Лабораторна робота № 4

Використання функціональних інтерфейсів

Мета роботи: Ознайомитись з технологією внутрішніх/вкладених класів та вивчити особливості використання статичних вкладених та локальних і анонімних внутрішніх класів. Вивчити особливості енумераторів у Java.

1. Теоретичні відомості

На сьогоднішній день все більш популярним стає *функціональне програмування*. Його використання дозволяє писати більш короткий і передбачуваний код і має інші переваги, такі як підвищення надійності коду, підвищення швидкості роботи програм (за рахунок *мемоізації* аргументів і результатів функції) і легкість організації його виконання декількома потоками. Це досягається внаслідок використання *чистих функцій*, які не мають побічних ефектів введення-виведення і пам'яті (вони залежать тільки від своїх параметрів і повертають тільки свій результат).

У функціональних мовах програмування (Lisp, Haskell, Elm) на перший план виходять функції. Вони існують самі по собі. Можливо присвоювати їх змінним і передавати через аргументи іншим функціям. Популярні мови програмування, такі як JavaScript, Python, Java та інші, мають засоби функціонального програмування. В Java вони базуються на *функціональних інтерфейсах* та *лямбда-виразах*.

Функціональний інтерфейс - це інтерфейс з єдиним абстрактним методом, який називається *функціональним методом*. Функціональні інтерфейси ідеально підходять для визначення однієї проблеми або операції. У Java 8 API було вдосконалено для використання функціональних інтерфейсів. Багато функціональних інтерфейсів можуть містити статичні та `default` методи, що робить їх можливими до розширювання.

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

Якщо StringProcessor містив би більше одного абстрактного методу, анотація `@FunctionalInterface` призвела б до генерування помилки компіляції.

До Java 8 використання інтерфейсу виконувалось створенням класу, що його реалізує та подальшому створенню об’єкта:

```
class NamedStringProcessor implements StringProcessor {
    @Override
    public String process(String s) {
        return s;
    }
}
public class Main {
    public static void main(String[] args) {
        NamedStringProcessor namedSP =
```

```
                                        new NamedStringProcessor();
        System.out.println(namedSP.process("hello"));
    }
}
```

Програма виведе `hello` після запуску. Також можлива реалізація інтерфейсу як анонімного внутрішнього класу:

```
        StringProcessor anonSP = new StringProcessor() {
            @Override
            public String process(String x) {
                return x.toUpperCase();
            }
      };
        System.out.println(anonSP.process("hello"));
```

Програма виведе `HELLO` після запуску.

Лямбда-вираз - це синтаксична конструкція, яка з'явилась в Java 8 і використовується для представлення оператора (або блока операторів, що складають тіло функції). Основна форма лямбда-виразу така:

```
список_аргументів_лямбда-виразу -> тіло_лямбда-виразу
```

Лямбда-вирази можна розглядати як метод без модифікаторів доступу, значення, що повертається, та імені. Лямбда-вирази використовуються для представлення функціональних інтерфейсів і мають тип функціонального інтерфейсу. Код, зазначений у `тілі_лямбда-виразу`, забезпечує реалізацію функціонального методу. Аргументи функціонального методу вказуються в `списку_аргументів_лямбда-виразу`:

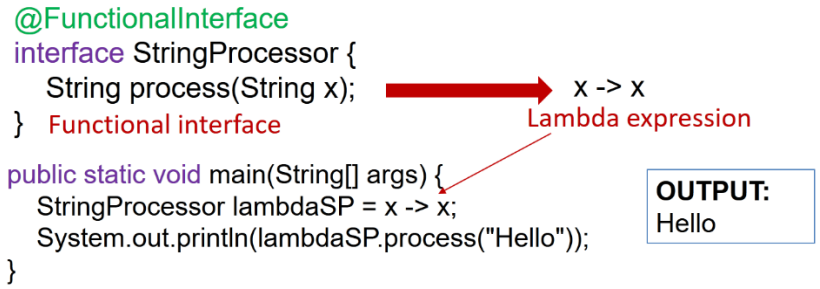

Рис. 1. Заміна функціонального інтерфейсу лямбда-виразом

У разі, якщо функціональний метод є `void`, у `тілі_лямбда-виразу` повинен знаходитись оператор (блок опреаторів), який нічого не повертає:

```
@FunctionalInterface
public interface FIVoid {

    void method1(int x);
}
… in some class …

public static void main(String[] args) {
    FIVoid fiVoid = new FIVoid() { //об'єкт анонімного класу,
```

```
        @Override                       //що реалізує інтерфейс
        public void method1(int x) {
            System.out.println(x);
        }
    };
    FIVoid lambdaVoid = x -> System.out.println(x); //лямбда-
                                                    //вираз
    fiVoid.method1(5);
    lambdaVoid.method1(5);
}
```

Створення об’єкту через анонімний внутрішній клас, що реалізує функціональний інтерфейс, та створення об’єкту через лямбда-вираз є еквівалентним. В сучасних версіях IDE існує можливість автоматичного перетворення коду у ці форми (Рис. 2).

