Лабораторна робота № 5

Функціональне програмування при роботі зі структурами Java та потоками даних

Мета роботи: Опанувати технології та засоби функціонального програмування для ітерації масивів та колекцій, роботи з колекціями, компараторами, об’єктами `Optional<T>` та потоками даних `Stream<T>`.

1. Теоретичні відомості

Обхід елементів масивів та колекцій є частою дією при програмуванні. Обхід таких структур даних був спрощений завдяки використанню об’єктів `Consumer<T>`, які можуть бути використані для заміни циклів, пов'язаних з ітераторами.

Наприклад, є клас `Car`:

```
public class Car {
    private String make;
    private String model;
    public Car(String ma, String mo) {
        make = ma;
        model = mo;
    }
    @Override
    public String toString() {
        return make + " " + model;
    }
}
```

Java API забезпечує класичний підхід для обходу об'єктів-елементів колекцій - інтерфейс `Iterator<E>` з його абстрактними методами `boolean hasNext()` та `E next()`. В Java 8 до інтерфейсу `Iterator<E>` був доданий метод `default void forEachRemaining(Consumer<? super E> action)` для обходу елементів колекцій (також був доданий метод `default void remove()` - для видалення елементів колекцій).

```
public static void main(String[] args) {
    List<Car> cars = Arrays.asList(
            new Car("Nissan", "Sentra"),
            new Car("Chevrolet", "Vega"),
            new Car("Hyundai ", "Elantra")
    );
    /*Класичний обхід*/
    Iterator<Car> it = cars.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
    /*Обхід з використанням лямбда-виразу для метода
      void accept(E) інтерфесу Consumer<E>*/
```

```
    cars.iterator().forEachRemaining(x ->
                                        System.out.println(x));
     /*Лямбда-вираз - використання посилання на метод*/
    cars.iterator().forEachRemaining(System.out::println);
}
```

Метод `forEachRemaining` фактично реалізує цикл `while` по колекції:

```
default void forEachRemaining(Consumer<? super E> action) {
    Objects.requireNonNull(action);//перевірка action на null
    while (hasNext())
        action.accept(next());
}
```

Виклик метода `forEachRemaining` інтерфейса `Iterator<E>` можливий для колекцій або об'єктів, які реалізують інтерфейс `Iterable<E>`. У випадку масивів певних примітивних типів Java API пропонує інтерфейс `PrimitiveIterator<T, T_CONS> extends Iterator<T>`. Він узагальнений для двох типів, `T` і `T_CONS`. Тип `T` може бути `Integer`, `Long` або `Double`, а тип `T_CONS` повинен бути спеціалізацією інтерфейса `Consumer` для вказаних примітивних типів - `IntConsumer`, `LongConsumer` або `DoubleConsumer`, відповідно. Інтерфейс `PrimitiveIterator` має метод `void forEachRemaining(T_CONS action)`, що приймає як параметр відповідну спеціалізацію інтерфейса `Consumer` для вказаних примітивних типів:

```
public interface PrimitiveIterator<T, T_CONS>
                                        extends Iterator<T> {
    void forEachRemaining(T_CONS action);
}
```

Наприклад, для ітерації масиву `int` створимо клас-реалізацію `PrimitiveIterator`, який "огортає" масив `int`. Оскільки `PrimitiveIterator` успадковує `Iterator`, потрібна реалізація абстрактних методів останнього:

```
public class IntIteratorGen
         implements PrimitiveIterator<Integer, IntConsumer> {
    private int[] array;   //огорнений масив
    private int cursor;
    public IntIteratorGen(int... a) {
        cursor = 0;
        array = Arrays.copyOf(a, a.length);
    }
    /*Ітерація масиву*/
    @Override
    public void forEachRemaining(IntConsumer c) {
        while(hasNext()) {
            c.accept(array[cursor]);
            cursor++;
        }
```

```
    }


    /*Реалізація абстрактних методів Iterator<E>*/
    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }
    @Override
    public Integer next() {
        int i = 0;
        if (hasNext()) {
            i = array[cursor];
            cursor++;
        }
        return i;
    }
}
```

Тепер можна використовувати спроектований клас для ітерації масивів `int` з використанням функціонального програмування:

```
//у деякому класі
public static void main(String[] args) {
    IntIteratorGen intItGen =
             new IntIteratorGen(1, 2, 3, 4, 5);
    intItGen.forEachRemaining((IntConsumer)x ->
                          System.out.print(x + ",”));
}
```

`Consumer` потрібно привести до `IntConsumer`, щоб уникнути неоднозначного посилання, оскільки метод `forEachRemaining` інтерфейсу `Iterator` приймає `Consumer<E>`.

Неузагальнені спеціалізації інтерфейсу `PrimitiveIterator` для `Integer`, `Long` і `Double` доступні як вкладені інтерфейси `PrimitiveIterator`.

```
public static interface PrimitiveIterator.OfInt
            extends PrimitiveIterator<Integer, IntConsumer> {
    default void forEachRemaining(IntConsumer action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(nextInt());
    }
@Override
    default Integer next() {
        if (Tripwire.ENABLED)
            Tripwire.trip(getClass(),
              "{0} callingrimitiveIterator.OfInt.nextInt()");
            return nextInt();
```

```
    }
    int nextInt();
    ...
}
```

Вкладені інтерфейси `PrimitiveIterator.OfLong` and `PrimitiveIterator.OfDouble` спроектовані аналогічно. Методи `forEachRemaining` та `next` у цих інтерфейсах є `default` методами і вони реалізовані, тому потрібно визначити лише методи `hasNext`, `nextInt` (`nextLong` і `nextDouble`).

