A return statement provided for lambda expression that
returns value*/
StringProcessor spBlock = x -> {
    System.out.print(x + " ");
    return x + x;
};
System.out.println(spBlock.process("Hello"));
/*Local variable in lambda body*/
int z = 2;
FIVoid lambdaLocal = x -> {
    int z =4;     //ERROR: z already defined
    int y =4;     //OK
    System.out.println(x + z);
};
lambdaLocal.method1(8);
```

OUTPUT:
Hello HelloHello

OUTPUT:
12

Lambda Bodies in Block Form

```
/*A lambda expression may contain if statements and loops*/
FIVoid lambdaOddSum = x -> {
    int oddSum = 0;
    for (int i = 0; i <= x; ++i) {
    if ( i%2 != 0)
        oddSum += i;
    }
    System.out.println(oddSum);
};
lambdaOddSum.method1(7);
```

OUTPUT:
16

Lambda Bodies in Block Form

```
/*A lambda expression may contain exception handling*/
int[] array = {11,12,13,14,15};
FIVoid lambdaSubscript = x -> {
    try {
        int value = array[x];
        System.out.println(value);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Index " + x + " is out of bounds.");
    }
};
lambdaSubscript.method1(5);
lambdaSubscript.method1(4);
```

OUTPUT:

Index 5 is out of bounds.

15

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).

Interface java.util.function.Predicate

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

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

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
    }
}
```

Passing Predicate to method

```
public class PredicateMethods {
    public static void result(Predicate<Integer> p, Integer arg) {
        if (p.test(arg)) {
            System.out.println("Predicate is true for " + arg);
        } else {
            System.out.println("Predicate is false for " + arg);
        }
    }
    /*The same outer logic applied to different logic of predicate's
      test method*/
    public static void main(String[] args) {
        Predicate<Integer> p1 = x -> x == 5;
        result(p1, 5);                                  //true
        result(y -> y % 2 == 0, 5);                     //false
        result(z -> z * 2.5 == 12.5, 5);                //true
    }
}
```

Passing Generic Predicate to method

```
public class PredicateHelper {
    public static <T> void result(Predicate<T> p, T arg) {
        if (p.test(arg)) {
            System.out.println("Predicate is true for " + arg);
        } else {
            System.out.println("Predicate is false for " + arg);
        }
    }
}
public static void main(String[] args) {
        Predicate<Integer> p1 = x -> x > 2;
        Predicate<String> p2 = s -> s.charAt(0) == 'H';
        result(p1, 6);                    //Predicate is true for 6
        result(p2, "Hello");              //Predicate is true for Hello
    }
}
```

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.

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 another.
static <T> **Predicate**<T>	**isEqual**(**Object** targetRef)Returns a predicate that tests if two arguments are equal according to **Objects.Equals(Object, Object)**.
default **Predicate**<**T**>	**negate**() Returns a predicate that represents the logical negation of this predicate.
static <T> **Predicate**<T>	**not**(**Predicate**<? super T> target) Returns a predicate that is the negation of the supplied predicate.
default **Predicate**<**T**>	**or**(**Predicate**<? super **T**> other)Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	**test**(**T** t)Evaluates this predicate on the given argument.

Predicates Chaining

```
public class PredicateHelper {
    public static <T> void result(Predicate<T> p, T arg) {
        if (p.test(arg)) {
            System.out.println("Predicate is true for " + arg);
        } else {
            System.out.println("Predicate is false for " + arg);
        }     }  }
public static void main(String[] args) {
        Predicate<Integer> p1 = x -> x > 7;
        result(p1.and(y -> y % 2 == 1), 3);     //false
        result(p1.or(x -> x < 3), 9);           //true
        System.out.println(p1.and(Predicate.not(x -> x % 2 == 1))
                                 .test(8));     //true
        System.out.println(p1.or(Predicate.isEqual(3))
                                 .test(3));      //true
        System.out.println(p1.negate().test(9));    //false       }
```

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
    ...
}
```

Specialized Predicates

```
public static void main(String[] args) {
        IntPredicate i = x -> x > 5;
        LongPredicate l = y -> y % 2 == 0;
        DoublePredicate d = z -> z > 8.0;