Якщо лямбда-вираз лише викликає інший метод (наприклад, статичний метод `System.out.println(x)`, як у попередньому прикладі), то у тілі лямбда-виразу можна вказати *посилання на метод*, що визивається, синтаксичною конструкцією:

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

Для попереднього прикладу лямбда-вираз при цьому буде виглядати:

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

Зауважимо, що в IDE є команди автоматичного перетворення лямбда-виразу у такий вигляд (*Use member reference* - у Netbeans IDE) (Рис. 2).

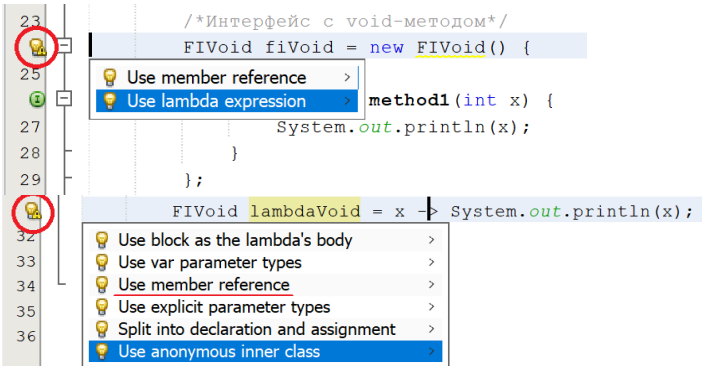

Рис. 2. Засоби автоматичного представлення функціонального коду в різних формах в IDE Netbeans

У разі, якщо виконується посилання на конструктор класу, синтксисична конструкція у тілі лямбда-виразу буде мати вигляд:

```
class_name :: new
```

Якщо функціональний метод має декілька операторів, то тіло лямбда-виразу містить блок операторів у фігурних дужках після якого ставиться крапка з комою:

```
    FIVoid lambdaBlock = x -> {
        x++;
        System.out.println(x);
    };
    lambdaBlock.method1(5);    //6
```

У попередніх прикладах тип аргументу функціонального методу явно не вказаний, оскільки він може бути виведений автоматично з типу аргументу функціонального методу функціонального інтерфейсу. Якщо тип не може бути виведений автоматично, то він вказується явно перед ім'ям аргументу і аргументи беруться у круглі дужки:

```
class A {
    int i;
}

@FunctionalInterface
public interface Z<T> {
    int m(T t);
}
…
public static void main(String[] args) {
//    m2(t -> t.i);   // ERROR: variable t of type Object
    m2((A t) -> t.i); // OK
}
```

Якщо функціональний метод не містить аргументів, це потрібно вказати порожніми круглими скобками:

```
() -> 5
```

У Java 8 був доданий пакет java.util.function. Він містить функціональні інтерфейси, призначені для допомоги у функціональному програмуванні. Ці функціональні інтерфейси відповідають одній з чотирьох базових моделей (Табл. 1).

Табл. 1. Базові моделі функціональних інтерфейсів

Модель	Чи має аргумент	Чи повертає значення	Опис
Predicate	так	boolean	Перевіряє аргумент і повертає `true` або `false`
Function	так	так	Перетворює один тип у інший
Consumer	так	ні	Споживає вхідні дані
Supplier	ні	так	Генерує вихідні дані

Predicate - це функціональний інтерфейс, функціональний метод якого називається `test` та оцінює прописану у тілі метода умову для вхідного аргументу узагальненого типу. Тестовий метод повертає `true`, якщо умова істина, та `false` - інакше. Нижче наведений функціональний інтерфейс `Predicate<T>` та приклад його використаня:

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

```
    boolean test(T t);
    ...    // статичні та default методи
}

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
    }
}
```

Предікат може бути переданий як аргумент методу. Коли об'єкт `Predicate` передається методу і викликається метод `test` цього об'єкта, виконується будь-яка умова, визначена у методі test об'єкта `Predicate`. Отже, якщо визначені кілька об'єктів `Predicate`, одна і та ж логіка метода застосовується до кожної (різної) умові шляхом передачі різних предикатів в метод. Наприклад:

```
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
    }
}
```

Зверніть увагу, що другий та третій предикати передаються як аргумент метода у вигляді лямбда-виразу (це значно коротша та більш виразна форма запису коду).