Класи `IntIterator`, `LongIterator` та `DoubleIterator`, визначені наступним чином, можуть використовуватися для обходу масивів Java примітивних типів `int`, `long` та `double`, відповідно.

```
public class IntIterator implements PrimitiveIterator.OfInt {
    private int[] array;
    private int cursor;

    public IntIterator(int... a) {
        cursor = 0;
        array = Arrays.copyOf(a, a.length);
    }

    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }

    @Override
    public int nextInt() {
        int i = 0;
        if(hasNext()){
            i = array[cursor];
            cursor++;
        }
        return i;
    }
}
```

Класи `LongIterator` та `DoubleIterator` визначаються аналогічно. Наступна програма демонструє використання класів для ітерації масивів примітивних типів.

```
public static void main(String[] args) {
    IntIterator iit = new IntIterator(1, 2, 3, 4, 5);
    iit.forEachRemaining((IntConsumer) x ->
                                    System.out.print(x + " "));
    System.out.println();
    LongIterator lit = new LongIterator(6, 7, 8, 9, 10);
    lit.forEachRemaining((LongConsumer) x ->
                                    System.out.print(x + " "));
```

```
    System.out.println();
    DoubleIterator dit = new DoubleIterator(20.1, 21.2, 22.3,
                                            23.4, 24.5);
    dit.forEachRemaining((DoubleConsumer) x ->
                                    System.out.print(x + " "));
    System.out.println();
}
```

Використання вкладених інтерфейсів дещо спрощує ітерацію масивів примітивних типів `int`, `long` та `double` порівняно з використанням батьківського інтерфейсу `PrimitiveIterator`. Але останній може бути використаний для ітерації масивів інших примітивних типів (через реалізацію відповідного консьюмера).

Інтерфейс `Spliterator<T>` визначає метод `default forEachRemaining(Consumer<? super T> action)`, який приймає об'єкт класу, що реалізує функціональний інтерфейс `Consumer`. Сплітератор корисний для розділення колекції на частини за допомогою методу `Spliterator<T> trySplit()` з подальшою обробкою їх різними потоками виконання (thread) одночасно.

```
public interface Spliterator<T> {
    default void forEachRemaining(Consumer<? super T>
                                                    action);
    Spliterator<T> trySplit();
    ...
}
```

Приклад використання інтерфейсу `Spliterator<T>`:

```
public static void main(String[] args) {
    List<Car> cars = Arrays.asList(
        new Car("Nissan", "Sentra"),
        new Car("Chevrolet", "Vega"),
        new Car("Hyundai", "Elantra"),
        new Car("Buick", "Regal")
    );
    /*Обхід усього списку з а допомогою Spliterator*/
    Spliterator<Car> spliterator = cars.spliterator();
    spliterator.forEachRemaining(x ->
              System.out.println("In splititerator: " + x));
    /*Для повторного обходу потрібно ще раз отримати
      Spliterator*/
    spliterator = cars.spliterator();
    Spliterator<Car> firstHalf = spliterator.trySplit();
    firstHalf.forEachRemaining(x ->
                    System.out.println("In 1st half: " + x));
    spliterator.forEachRemaining(x ->
                    System.out.println("In 2nd half: " + x));
}
```

У Java 8 до інтерфейсу `Iterable<E>` був доданий метод `forEach(Consumer<? super T> action)`:

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

Цей метод зручний для обходу будь-яких `Iterable<E>` об'єктів, наприклад, колекцій:

```
public static void main(String[] args) {
    List<Car> cars = Arrays.asList(
            new Car("Nissan", "Sentra"),
            new Car("Chevrolet", "Vega"),
            new Car("Hyundai", "Elantra")
    );
    /*Обхід колекції*/
    cars.forEach(x -> System.out.println(x));
    /*Те ж саме через посилання на метод*/
    cars.forEach(System.out::println);
}
```

У виподку, якщо об'єкт `Iterable<T>` містить масив `int`, `long` або `double`, метод `Iterator<T> iterator()` такого об'єкта може повертати `PrimitiveIterator<T, T_CONS>` (тут `T` - огортка відповідного примитивного типу), наприклад:

```
public class MyInts implements Iterable<Integer> {
    private int[] array;
    public MyInts(int... a) {
        array = Arrays.copyOf(a, a.length);
    }
    @Override
    public PrimitiveIterator<Integer,IntConsumer> iterator(){
        return new IntIter();
    }
    private class IntIter
         implements PrimitiveIterator<Integer, IntConsumer> {
        private int cursor;
        public IntIter() {
            cursor = 0;
        }
        @Override
        public void forEachRemaining(IntConsumer c) {
            while (hasNext()) {
```

```
                c.accept(array[cursor]);
                cursor++;
            }
        }

        @Override
        public boolean hasNext() {
            return cursor < array.length;
        }

        @Override
        public Integer next() {
            int i = 0;
            if (hasNext()) {
                i = array[cursor];
                cursor++;
            }
            return i;
        }
    }
}
```