        System.out.println(i.test(2));                  //false
        System.out.println(l.or(a -> a == 6L)           //true or false
                .test(10L));                            //=true
        System.out.println(d.and(b -> b < 9.0)          //true and true
                .test(8.5));                            //=true

        …
```

Specialized Predicates and Binary Predicate

```
…
    /*Using primitive data types for which there are
      no specialized predicates */
    byte b = 3;
    System.out.println(i.test(b));          //false
    short s = 1000;
    System.out.println(i.test(s));          //true
    char ch='a';
    System.out.println(i.test(ch));         //true
    float f = 5.25f;
    System.out.println(d.test(f));          //false
  }
```

It is often useful to create a single predicate of two different types - BiPredicate

Binary Predicate

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

It is often useful to create a single predicate of two different types

```
public static void main(String[] args) {
    BiPredicate<String, Integer> bi = (x, y) -> x.equals("Manager")
                && y > 100000;
    String position = "Manager";
    int salary = 150000;
    System.out.println(bi.test(position, salary));  //true
}
```

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
}
public class TestTest {
    public static void main(String[] args) {
        Function<String, Integer> f = x -> Integer.parseInt(x);
        Integer i = f.apply("100");
        System.out.println(i);          //100
    }
}
```

Passing Function to method

```
public class Transformer {
    private static <T, R> R transform(T t, Function<T, R> f) {
        return f.apply(t);
    }

    public static void main(String[] args) {
        Function<String, Integer> fsi = x -> Integer.parseInt(x);
        Function<Integer, String> fis = x -> Integer.toString(x);

        Integer i = transform("100", fsi);
        String s = transform(200, fis);
        System.out.println(i);          //100
        System.out.println(s);          //200
    }
}
```

Passing Function to method

```
public class NumberParser {
    private static <R extends Number, String> R parse(String x,
                                                      Function<String, R> f) {
        return f.apply(x);      }
    public static void main(String[] args) {
        ArrayList<Function<String, ? extends Number>> list =
                                                      new ArrayList<>();
        list.add(x -> Byte.parseByte(x));
        list.add(x -> Short.parseShort(x));
        list.add(x -> Integer.parseInt(x));
        list.add(x -> Long.parseLong(x));
        list.add(x -> Float.parseFloat(x));
        list.add(x -> Double.parseDouble(x));
        String[] numbers = {"10", "20", "30", "40", "50", "60"};
        Number[] results = new Number[numbers.length];
        for (int i = 0; i < numbers.length; ++i) {
            results[i] = parse(numbers[i], list.get(i));
            System.out.print(results[i] + " "); //10 20 30 40 50.0 60.0  }   }    }
```

Function methods

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..

Functions Chaining

```
public static void main(String[] args) {
    Function<String, Boolean> fsb = x -> Boolean.parseBoolean(x);
    Function<Boolean, Integer> fbi = x -> x == true ? 1 : 0;
    System.out.println(fsb.andThen(fbi)
            .apply("true"));                          //1
    System.out.println(fbi.compose(fsb)
            .apply("true"));                          //1
    Function<String, String> f = Function.identity();
    System.out.println(f.apply("HELLO"));             //HELLO
}
```

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);     }
public static void main(String[] args) {
    IntFunction<String> fi = x -> String.valueOf(x);
    DoubleFunction<Boolean> fd = x -> x > 5.0;
    LongFunction<Integer> fl = x -> (int) x;
    System.out.println(fi.apply(5));          //5
    System.out.println(fd.apply(4.5));        //false
    System.out.println(fl.apply(20L));        //20        }    }
```

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);     }
public static void main(String[] args) {
    ToIntFunction<String> ti = x -> Integer.parseInt(x);
    ToLongFunction<Double> tl = x -> x.longValue();
    ToDoubleFunction<Integer> td = x -> x.doubleValue();
    System.out.println(ti.applyAsInt("5"));          //5
    System.out.println(tl.applyAsLong(5.1));         //5
    System.out.println(td.applyAsDouble(7));    //7.0}     }
```

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 IntToDoubleFunction {
    double applyAsDouble(int value);
}
```

public interface **IntToLongFunction** { … similarly…}
public interface **DoubleToIntFunction** { … similarly…}
public interface **DoubleToLongFunction** { … similarly…}
public interface **LongToDoubleFunction** { … similarly…}
public interface **LongToIntFunction** { … similarly…}

Specialized non-generic primitive types converting Functions