Багато функціональних інтерфейсів у пакеті java.util.function мають `default` та `static` методи, які повертають нові об'єкти цього ж або іншого функціонального інтерфейсу, методи яких, у свою чергу, можна викликати (таким чином формується *ланцюжок викликів методів функціональних інтерфейсів - Chains of Functional Interfaces*). Використовуючи цю техніку, можуть бути запрограмовані довгі ланцюги функціональних інтерфейсів для виконання послідовних обчислень відповідно до логіки Вашої програми. Методи інтерфейса `Predicate<T>` наведені у Табл. 2.

Табл. 2. Методи інтерфейса `Predicate<T>`

Модифікатор та тип, що повертається	Ім'я та опис метода
`default` **`Predicate<T>`**	**`and`**`(`**`Predicate`**`<? super` **`T`**`> other)` Повертає складений предикат, який є short-circuiting AND поточного предикату та предикату `other`
`static <T>` **`Predicate<T>`**	**`isEqual`**`(`**`Object`** `targetRef)` Повертає предикат, який перевіряє аргумент метода на еквівалентність аргументу предиката
`default` **`Predicate<T>`**	**`negate`**`()` Повертає предикат, який є логічним запереченням поточного предиката
`static <T>` **`Predicate<T>`**	**`not`**`(`**`Predicate`**`<? super T> target)` Повертає предикат, який є логічним запереченням поточного предикату
`default` **`Predicate<T>`**	**`or`**`(`**`Predicate`**`<? super` **`T`**`> other)` Повертає складений предикат, який є short-circuiting OR поточного предикату та предикату `other`
`boolean`	**test(T** t) Перевіряє поточний предикат для аргумента

Нижче наводиться приклад застосування ланцюжків викликів методів для предикату.

```
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);
        }
    }
}
//у деякому-класі
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
}
```

Метод `not` (був доданий в Java 11) змінює на протилежний результат обчислення, виконаного предикатом-аргументом. У той час як метод `negate` змінює результат поточного предиката, з якого цей метод викликається.

Java API надає *non-generic* інтерфейси IntPredicate, LongPredicate та DoublePredicate, які можна використовувати для тестування цілих, довгих та подвійних дійсних чисел, наприклад для цілих:

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

Часто корисно створити єдиний предикат двох аргументів різних типів. Цю можливість надає інтерфейс BiPredicate. Нижче наведений приклад його використання для створення комбінованої умови для аргумента рядка та аргумента цілого числа:

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


//у деякому класі
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
}
```

Функція - це функціональний інтерфейс з двома параметрами типу `T` і `R`. Її функціональний метод, який називається `apply`, приймає аргумент типу `T` і повертає об'єкт типу `R`. Функції ідеально підходять для перетворення об'єкта типу `T` в об’єкт типу `R`.

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

Приклад використання функції (у даному випадку як аргумента метода):

```
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
    }
}
```

Окрім функціонального методу `apply(T t)` функція має дефолтні і статичні методи, що дозволяють пов'язувати виклики методів функції в ланцюжки (Табл. 3).

Табл. 3. Методи інтерфейса `Function<T, R>`

Модифікатор та тип, що повертається	**Ім'я та опис метода**
`default <V>` **`Function<T, V>`**	**`andThen(Function`**`<? super` **`R, ? extends V> after)`** Повертає складену функцію, яка спочатку застосовує поточну функцію до її аргументу, а потім - функцію **`after`** до результату поточної
`R`	**`apply`**`(`**`T`**` t)` Застосовує поточну функцію до її аргументу
`default <V>` **`Function`**`<V,` **`R`**`>`	**`compose(Function`**`<? super V, ? extends` **`T`**`> before)` Повертає складену функцію, яка спочатку застосовує функцію before до свого аргументу, а потім - поточну функцію до отриманого результату
`static <T>` **`Function`**`<T, T>`	**`identity()`** Повертає функцію, яка завжди повертає свій аргумент

Приклад використання методів інтерфейсу `Function<T, R>`:

```
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
}
```

`andThen``:` Метод `apply` перетворює рядок «true» в логічне значення true, яке вводиться в функцію, виконувану методом `andThen`. Ця функція перетворює логічне значення true в цілочисельне значення 1.

compose: Метод `compose` використовує функцію `fsb` для перетворення рядка «true» в логічне значення true. Потім метод `apply` використовує функцію `fbi` для перетворення true в цілочисельне значення 1.

identify: Метод `identity` створює функцію, метод `apply` якої повертає її вхідний параметр. Деякі методи використовують об'єкти `Function`, які відображають вхідний параметр на вихідний. Передача в такі методи метода `Function.identity` викличе обробку методом без зміни вхідного параметра.

Java API надає інтерфейси `IntFunction`, `LongFunction` та `DoubleFunction`, які перетворюють відповідно **з типів** `int`, `long` та `double`. Ці інтерфейси мають узагальний параметр типу, який визначає тип об'єкта, повернутий із методу `apply`.

```
@FunctionalInterface
public interface IntFunction<R> { //LongFunction та
                                  //DoubleFunction аналогічно
    R apply(int value);
}
//у деякому класі
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
}
```

Java API надає інтерфейси `ToIntFunction`, `ToLongFunction` та `ToDoubleFunction`, які перетворюють аргумент узагальненого типу у примітивні типи `int`, `long` та `double`, відповідно.

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

Java API надає не узагальнені спеціалізації інтерфейсу `Function`, які виконують конвертування між типами `int`, `long` та `double`.

```
@FunctionalInterface
```

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

public interface IntToLongFunction { … аналогічно…}
public interface DoubleToIntFunction { … аналогічно …}
public interface DoubleToLongFunction { … аналогічно …}
public interface LongToDoubleFunction { … аналогічно …}
public interface LongToIntFunction { … аналогічно …}
//у деякому класі
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
}
```

Іноді для перетворення за допомогою інтерфейсу `Function` потрібні вхідні параметри двох різних типів. Це забезпечує інтерфейс `BiFunction<T,U,R>` (Binary Function):

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

Параметр x визначає, парний чи непарний результат, тоді як параметр z визначає регістр рядка.

Єдиний додатковий метод інтерфейсу BiFunction - default метод andThen застосовує додаткову операцію після того, як операція, зазначена в методі apply, завершена. Цей метод дозволяє організовувати ланцюжок викликів методів інтерфейсу BiFunction:

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

```
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
}
```

Java API надає інтерфейси ToIntBiFunction, ToLongBiFunction та ToDoubleBiFunction, які перетворюють два аргументи узагальненого типу у примітивні типи int, long та double, відповідно.

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    int applyAsInt(T t, U u);
}
//ToLongBiFunction та ToDoubleBiFunction аналогічно
//У деякому класі
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
}
```

У разі, якщо тип вхідного параметра об’єкта `Function` збігається з типом об’єкта, що повертається, замість функції може бути використаний інтерфейс `UnaryOperator<T>`.

`UnaryOperator<T>` - це функціональний інтерфейс з одним параметром типу. `UnaryOperator<T>` успадковує `Function<T, T>`. Як і для функції, для `UnaryOperator<T>` повинен бути наданий лямбда-вираз, який представляє тіло метода `apply` з одним аргументом. `UnaryOperator` має власну реалізацію статичного метода `identity()` та підтримує методи `andThen` та `compose` інтерфейсу `Function`. Це корисно і для реалізації операцій над одним операндом.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
//У деякому класі
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
    /* Метод identity повертає унарний оператор, який завжди
повертає свій вхідний аргумент*/
    UnaryOperator<String> sident = UnaryOperator.identity();
    System.out.println(sident.apply("My"));     //My
}
```

Повинна бути обов'язково використана префиксная форма для інкремента і декремента, щоб змінювалося значення на виході.

Java API надає узагальнені спеціалізації інтерфейсу `UnaryOperator`, які виконують одну операцію над аргументом `int`, `long` або `double`, відповідно. Ці інтерфейси включають `IntUnaryOperator`, `DoubleUnaryOperator` та `LongUnaryOperator`.

```
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int operand);
...
}
// DoubleUnaryOperator та LongUnaryOperator аналогічно
//у деякому класі
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
}
```