Тоді використання цього класу для обходу масиву може мати вигляд:

```
public static void main(String[] args) {
    MyInts my = new MyInts(1, 2, 3, 4, 5);
    my.forEach(x -> System.out.println(x));
    System.out.println();
    /*Можна використовувати ітератор явно*/
    my.iterator()
    .forEachRemaining((IntConsumer) x ->
                                        System.out.println(x));
}
```

Колекції `Map<K, V>` не є `Iterable<E>` об’єктами. Тим не менш, для їх обходу передбачений власний метод `forEach`. На відміну від об'єктів `Iterable<E>`, метод `forEach` яких приймає об’єкт `Consumer`, метод `forEach` інтерфейсу `Map<K, V>` приймає об'єкт `BiConsumer`, який обробляє ключ і значення запису:

```
default void forEach(BiConsumer<? super K,? super V> action){
    Objects.requireNonNull(action);
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch (IllegalStateException ise) {
            //зазвичай, коли записів більше немає
            throw new ConcurrentModificationException(ise);
```

```
        }
        action.accept(k, v);
    }
}
```

Приклад обходу колекції `Map<K, V>`:

```
public static void main(String[] args) {
    Map<String, Double> employeeSalaries = new TreeMap<>();
    employeeSalaries.put("Joe Smith", 100000.0);
    employeeSalaries.put("Maggie Jones", 110000.0);
    employeeSalaries.put("Larry Rodriguez", 105000.0);
    /*forEach method accepts a BiConsumer */
    employeeSalaries.forEach((x, y) -> System.out.println(x
                        + " makes $" + y + " annually."));
    }
```

Після запуску програма виведе:

```
Joe Smith makes $100000.0 annually.
Larry Rodriguez makes $105000.0 annually.
Maggie Jones makes $110000.0 annually.
```

Оскільки колекція `Set<E>` є `Iterable<E>` об’єктом, то її можна обійти як за допомогою методу `forEachRemaining` ітератора колекції `Iterator<E>`, так і методом `forEach` інтерфейсу `Iterable<E>`.

```
public static void main(String[] args) {
    Set<String> colors = new TreeSet<>();
    colors.add("red");
    colors.add("green");
    colors.add("blue");
    colors.iterator().forEachRemaining(x ->
                                    System.out.print(x + " "));
    System.out.println();
    colors.forEach(x -> System.out.print(x + " "));
    System.out.println();
}
```

Функціональні інтерфейси можна використовувати для виконання декількох важливих операцій над колекціями. Метод `removeIf` інтерфейса `Collection<E>` можна використовувати для видалення елементів з об'єкта `Collection`, ітератор якого підтримує операцію видалення (`List, Set`).

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
```

```
        }
    }
    return removed;
}
```

Якщо метод `test` предиката, наданий методу `removeIf`, повертає `true` для елемента колекції, цей елемент видаляється з колекції. Наприклад:

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("Super");
    list.add("Random");
    list.add("Silly");
    list.add("Strings");
    list.removeIf(x -> x.startsWith("S"));
    list.forEach(x -> System.out.println(x));      //Random
    Set<String> names = new TreeSet<>();
    names.add("Jeremy");
    names.add("Javier");
    names.add("Rose");
    names.removeIf(x -> x.charAt(0) == 'J');
    names.forEach(x -> System.out.println(x));      //Rose
}
```

Клас `java.utils.Arrays` містить кілька перевантажених методів `setAll`, які додають до масива, переданого як перший аргумент, елементи, які надають оператор або функція, передані як другий аргумент. Вхідним параметром функціонального метода оператора або функції є індекс елемента масива, повертає метод сам елемент, який буде доданий до масиву за вказаним індексом:

```
static void setAll(int[] array, IntUnaryOperator generator)
static void setAll(long[] array, IntToLongFunction generator)
static void setAll(double[] array,
                                IntToDoubleFunction generator)
static <X> void setAll(X[] array, IntFunction<? extends X>
                                                    generator)
```

Наведемо приклад використання цих методів:

```
public static void main(String[] args) {
    /*Додає до масиву елементи, значення яких дорівнює
      їх індексу*/
    IntUnaryOperator iop = x -> x;
    int[] arr = new int[4];
    Arrays.setAll(arr, iop);
    for (int i : arr) {
        System.out.println(i);
    }
    /* Додає до масиву елементи, значення яких дорівнює 5*/
    IntToLongFunction gen5 = x -> 5;
```

```
    long[] larr = new long[4];
    Arrays.setAll(larr, gen5);
    for (long l : larr) {
        System.out.println(l);
    }
    /*Додає до масиву double елементи з випадковими
      значеннями від 0.0 до 1.0*/
    IntToDoubleFunction i2d = x ->
                                    (new Random()).nextFloat();
    double[] darr = new double[4];
    Arrays.setAll(darr, i2d);
    for (double d : darr) {
        System.out.println(d);
    }
    /*Додає до масиву рядки з символу "S", який повторюється
      значення індексу елемента раз*/
    IntFunction<String> is = x -> {
        String s = "";
        for (int i = 0; i <= x; i++) {
            s += "S";
        }
        return s;
    };
    String[] sarr = new String[4];
    Arrays.setAll(sarr, is);
    for (String s : sarr) {
        System.out.println(s);
    }
}
```