```
public static void main(String[] args) {
    DoubleToIntFunction di = x -> (new Double(x)).intValue();
    DoubleToLongFunction dl = x -> (new Double(x)).longValue();
    IntToDoubleFunction id = x -> (new Integer(x)).doubleValue();
    IntToLongFunction il = x -> (new Integer(x)).longValue();
    LongToDoubleFunction ld = x -> (new Long(x)).doubleValue();
    LongToIntFunction li = x -> (new Long(x)).intValue();
    System.out.println(di.applyAsInt(4.1));          //4
    System.out.println(dl.applyAsLong(5.2));     //5
    System.out.println(id.applyAsDouble(6));     //6.0
    System.out.println(il.applyAsLong(7));          //7
    System.out.println(ld.applyAsDouble(8));    //8
    System.out.println(li.applyAsInt(9));           //9
}
```

Binary Function

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

The BiFunction specifies two type parameters for input types in addition to the output type parameter

```
public static void main(String[] args) {
    BiFunction<Integer, Character, String> bi = (x, z) -> {
        if (Character.isUpperCase(z))
            return (x%2) == 0 ? "EVEN" : "ODD";
        }
        return (x%2) == 0 ? "even" : "odd";
    };
    System.out.println(bi.apply(4,'U'));      //EVEN
}
```

Binary Functions Chaining

default <V> BiFunction<T, U, V> **andThen**(Function<? super R, ? extends V> after)

The only default method BiFunction interface

```
public static void main(String[] args) {
    BiFunction<Integer, Character, String> bi = (x, z) -> {
        if (Character.isUpperCase(z))
            return (x%2) == 0 ? "EVEN" : "ODD";
        }
        return (x%2) == 0 ? "even" : "odd";
    };
    Function<String, Double> bi2 = x ->
                                    x.equalsIgnoreCase("even") ? 3.0 : 4.0;
    Double d = bi.andThen(bi2)      // Function<String, Double>
            .apply(4,'U');          // BiFunction<Integer,Character, String>
    System.out.println(d);    //3.0
}
```

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 and ToDoubleBiFunction are similar

Specialized BiFunctions – convert TO primitive types

```
public static void main(String[] args) {
    ToIntBiFunction<String, Double> tib = (x,z) ->
                    Integer.parseInt(x) + (new Double(z)).intValue();
    ToLongBiFunction<Double, String> tlb = (x,z) ->
                                    x.longValue() + Long.parseLong(z);
    ToDoubleBiFunction<Integer, Long> tdb = (x,z) ->
                                        (new Integer(x)).doubleValue()
                                    + (new Long(z)).doubleValue();

    System.out.println(tib.applyAsInt("5",4.2));          //9
    System.out.println(tlb.applyAsLong(5.1, "6"));        //11
    System.out.println(tdb.applyAsDouble(7, 8L));         //15.0
}
```

UnaryOperator

When a Function object's input type is the same as its output type, an **Operator** interface can be used in place of a Function. Operator interfaces define a single type parameter.

UnaryOperator is a functional interface with one type parameter such that UnaryOperator<T> inherits Function<T, T>. Like Function, a lambda expression that represents the **apply** method with a single argument must be provided. UnaryOperator supports **andThen**, **compose**, and **identity** as well. It 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;
    }
}
```

added one static method – identity()

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

UnaryOperator

```
public static void main(String[] args) {
    UnaryOperator<String> concat = x -> x + x;
    UnaryOperator<Integer> increment = x -> ++x;
    UnaryOperator<Long> decrement = x -> --x;
    System.out.println(concat.apply("My"));     //MyMy
    System.out.println(increment.apply(4));     //5
    System.out.println(decrement.apply(4L));    //3
    /*Method identify returns a unary operator that always returns
      its input argument.*/
    UnaryOperator<String> sident = UnaryOperator.identity();
    System.out.println(sident.apply("My"));     //My
}
```

```
UnaryOperator<Object> defaultParser = obj -> obj;
```

```
UnaryOperator<Object> defaultParser = UnaryOperator.identity();
```