Описані вище спеціалізації `UnaryOperator`, окрім функціонального, містять додаткові методи `andThen`, `compose` та `identity`, які використовують відповідні спеціалізації `UnaryOperator` як вхідні аргументи та типи об'єкту, що повертається. Ці методи дозволяють організовувати ланцюжки виклику методів таких інтерфейсів.

```
default IntUnaryOperator andThen(IntUnaryOperator after) {…}
default IntUnaryOperator compose(IntUnaryOperator before) {…}
static IntUnaryOperator identity() {
        return t -> t;
}
//у деякому класі
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<T>` - це функціональний інтерфейс з одним параметром типу. `BinaryOperator<T>` успадковує `BiFunction<T, T, T>`. Як і для `BiFunction`, для `BinaryOperator<T>` має бути наведений лямбда-вираз, який представляє тіло методу `apply` із двома аргументами. `BinaryOperator` корисний для реалізації операцій над двома операндами. Окрім успадкованих від `BiFunction<T,T,T>` методів (функціонального `R apply(T t, U u)` та `default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after)`), `BinaryOperator` має власні статичні методи `minBy` та `maxBy`:

```
@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;
    }
}
```

Метод `minBy` повертає об'єкт `BinaryOperator`, який налаштований на повернення меншого відповідно до вказаного компаратора з двох елементів-аргументів методу `apply`, що буде викликаний з цього `BinaryOperator`.

Метод `maxBy` повертає об'єкт `BinaryOperator`, який налаштований на повернення більшого відповідно до вказаного компаратора з двох елементів-аргументів методу `apply`, що буде викликаний з цього `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
}
```

Java API надає не узагальнені спеціалізації інтерфейсу `BinaryOperator`, які виконують одну операцію з двома примітивними аргументами `int`, `long` та `double`. Ці інтерфейси включають `IntBinaryOperator`, `DoubleBinaryOperator` та `LongBinaryOperator`.

```
@FunctionalInterface
public interface IntBinaryOperator {
    int applyAsInt(int left, int right);
}
// DoubleBinaryOperator та LongBinaryOperator аналогічно
//у деякому класі
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<T>` - це функціональний інтерфейс, який деяку дію над об'єктом. Його функціональний метод `accept`, приймає аргумент типу `T` і повертає тип `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); };
    }
}
```

Приклад використання - `Consumer<Integer>`, метод `accept` якого додає свій аргумент до статичного поля `sum`:

```
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);
        System.out.println(sum);           //9
    }  }
```

Метод `default andThen` інтерфейса `Consumer<T>` обробляє вхідний аргумент після завершення методу `accept`:

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

//Приклад виористання:
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
    }
}
```

У наведеній програмі ланцюжок викликів методів `Consumer` створює складений `Consumer`: спочатку метод `accept` об’єкта `consum` додає аргумент до накопичуваної у статичному полі `sum` суми, а потім виконується метод `andThen`, що приймає як аргумент `Consumer conprod`, метод `accept` якого обчислює

добуток від результату consum і накопичує його у статичному полі prod. Процес повторюється для аргументів 4 та 5.

Довгі ланцюжки викликів методів споживачів Consumer можна скласти і для виконання операції, яка не вимагає повернення результату. Як і у випадку з іншими функціональними інтерфейсами, складений об'єкт Consumer перших двох операцій стає першим операндом другого складеного об'єкта Consumer тощо. Цей прийом можна, наприклад, для обчислення полінома:

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

Значення полінома для даного x можна знайти, обчисливши доданок четвертого ступеня за допомогою виклику об'єкта Consumer, а потім обчисливши третій ступінь і так далі до постійного доданку, використовуючи об'єкти Consumer, переданих методу andThen у ланцюжку викликів методів Consumer.

```
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
    }
}
```

Остання ланка в ланцюжку викликів методів Consumer містить оператор System.out.println, який друкує результат рівняння. Ланцюжки викликів методів часто закінчуються об'єктом Consumer з термінальною операцією, що друкує результат обробки - опреацією, яка не повертає новий об'єкт Consumer, часто є void.

Java API надає узагальнені спеціалізації інтерфейсу Consumer: IntConsumer, LongConsumer і DoubleConsumer, які обробляють примітивні типи int, long та double, відповідно.

```
@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 та LongConsumer анлогічно
//Приклад використання:
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
}
```

Часто корисно обробляти дані двох різних узагальнених типів, це виконує функціональний інтерфейс `BiConsumer<T,U>`. Його метод `accept` приймає аргументи типів `T` і `U` і є `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);
        };
    }
}
```

Приклад використання інтерфейсу `BiConsumer<T,U>`:

```
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
    }
}
```

Java API надає інтерфейси `ObjIntConsumer`, `ObjLongConsumer` та `ObjDoubleConsumer`, які використовують як другий аргумент методу `apply` примитивні типи `int`, `long` та `double`.