Усі елементи у списку можуть бути змінені за допомогою `default` методу `replaceAll` та `UnaryOperator` інтерфейсу List<E>, який вказує, як виконати модифікацію.

```
default void replaceAll(UnaryOperator<E> operator) {
    Objects.requireNonNull(operator);
    final ListIterator<E> li = this.listIterator();
    while (li.hasNext()) {
        li.set(operator.apply(li.next()));
    }
}
```

Аналогічно, усі елементи `Map` також можуть бути змінені за допомогою `default` методу `replaceAll` за замовчуванням та `BiFunction`, яка вказує, як виконати модифікацію.

```
default void replaceAll(BiFunction<? super K, ? super V,
                                    ? extends V> function) {
    Objects.requireNonNull(function);
```

```
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch (IllegalStateException ise) {
            throw new ConcurrentModificationException(ise);
        }
        v = function.apply(k, v);
        try {
            entry.setValue(v);
        } catch (IllegalStateException ise) {
            throw new ConcurrentModificationException(ise);
        }
    }
}
```

Наведемо приклад такої заміни:

```
public static void main(String[] args) {

    List<Integer> list = Arrays.asList(16, 12, 8, 4);
    UnaryOperator<Integer> div4 = x -> x / 4;
    list.replaceAll(div4);
    list.forEach(x -> System.out.print(x + " "));
    System.out.println();

    Map<String, String> map = new TreeMap<>();
    map.put("Smith", "Robert");
    map.put("Jones", "Alex");
    BiFunction<String, String, String> bi = (k,v) ->
                                              "Mr. " + v;
    map.replaceAll(bi);
    map.forEach((k,v) -> System.out.println(v + " " + k));
}
```

Після запуску програма видає:

```
4 3 2 1
Mr. Alex Jones
Mr. Robert Smith
```

Обчислення масивів можна виконувати паралельно, щоб пришвидшити роботу програм, якщо це не змінює результати обчислень. Клас `java.utils.Arrays` має кілька перевантажених методів `parallelPrefix`, які виконують паралельні обчислення елементів масиву або піддіапазону елементів масиву. Методи приймають як аргументи масив і бінарний оператор, який визначає обчислення, яке потрібно виконати.

```
static void parallelPrefix(double[] array, int fromIndex,
                           int toIndex, DoubleBinaryOperator op);
static void parallelPrefix(double[] array,
```

```
                                          DoubleBinaryOperator op);
static void parallelPrefix(int[] array, int fromIndex,
                           int toIndex, IntBinaryOperator op);
static void parallelPrefix(int[] array,
                                            IntBinaryOperator op);
static void parallelPrefix(long[] array, int fromIndex,
                          int toIndex, LongBinaryOperator op);
static void parallelPrefix(long[] array,
                                           LongBinaryOperator op);
static void parallelPrefix(T[] array, int fromIndex,
                           int toIndex, BinaryOperator<X> op);
static void parallelPrefix(T[] array, BinaryOperator<X> op);
```

Наступний приклад визначає масив `int`. `IntBinaryOperator` визначає множення двох елементів масиву, що виконується паралельно.

```
public static void main(String[] args) {
    int[] arr = {2,3,4,3};
    IntBinaryOperator op = (x,y) -> x*y;
    Arrays.parallelPrefix(arr, op);
    for (int i : arr) {
        System.out.print(i + " ");    //2 6 24 72
    }
}
```

В Java 8 з'явився Stream API, який, на жаль не включає в себе `Map`-и. Проте, в `Map` було додано кілька корисних методів, зокрема, методи, що дозволяють виконувати обчислення на записах `Map`. Інтерфейс `Map` забезпечує наступні `default` методи, які виконують обчислення для запису, використовуючи вказану `BiFunction`. Якщо функція призводить до нуля, запис видаляється.

```
default V compute<K key, BiFunction<? super K, ? super V,
                                 ? extends V> remappingFunction);
default V computeIfAbsent<K key, Function<? super K,
                                   ? extends V> mappingFunction);
default V computeIfPresent<K key, BiFunction<? super K,
                  ? super V, ? extends V> remappingFunction);
```