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**, **DoubleUnaryOperator**, and **LongUnaryOperator**.

```
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int operand);
...
}
```

DoubleUnaryOperator and LongUnaryOperator are similar

Specialized UnaryOperators

```
public static void main(String[] args) {
    IntUnaryOperator iuo = x -> x + 5;
    LongUnaryOperator luo = x -> x / 3L;
    DoubleUnaryOperator duo = x -> x * 2.1;

    System.out.println(iuo.applyAsInt(4));          //9
    System.out.println(luo.applyAsLong(9L));        //3
    System.out.println(duo.applyAsDouble(4.1)); //8.61
}
```

Specialized UnaryOperators Chaining

The UnaryOperator specializations provide **andThen**, **compose**, and **identity** methods that use UnaryOperator specializations as arguments and return types

```
default IntUnaryOperator andThen(IntUnaryOperator after)  …

public static void main(String[] args) {
    IntUnaryOperator iuo = x -> x + 5;
    LongUnaryOperator luo = x -> x / 3L;
    DoubleUnaryOperator duo = x -> x * 2.1;

    System.out.println(iuo.andThen(x -> x * 2).applyAsInt(4));    //18
    System.out.println(luo.compose(x -> x * 6).applyAsLong(4));  /8
    System.out.println(duo.andThen(DoubleUnaryOperator
                .identity()).applyAsDouble(4.1));                 //8.61
}
```

BinaryOperator

BinaryOperator is a functional interface with one type parameter such that BinaryOperator<T> inherits BiFunction<T, T, T>. 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) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;
    }
    public static <T> BinaryOperator<T> maxBy(Comparator<?
                                                super T> comparator) {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;
    }   }
```

added two static methods

BinaryOperator

```
public static void main(String[] args) {
    BinaryOperator<String> concat = (x, y) -> x + y;
    BinaryOperator<Integer> subtract = (x, y) -> x - y;
    BinaryOperator<Long> multiply = (x, y) -> x * y;

    System.out.println(concat.apply("AB", "CD"));      //ABCD
    System.out.println(subtract.apply(4, 1));          //3
    System.out.println(multiply.apply(4L, 3L));        //12

    BinaryOperator<String> maxLengthString = BinaryOperator
                        .maxBy(Comparator.comparingInt(String::length));
    System.out.println(maxLengthString.apply("two", "three"));      //three
    BinaryOperator<String> minLengthString = BinaryOperator
                        .minBy(Comparator.comparingInt(String::length));
    System.out.println(minLengthString.apply("two", "three"));       //two
}
```

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**, **DoubleBinaryOperator**, and **LongBinaryOperator**.

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

DoubleBinaryOperator and LongBinaryOperator are similar

Specialized BinaryOperators

```
public static void main(String[] args) {
    IntBinaryOperator ibo = (x, y) -> x + y + 5;
    LongBinaryOperator lbo = (x, y) -> (x + y) / 3L;
    DoubleBinaryOperator dbo = (x, y) -> x * y * 0.5;

    System.out.println(ibo.applyAsInt(4, 2));                //11
    System.out.println(lbo.applyAsLong(9L, 3L));             //4
    System.out.println(dbo.applyAsDouble(4.0, 6.0));         //12.0
}
```

Consumer

Consumer is a functional interface that is used to process data.
A Consumer object is specified with type parameter T. Its 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); };
    }
}
```

Consumer

```
public class TestConsumer {

    private static int sum = 0;

    public static void main(String[] args) {
        Consumer<Integer> con = x -> sum += x;
        con.accept(4);
        con.accept(5);
        Consumer<Integer> pcon = System.out::println;
        pcon.accept(sum);        //9
    }
}
```

Consumers Chaining

The Consumer interface's default **andThen** method further processes the input argument after the accept method completes.

```
default Consumer<T> andThen(Consumer<? super T> after)  …

public class TestConsumerAndThen {
    private static int sum = 0;
    private static int prod = 1;
    public static void main(String[] args) {
        Consumer<Integer> consum = x -> sum += x;
        Consumer<Integer> conprod = x -> prod *= x;
        consum.andThen(conprod).accept(4);
        consum.andThen(conprod).accept(5);
        System.out.println("sum = " + sum
                           + " prod =" + prod);   //sum = 9 prod =20
    }
}
```