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

     void accept(T t, int value);

}
//ObjLongConsumer та ObjDoubleConsumer аналогічно
//Приклад використання спеціалізованих інтерфейсів:
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
    }
}
```

Функціональний інтерфейс `Supplier<T>` використовується для генерації даних. Його функціональний метод `get` не приймає аргументів і повертає об'єкт типу `T`.

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

    T get();

}
```

Приклад використання інтерфейсу `Supplier<T>` (у деякому класі):

```
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());
}
```

Після запуску програма видасть подібний результат:

```
73
56
Enter a string: Hello
 Hello
Enter a string: World
 World
```

Зверніть увашу на можливість організації в лямбда-виразі функціонального методу об'єкта `Supplier` процедури введення даних.

Java API надає `BooleanSupplier`, `IntSupplier`, `LongSupplier` та `DoubleSupplier`, які є узагальненими спеціалізаціями інтерфейсу `Supplier<T>`. Вони генерують примітивні типи `boolean`, `int`, `long` та `double`, відповідно.

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
//IntSupplier, LongSupplier та DoubleSupplier аналогічно
```

Приклад використання спеціалізованих інтерфейсів `Supplier`:

```
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());
                                        //3868393452067931422
        System.out.println(genDbl.getAsDouble());
                                        //0.9980533053789374
    }
}
```

2. Завдання

1. Оберіть завдання у наведеній нижче таблиці. Номер варіанта визначте за формулою V = (№ mod 22) +1, де № - Ваш порядковий номер в журналі академгрупи.

V	Завдання	V	Завдання
1	Створіть функціональний інтерфейс з ім'ям `InputStreamOpener` з функціональним методом `open`, який приймає аргумент `String` і повертає `InputStream`. Напишіть три реалізації: перша - повинна повертати об'єкт `DataInputStream`, друга - об'єкт `ObjectInputStream`, третя - об'єкт `BufferedInputStream`. Використовуйте анонімні класи для всіх реалізацій. Продемонструйте використання реалізацій в основній програмі.	12	Служба внутрішніх доходів визначає утриманцем особу, яка відповідає всім наведеним нижче умовам стосовно Вас: - Ваш син, дочка, пасинок чи прийомна дитина; - має вік до 19 років та молодший за Вас, або студент віком до 24 років і молодший за Вас, або є довічним інвалідом; - не заробляє більше половини свого грошового забезпечення; - не подавав спільну податкову декларацію або подавав спільну декларацію лише з метою відшкодування витрат; - прожив з Вами більше аніж півроку. Використовуючи єдиний ланцюжок предикатів, визначте, чи є особа утриманцем.
2	Створіть узагальнений функціональний інтерфейс `ListManipulator<T>` з функціональним методом `manipulate`, який приймає `List<T>` і `T` як аргументи і нічого не повертає. `ListManipulator<T>` містить `default` метод `create`, який створює `ArrayList` з трьома рядками. Цей метод не бере аргументів і повертає `List<T>`. Напишіть три реалізації `ListManipulator`: `sListAdd`, яка реалізує `ListManipulator<String>` і функціональний метод якої додає рядок в кінець списку; та `iListAdd`, яка реалізує `ListManipulator<Integer>` і функціональний метод якої додає ціле число в кінець списку; та `iListRmv`, яка реалізує `ListManipulator<Integer>` і функціональний метод якої видаляє перше входження цілого числа зі списку. Реалізації `iListAdd` та `iListRmv` перевизначають `default` метод реалізацією, яка встановлює початкову місткість `ArrayList` до 50 елементів та додає до списку 3 числа. Продемонструйте три реалізації в основній програмі.	13	Колода карт складається з 52 гральних карт (2–10, Валет, Дама, Король та Туз) і кожного з 4 мастей (Чирви, Трефи, Піки та Бубни). Напишіть програму, у якій у єдиному циклі, який використовується біфункція для створення колоди гральних карт. Організуйте перечислення для зберігання назв та мастей карт.

V	Завдання	V	Завдання