Наприклад:

```
public static void main(String[] args) {

    Map<String, Integer> map = new TreeMap<>();
    map.put("RED", 32);
    map.put("GREEN", null);

    /* Метод compute виконує для записів обчислення, визначені об'єктом BiFunction. Якщо BiFunction повертає null, запис видаляється */
    BiFunction<String, Integer, Integer> bin = (k, v) ->
                                        v == null ? null : v / 4;
    System.out.println(map.compute("RED", bin));    //8
```

```
        System.out.println(map.compute("GREEN", bin)); //null
                                                  //запис видаляється
        System.out.println(map.compute("YELLOW", bin)); //null
                                                //нічого не робиться
        map.forEach((k, v) -> System.out.print(k + " " + v
                                                  + ", ")); //RED 8
        System.out.println();
        /*Метод computeIfPresent виконує обчислення для записів,
визначені об'єктом BiFunction, якщо запис існує і його значення
не null (для значення null - нічого не робиться). Якщо
BiFunction повертає null, запис видаляється.
        Відновлення видаленого запису*/
        map.put("GREEN", null);
        BiFunction<String, Integer, Integer> bi = (k, v) -> v/4;
        System.out.println(map.computeIfPresent("RED", bin)); //2
        System.out.println(map.computeIfPresent("GREEN", bin));
                                        //null - нічого не робиться
        System.out.println(map.computeIfPresent("YELLOW", bin));
                                        //null - нічого не робиться
        map.forEach((k, v) -> System.out.print(k + " " + v
                                    + ", ")); //GREEN null, RED 2,
        System.out.println();
        /* Метод computeIfAbsent виконує обчислення для записів,
визначені об'єктом Function, якщо запис відсутній або якщо зна-
чення записи null. Якщо функція повертає null, запис не дода-
ється. Метод приймає об'єкт Function замість BiFunction, оскі-
льки жодне існуюче значення не потрібно обробляти */
        Function<String, Integer> fi = k -> k.length();
        Function<String, Integer> finull = k -> null;
        System.out.println(map.computeIfAbsent("RED", fi));
                                          //2 - нічого не робиться
        System.out.println(map.computeIfAbsent("GREEN", fi));
                                  //5 (кількість символів у ключі)
        System.out.println(map.computeIfAbsent("YELLOW", fi));
                                  //6 (кількість символів у ключі)
        System.out.println(map.computeIfAbsent("BLACK", finull));
                                           //null (запис не додається
                                 //оскільки функція повернула null)
        map.forEach((k, v) -> System.out.print(k + " " + v
                               + ", ")); //GREEN 5, RED 2, YELLOW 6,
        System.out.println();
}
```

Метод `merge` інтерфейсу `Map` використовується для модифікації значень частини записів шляхом встановлення нових значень відповідно до функції зіставлення. Якщо запис не існує, створюється новий запис із зазначеним ключем та

значенням. Якщо функція зіставлення повертає `null`, запис видаляється з карти. Нове значення не може бути `null`.

```
default V merge(K key, V value, BiFunction<? super V,
                  ? super V,? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    Objects.requireNonNull(value);
    V oldValue = get(key);
    V newValue = (oldValue == null) ? value :
               remappingFunction.apply(oldValue, value);
    if (newValue == null) {
        remove(key);
    } else {
        put(key, newValue);
    }
    return newValue;
}
```

Розглянемо застосування цього методу на наступному прикладі. Даний клас:

```
public class MyClass {
    int i1;
    int i2;
    String s;
    public MyClass(int x, int y, String z) {
        i1 = x;
        i2 = y;
        s = z;
    }
    @Override
    public String toString() {
        return i1 + " " + i2 + " " + s;
    }
}
//У деякому класі
public static void main(String[] args) {
    Map<String, MyClass> m = new TreeMap<>();
    m.put("k1", new MyClass(1, 2, "Dog"));
    BiFunction<MyClass, MyClass, MyClass> changeI2 = (ov, nv)
                          -> new MyClass(ov.i1, nv.i2, ov.s);
    BiFunction<MyClass, MyClass, MyClass> changeS = (ov, nv)
                          -> new MyClass(ov.i1, ov.i2, nv.s);
    System.out.println(m.merge("k1", new MyClass(0, 5, null),
                                        changeI2));   //1 5 Dog
    System.out.println(m.merge("k1", new MyClass(0, 0,
                                "Cat"), changeS));   //1 5 Cat
    System.out.println(m.merge("k2", new MyClass(6, 7,
                               "Bird"), changeS));   //6 7 Bird
    m.forEach((k, v) -> System.err.print(k + " " + v
```

```
                                + ", ")); //k1 1 5 Cat, k2 6 7 Bird,
}
```

`BiFunction changeI2` та метод `merge` змінюють значення поля `i1` запису `"k1"` на `5`. `BiFunction changeS` і метод `merge` змінюють значення поля `s` запису `"k1"` на `"Cat"`. Запис `"k2"` не існує, тому створюється новий запис із зазначеним значенням без використання функції зіставлення.

Оскільки інтерфейс `Set` підтримує операцію видалення, метод `removeIf` можна використовувати для видалення його елементів, якщо виконується відповідна умова, передана предикатом.

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
//У деякому класі
public static void main(String[] args) {
    Set<String> names = new TreeSet<>();
    names.add("Jeremy");
    names.add("Javier");
    names.add("Rose");

    names.removeIf(x -> x.charAt(0) == 'J');
    names.forEach(System.out::println);              //Rose
}
```

Функціональні інтерфейси змінили спосіб порівняння даних у Java завдяки вдосконаленню інтерфейсу `Comparator` шляхом додання декількох нових методів. `Comparator<T>` - це функціональний інтерфейс, який використовується для порівняння двох об'єктів типу `T`. Його функціональний метод `int compare(T o1, T o2)`.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

Результат методу порівняння такий:
позитивне ціле число, якщо `o1 > o2`
від'ємне ціле число, якщо `o1 < o2`
0, якщо `o1 = o2`.

Розглянемо приклад компаратора з використанням функціоналного програмування:

```
public class Comparing {
    public static String removeVowels(String s) {
        return s.replaceAll("[aeiou]", "");
    }
    public static void main(String[] args) {
        Comparator<String> byConsonants = (x, y)
               -> removeVowels(x).compareTo(removeVowels(y));
        System.out.println(byConsonants.compare("Larry",
                                        "Libby"));        //16
        Comparator<Integer> byIntCmpareTo = (x, y) ->
                                              x.compareTo(y);
        System.out.println(byIntCmpareTo.compare(1000,
             1002));     //-1 (Integer compareTo realization)
        Comparator<Integer> byIntDifference = (x, y) ->
                                         x - y;
        System.out.println(byIntDifference.compare(1000,
                                          1002));         //-2
    }
}
```

До `Comparator<T>` в Java 8 були додані додатково такі методи:

```
static<T extends Comparable<? super T>> Comparator<T>
                                                  naturalOrder();
static<T extends Comparable<? super T>> Comparator<T>
                                                  reverseOrder();
static <T> Comparator<T> nullsFirst(Comparator<? super T>
                                                    comparator);
static <T> Comparator<T> nullsLast(Comparator<? super T>
                                                    comparator);
default Comparator<T> reversed();

static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> keyExtractor);

static <T, U> Comparator<T> comparing(Function<? super T,
? extends U> keyExtractor, Comparator<? super U> comparator);
```

Розглянемо приклади використання методів:

```
public static void main(String[] args) {
    /*Порівнює два рядка, використовуючи природний порядок,
      який для об'єктів String лексикографічно порівнює кожен
      символ. Наступний приклад відображає -8, оскільки "a"
      передує "i" в алфавіті на вісім літер.*/
    Comparator<String> natural = Comparator.naturalOrder();
    System.out.println(natural.compare("Larry", "Libby"));
                                                        //-8
```

```
    Comparator<String> reversed = Comparator.reverseOrder();
    System.out.println(reversed.compare("Larry", "Libby"));
                                                          //8
    /*Для запобігання NullPointerException при передачі null
     методу порівняння, використовуйте метод nullsFirst, який
     створює новий компаратор, що обробляє null як менший,
     ніж non-null об'єкти*/
    Comparator<String> byConsonants = (x, y)
            -> removeVowels(x).compareTo(removeVowels(y));
    System.out.println(Comparator.nullsFirst(byConsonants)
                              .compare("Larry", null));     //1
    /*Обробляє null як більший, ніж non-null об'єкти */
    System.out.println(Comparator.nullsLast(byConsonants)
                              .compare("Larry", null));    //-1
    /*Порівнює в зворотному порядку */
    System.out.println(byConsonants.reversed()
                    .compare("Larry", "Libby"));       //-16
    }
}
```

Метод `reverseOrder` створює новий компаратор, який порівнює в зворотному порядку природного впорядкування. Метод `reversed` бере існуючий компаратор і змінює порядок його порівняння на зворотний.

Для розгляду методів `comparing`, створимо клас сутності:

```
public class Student {

    String name;
    Integer id;
    Double gpa;

    public Student(String n, int i, double g) {
        name = n;
        id = i;
        gpa = g;
    }

    @Override
    public String toString() {
        return name + " " + id + " " + gpa;
    }
}
```

та клас-оболонку списку:

```
public class ListWrapper {
    List<Integer> list;
    public ListWrapper(Integer... i) {
        list = Arrays.asList(i);
    }
}
```

Тоді, у деякому коді:

```
public static void main(String[] args) {
    Student s1 = new Student("Larry", 1000, 3.82);
    Student s2 = new Student("Libby", 1001, 3.76);
    /*Порівняння студентів по середньому балу*/
    Function<Student, Double> gpaKey = x -> x.gpa;
    Comparator<Student> byGpa=Comparator.comparing(gpaKey);
    System.out.println(byGpa.compare(s1, s2));              //1
    /*Порівняння студентів по id*/
    Comparator<Student> byId = Comparator.comparing(x->x.id);
    System.out.println(byId.compare(s1, s2));              //-1
    /*Порівняння студентів по імені*/
    Comparator<Student> byName = Comparator.comparing(x ->
                                                    x.name);
    System.out.println(byName.compare(s1, s2));            //-8
    /*Порівняння студентів по імені враховуючи тільки
      приголосні*/
    Comparator<Student> byNameConsonants =
            Comparator.comparing(x -> x.name, (x, y) ->
                removeVowels(x).compareTo(removeVowels(y)));
    System.out.println(byNameConsonants.compare(s1, s2));//16
    /*Порівняння студентів по середньому балу, скругленому
      вгору*/
    Comparator<Student> byGpaCeil = Comparator.comparing(x ->
       x.gpa, (x, y) -> (int) (Math.ceil(x) - Math.ceil(y)));
    System.out.println(byGpaCeil.compare(s1, s2));         //0
                          //(оскільки і для 3.82, і для 3.76
                          //скруглене значення дорівнює 4.0)
    ListWrapper list1 = new ListWrapper(2, 4, 6);
    ListWrapper list2 = new ListWrapper(1, 3, 5);
    /*ERROR: list not comparable*/
//  Comparator<ListWrapper> byList = Comparator.comparing(x->
                                                    x.list);
    Comparator<List<Integer>> byElement0 = (x, y) ->
                                  x.get(0).compareTo(y.get(0));
    Comparator<ListWrapper> byList = Comparator.comparing(x->
                                          x.list, byElement0);
    System.out.println(byList.compare(list1,list2));       //1
}
```

API Java забезпечує спеціалізації методів порівняння інтерфейсу `Comparator<T>`, які витягують `Double`, `Integer` та `Long` ключі з об'єктів перед їх порівнянням на основі природного впорядкування.