Terminal Operation Consumer

$fx=5x^4+7x^3+4x^2+3x+8$

```
public class ComputePolynomial {
    private static int fx = 0;
    public static void main(String[] args) {
        Consumer<Integer> poly = x -> fx += 5 * (int) Math.pow(x, 4);
        poly.andThen(x -> fx += 7 * (int) Math.pow(x, 3))
                .andThen(x -> fx += 4 * (int) Math.pow(x, 2))
                .andThen(x -> fx += 3 * x)
                .andThen(x -> fx += 8)
                .andThen(x -> System.out.println(fx))  //Terminal Operation
                                                       // Consumer
                .accept(2);                            //166
    }
}
```

The last link in the consumer chain contains a **System.out.println** statement that prints the result of the equation. Method chains frequently end in a **Terminal Operation Consumer** that displays the result of the processing that occurred along the chain.

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

Specialized Consumers

```
public class ConsumerSpecials {
    private static int a = 0;
    private static long b = 0;
    private static double c = 1.0;
    public static void main(String[] args) {
        IntConsumer ic = x -> a = x + 3;
        LongConsumer lc = x -> b = x / 2L;
        DoubleConsumer dc = x -> c = x * c;
        ic.andThen(x -> System.out.println(a)).accept(2);      //5
        lc.andThen(x -> System.out.println(b)).accept(6L);     //3
        dc.andThen(x -> System.out.println(c)).accept(4.0);    //4.0
    }
}
```

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);
        };
    }
}
```

BiConsumer

```
public class TestBiConsumer {
    private static int sum = 0;
    private static String components = "";
    public static void main(String[] args) {
        BiConsumer<Integer, String> bi = (x, y) -> {
            sum += x;
            components += y;
        };
        bi.andThen((x, y) -> System.out.println(x + " " + y))
                .accept(6, "Term 1");                                   //6 Term 1
        bi.andThen((x, y) -> System.out.println("add " + x + " " + y
                + " result = " + sum + " " + components))
                .accept(7, ",Term 2");
                                        //add 7 ,Term 2 result = 13 Term 1,Term 2
    }
}
```

Specialized BiConsumers

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

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

    void accept(T t, int value);

}
```

ObjLongConsumer and ObjDoubleConsumer are similar

Specialized BiConsumers

```
public class BiConsumerSpecialis {

    public static void main(String[] args) {
        ObjIntConsumer<String> oic = (x, y)
                -> System.out.println(x + " = " + y);
        ObjLongConsumer<String> olc = (x, y)
                -> System.out.println(Long.parseLong(x) + y);
        ObjDoubleConsumer<String> odc = (x, y)
                -> System.out.println(x + Double.toString(y));

        oic.accept("Value", 4);                 // Value = 4
        olc.accept("7", 2L);                    // 9
        odc.accept("DBL", 4.1);                 // DBL4.1
    }
}
```

Supplier

Supplier is a functional interface that is used to generate data. 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();
}
```

Supplier

```
public static void main(String[] args) {
    Supplier<Integer> generateInteger = () -> {
        Random rand = new Random();
        return rand.nextInt(100);
    };

    Supplier<String> generateString = () -> {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter a string:");
        return scan.nextLine();
    };

    System.out.println(generateInteger.get());
    System.out.println(generateInteger.get());
    System.out.println(generateString.get());
    System.out.println(generateString.get());
}
```

Output:

73

56

Enter a string: **Hello**

Hello

Enter a string: **World**

World

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

Specialized Suppliers

```
public class TestSpecials {
    public static Random rand = new Random();
    public static void main(String[] args) {
        BooleanSupplier genBol = ()
                -> (rand.nextInt(2) == 1);
        IntSupplier genInt = () -> rand.nextInt();
        LongSupplier genLng = () -> rand.nextLong();
        DoubleSupplier genDbl = () -> rand.nextDouble();
        System.out.println(genBol.getAsBoolean());        // false
        System.out.println(genInt.getAsInt());          // 377421168
        System.out.println(genLng.getAsLong());
                                          // 386839345206793142 2


        System.out.println(genDbl.getAsDouble());
                                              // 0.998053305378937 4
    }  }
```