3	Створіть узагальнений функціональний інтерфейс з назвою `ToString<T>` з функціональним методом `convert`, який конвертує T в `String`. Напишіть дві реалізації: `l2s` – яка перетворює `List<String>` у рядок елементів-рядків, розділених комами; `m2s` - яка перетворює `Map<String, Integer>`, у рядок елементів ключ-значення, розділених комами, між ключем і значенням повинна бути двокрапка. Продемонструйте дві реалізації в основній програмі	14	Колекція `Map` містить такі записи: ключ - собака, значення - нащадок вовка; ключ - кот, значення - з дев'ятьма життями; ключ - щур, значення - гризун з довгим хвостом. Використовуйте функціональний ланцюжок для перемикання ключів і значень кожного запису. (Підказка: спочатку перетворіть `Map` в список, потім - змініть значення для ключів, а потім розмістіть їх у `Map`).
4	Створіть функціональний інтерфейс `Area` і використайте його для обчислення площі кола, прямокутника та рівнобедреного прямокутного трикутника. Включіть до інтерфейсу метод `numberOfSides`, який за замовчуванням повертає 4.	15	Напишіть оператор, який повертає рядок, що містить непарні символи вхідного рядка (іншими словами, символи, розташовані в позиціях 1, 3, 5 тощо).
5	Ціна вживаних автомобілів обчислюється як початкова ціна мінус 1000 доларів на рік з моменту виготовлення, мінус 500 доларів за кожні 10 000 миль пробігу. Ціна вживаних спортивних автомобілів додає до початкової ціни 250 доларів за кожен рік з моменту виготовлення. Створіть функціональний інтерфейс `ListValue`, який обчислює значення цін для списку, у який входять як типові автомобілі, так і спортивні автомобілі.	16	Клас A містить поле `int first` та `double second`. Напишіть `BinaryOperator`, який повертає об'єкт класу A зі значенням поля `first`, яке дорівнює сумі значень полів `first` двох об'єктів класу A, та значенням поля `second`, яке дорівнює різниці значень полів `second` цих об'єктів. Перевірте роботу програми для двох об'єктів класу A з довільними значеннями полів.
6	Створіть узагальнений функціональний інтерфейс `Summer` і використайте його для обчислення суми двох `Integer` чисел, суми двох `Double` та суми двох `Long`.	17	Для заданого рядка "The fault lies not from our stars", розробіть єдиний ланцюжок операторів для виконання наступних операцій: - об'єднайте рядок з " but from ourselves."; - замініть "from" на "in". - змініть регістр рядка на верхній.
7	Створіть функціональний інтерфейс `SumFromZero`, функціональний метод якого приймає один аргумент і повертає суму всіх чисел від нуля до аргументу включно. Продемонструйте дві його реалізації в основній програмі	18	Словник реалізований як Map ключів і значень: `Map<String, String> dictionary = Map.of("quest", "Be or not to be?", "answer", "It's Only Love", "route", "But there is no end of the road")`. Використовуючи `Consumer`, створіть рядок, що містить розділений комами список

V	Завдання	V	Завдання
			перших слів кожного значення пари, довжина ключа якої становить п’ять символів.
8	Напишіть предикат, який перевіряє, чи містить рядок лише цифри. Продемонструйте в основній програмі	19	Колекція складається з: - DVD-дисків, упорядкованих за назвою, компанією-виробником та об’ємом пам'яті в мегабайтах; - аудіофайли, упорядковані за назвою, форматом та об’ємом пам'яті в мегабайтах; - електронні книги, упорядковані за автором, назвою, кількістю сторінок та обсягом пам’яті в мегабайтах. За допомогою ланцюжка викликів методів `Consumer` створити список усіх назв колекції та інший список усіх обсягів пам'яті колекції.
9	Напишіть предикат, який реалізує такий логічний вираз. NOT(X < 100 OR X is odd) AND X > 20 Продемонструйте в основній програмі	20	Багатоквартирний будинок містить двокімнатні та трикімнатні квартири різних розмірів. За допомогою інтерфейсу `Consumer` обчисліть загальну квадратуру всіх кімнат у будівлі
10	Використовуючи інтерфейс `BiPredicate`, реалізуйте наступний логічний вираз і продемонструйте в головній програмі: NOT(X > 2 AND Y < X)	21	За допомогою об’єктів `Supplier` створіть словник, що зберігається у `Map`. Кожна пара у якості ключа має слово, а у якості значення - визначення цього слова. Напишіть `Supplier`, який запропонує користувачеві ввести слово. Якщо слово вже є у словнику, повинно бути виведене повідомлення про помилку. Напишіть інший `Supplier`, який який запропонує користувачеві ввести визначення слова. Напишіть третього постачальника, який запитає користувача, чи він закінчив вводити слова. Після завершення роботи користувача роздрукуйте вміст словника.