```
static <T> Comparator<T>
    comparingDouble(ToDoubleFunction<? super T>keyExtractor);
static <T> Comparator<T>
```

```
        comparingInt(ToIntFunction<? super T>keyExtractor);
static <T> Comparator<T>
        comparingLong(ToLongFunction<? super T>keyExtractor);
```

Розглянемо використання цих методів:

```
public static void main(String[] args) {
    Student s1 = new Student("Larry", 1000, 3.82);
    Student s2 = new Student("Libby", 1001, 3.76);
    ToDoubleFunction<Student> gpaKey2 = x -> x.gpa;
                                              //keyExtractor
    System.out.println(Comparator.comparingDouble(gpaKey2)
                      .compare(s1, s2));                  //1
  /*Порівняння цілих базуючись на натуральному порядку*/
//        System.out.println(Comparator.comparingInt(x ->
//                           x.id).compare(s1, s2)); // ERROR
/*Потрібно вказати компілятору тип параметру лямбда-виразу*/
    System.out.println(Comparator.comparingInt((Student x) ->
                           x.id).compare(s1, s2));     //-1

    LongWrapper l1 = new LongWrapper(4L);
    LongWrapper l2 = new LongWrapper(4L);
    ToLongFunction<LongWrapper> lKey = x -> x.l;
    System.out.println(Comparator.comparingLong(lKey)
                                 .compare(l1, l2));       //0
}
```

В інтерфейс `Comparator<T>` додані методи, які дозволяють проектувати ланцюжки викликів методів компараторів. `Default` методи `thenComparing` повертають компаратор, який використовується для подальшого порівняння, якщо поточний компаратор визначає, що об'єкти, які порівнюються, рівні.

```
default Comparator<T> thenComparing(Comparator<? super T>
                                                    other);

default <U extends Comparable<? super U>> Comparator<T>
thenComparing(Function<? super T, ? extends U> keyExtractor);

default <U> Comparator<T> thenComparing(Function<? super T,
                                 ? extends U> keyExtractor,
                        Comparator<? super U> keyComparator);
```

Розглянемо використання цих методів:

```
public static void main(String[] args) {
    Student s1 = new Student("Joseph", 1000, 3.82);
    Student s2 = new Student("Joseph", 1002, 3.82);
    Comparator<Student> byName = Comparator.comparing(x ->
                                                    x.name);
    Comparator<Student> byId = Comparator.comparing(x->x.id);
    Comparator<Student> byGpa=Comparator.comparing(x->x.gpa);
    System.out.println(byName.compare(s1, s2));           //0
    System.out.println(byName.thenComparing(byGpa)
```

```
                    .compare(s1, s2));   //byName->byGpa  0
    System.out.println(byName.thenComparing(byId)
                .thenComparing(byGpa)
                .compare(s1, s2));  //byName->byId->byGpa -1
                                            //while id compare
     System.out.println(byName.thenComparing(byGpa)
       .thenComparing(byId)
       .compare(s1, s2));  //byName->byGpa->byId          -1
                                            //while id compare
    Comparator<Student> byNameConsonants =
                Comparator.comparing(x -> x.name,
                (x, y) ->
                 removeVowels(x).compareTo(removeVowels(y)));
    Comparator<Integer> byDifference = (x, y) -> x - y;
    Comparator<Double> byCeil = (x, y) ->
                        (int) (Math.ceil(x) - Math.ceil(y));
    Student s3 = new Student("Jean", 1003, 3.86);
    Student s4 = new Student("Jen", 1005, 3.69);
    System.out.println(byNameConsonants
            .thenComparing(x -> x.gpa, byCeil)
            .thenComparing(x -> x.id, byDifference)
            .compare(s3, s4));       //byName->byGpa->byId -2
}
```

Java API забезпечує спеціалізації методу `thenComparing` інтерфейсу `Comparator`, які приймають об'єкти спеціалізованих функцій. Ці спеціалізації витягують `Integer`, `Long` та `Double` ключі з об'єктів перед порівнянням на основі природного впорядкування.

```
default Comparator<T> thenComparingInt(ToIntFunction<? super
T> keyExtractor);
default Comparator<T> thenComparingLong(ToLongFunction<?
super T> keyExtractor);
default Comparator<T> thenComparingDouble(ToDoubleFunction<?
super T> keyExtractor);
```

Наведемо приклад їх використання:

```
public static void main(String[] args) {
    Student s5 = new Student("Kaitlyn", 1006, 3.69);
    Student s6 = new Student("Jane", 1007, 3.69);
    Comparator<Student> byName = Comparator.comparing(x ->
                                                x.name);
    Comparator<Student> byGpa = Comparator.comparing(x ->
                                                x.gpa);
    Comparator<Student> byId = Comparator.comparing(x ->
                                                x.id);
    ToIntFunction<Student> byIdfunc = x -> x.id;
    System.out.println(byGpa.thenComparingInt(byIdfunc)
                            .compare(s5, s6));          //-1
```

```
    Student s7 = new Student("Robert", 1008, 3.86);
    Student s8 = new Student("Robert", 1009, 3.69);
//    ToDoubleFunction<Student> byGpafunc = x -> x.gpa;
    System.out.println(byName.thenComparingDouble(x -> x.gpa)
                             .compare(s7, s8));          //1
}
```

Найчастіше компаратори використовуватимуть для сортування списків та потоків. Інтерфейс `List<E>` має метод з іменем `sort`, який приймає аргумент `Comparator<E>`, який використовується для порівнянь під час сортування.