11	Використовуючи єдиний ланцюжок предикатів, визначте, чи є п’ятибуквене слово паліндромом. Паліндром – це слово, яке пишеться однаково і вперед, і назад. Наприклад, “kayak” – паліндром, тоді як “apple” - ні.	22	За допомогою функціонального інтерфейсу `Supplier` створіть програму програму для опитування покупців щодо задоволення придбаним товаром та якістю обслуговування. Питання опитування та місце для відповідей зберігайте у колекції Map<String, String> survey. Передбачте організацію введення відповідей з консолі об'єктом Supplier<String>, який повинен виводити повідомлення у разі введення непередбачуваних відповідей та

V	Завдання	V	Завдання
			пропонувати введення ще раз та дозволяти переривати опитування у будь-який момент.

2. Розробіть два варіанти програми: перший для реалізації функціональних методів повинен використовувати внутрішні анонімні класи, а другий - лямбда-вирази. При можливості у другому варіанті використовуйте посилання на методи класів. Порівняйте оба варіанта та визначте переваги кожного.
3. Наведіть код обох варіантів програми у звіт.

3. Контрольні питання

1. Чим відрізняється функціональне програмування від об’єктно-орієнтованого. Назвіть переваги, що надає функціональне програмування.
2. Що називають функціональним інтерфейсом? Чи може фунціональний інтерфейс містити методи окрім абстрактного?
3. Приведіть відомі Вам методи створення об’єкту класу, що реалізує функціональний інтерфейс.
4. Що називають лямбда-виразом, як він пов’язаний з функціональним інтерфейсом? Поясніть синтаксис лямбда-виразу.
5. Приведіть приклад функціонального інтерфейсу та відповідного лямбда-виразу для функціонального метода `void`. Запишіть лямбда-вираз через посилання на метод, що визивається.
6. Поясніть синтаксис лямбда-виразів для функціонального методу з декількома аргументами та для функціонального методу без аргументів.
7. Опишіть базові моделі функціональних інтерфейсів, які використовує Java API.
8. Опишіть функціональний інтерфейс `Predicate<T>` та наведіть приклад його використання.
9. Наведіть приклад передачі предиката як аргумента методу.
10. Опишіть методи функціонального інтерфейсу `Predicate<T>`, що дозволяють створювати ланцюжки виклику методів, та наведіть приклад такого ланцюжку.
11. Які спеціалізовані предикати надає Java API? Приведіть приклад їх використання.
12. Опишіть функціональний інтерфейс `BiPredicate<T, U>` та наведіть приклад його використання.
13. Опишіть функціональний інтерфейс `Function<T, R>` та наведіть приклад його використання.
14. Опишіть методи функціонального інтерфейсу `Function<T, R>`, що дозволяють створювати ланцюжки виклику методів, та наведіть приклад такого ланцюжку.
15. Які спеціалізовані функції надає Java API? Приведіть приклад їх використання.

16. Опишіть функціональний інтерфейс `BiFunction<T, U, R>` та наведіть приклад його використання.
17. Опишіть методи функціонального інтерфейсу `BiFunction<T, U, R >`, що дозволяють створювати ланцюжки виклику методів, та наведіть приклад такого ланцюжку.
18. Які спеціалізовані бінарні функції надає Java API? Приведіть приклад їх використання.
19. Опишіть функціональний інтерфейс `UnaryOperator<T>` та наведіть приклад його використання.
20. Які спеціалізовані унарні оператори надає Java API? Приведіть приклад їх використання.
21. Опишіть функціональний інтерфейс `BinaryOperator<T>` та наведіть приклад його використання.
22. Які додаткові методи бінарних операторів надає Java API? Приведіть приклад їх використання.
23. Які спеціалізовані бінарні оператори надає Java API? Приведіть приклад їх використання.
24. Опишіть функціональний інтерфейс `Consumer<T>` та наведіть приклад його використання.
25. Як можна побудувати ланцюжок викликів методів `Consumer<T>`? Наведіть приклад такого ланцюжка. Що називають термінальною операцією ланцюжка?
26. Які спеціалізовані консьюмери надає Java API? Приведіть приклад їх використання.
27. Опишіть функціональний інтерфейс `BiConsumer<T,U>` та наведіть приклад його використання.
28. Опишіть функціональний інтерфейс `Supplier<T>` та наведіть приклад його використання.
29. Які спеціалізовані сапплайери надає Java API? Приведіть приклад їх використання.