```
public static void main(String[] args) {
    List<Student> students = Arrays.asList(
            new Student("Joseph", 1623, 3.54),
            new Student("Annie", 1923, 2.94),
            new Student("Sharmila", 1874, 1.86),
            new Student("Harvey", 1348, 1.78),
            new Student("Philipp", 1004, 3.90),
            new Student("Annie", 1245, 2.87)
    );
    Comparator<Student> byGpaCeil = Comparator.comparing(x ->
                                    x.gpa, (x, y) ->
                        (int) (Math.ceil(x) - Math.ceil(y)));
    students.sort(byGpaCeil);
    students.forEach(System.out::println);        //byGpaCeil
    students.sort(byGpaCeil.thenComparing(x -> x.name));
    students.forEach(System.out::println); //byGpaCeil->
                                           //byName
    students.sort(byGpaCeil
            .thenComparing(x -> x.id)
            .thenComparing(x -> x.name));
    students.forEach(System.out::println);  //byGpaCeil->byId
                                            //->byName
}
```

Після запуску програма виведе:

```
Sharmila 1874 1.86      Harvey 1348 1.78        Harvey 1348 1.78
Harvey 1348 1.78        Sharmila 1874 1.86      Sharmila 1874 1.86
Annie 1923 2.94         Annie 1923 2.94         Annie 1245 2.87
Annie 1245 2.87         Annie 1245 2.87         Annie 1923 2.94
Joseph 1623 3.54        Joseph 1623 3.54        Philipp 1004 3.9
Philipp 1004 3.9        Philipp 1004 3.9        Joseph 1623 3.54
```

Об'єкти `Comparator` можуть працювати з масивами Java способом, подібним до списків, за допомогою методу `sort` класу `java.util.Arrays`.

```
public static void main(String[] args) {
    Student[] students = {
        new Student("Joseph", 1623, 3.54),
        new Student("Annie", 1923, 2.94),
        new Student("Sharmila", 1874, 1.86),
```

```
        new Student("Harvey", 1348, 1.78),
        new Student("Philipp", 1004, 3.90),
        new Student("Annie", 1245, 2.87)
    };
    Student[] studentsCopy = Arrays.copyOf(students,
                                           students.length);
    Comparator<Student> byGpaCeil = Comparator.comparing(x ->
       x.gpa, (x, y) -> (int) (Math.ceil(x) - Math.ceil(y)));
    Arrays.sort(students, byGpaCeil.thenComparing(x -> x.id)
                          .thenComparing(x -> x.name));
    for (Student student : students) {
        System.out.println(student);//byGpaCeil->byId->byName
    }
    Arrays.sort(studentsCopy, 2, 5, Comparator.comparing(x ->
                                                  x.name));
    final int NUM_STUDENTS = 1000;
    NumberFormat fmt = NumberFormat.getNumberInstance();
    fmt.setMaximumIntegerDigits(3);
    Student[] studentBody = new Student[NUM_STUDENTS];
    for (int i = 0; i < NUM_STUDENTS; i++) {
        studentBody[i] = new Student("S" + fmt.format(i), i,
                                                      0.0);
    }
    int index = Arrays.binarySearch(studentBody,
                                    new Student("S647", 0, 0.0),
    Comparator.comparing(x -> x.name));
    System.out.println("index = " + index + " "
            + studentBody[index]); //index = 647 S647 647 0.0
}
```

Кілька реалізацій `Map` можна створити з використанням компаратора для упорядкування його записів в певному порядку. Наприклад, записи в `TreeMap` зазвичай організовуються за природним упорядкуванням їх ключів. У наступному прикладі компаратор `byConsonants` використовує лише приголосні у ключах під час упорядкування записів.

```
public static void main(String[] args) {
    Comparator<String> byConsonants = (x, y) ->
                  removeVowels(x).compareTo(removeVowels(y));
    TreeMap<String, String> pets =
                                    new TreeMap<>(byConsonants);
    pets.put("gerbil", "small cute rodents");
    pets.put("guinea pig", "rodents, not pigs");
    pets.put("cat", "have nine lives");
    pets.put("chicken", "more populous than people");
    pets.forEach((x, y) -> System.out.println(x + ", " + y));
    Comparator<Map.Entry<String, String>> cmap =
                                     Map.Entry.comparingByKey();
```

```
    Map.Entry<String, String> cat = pets.ceilingEntry("cat");
    Map.Entry<String, String> chicken =
                                pets.ceilingEntry("chicken");
    System.out.println(cmap.compare(cat, chicken));      //-7
    Comparator<Map.Entry<String, String>> cmapCons =
                      Map.Entry.comparingByKey(byConsonants);
    System.out.println(cmapCons.compare(cat, chicken));  //12
    Comparator<Map.Entry<String, String>> cval
                              = Map.Entry.comparingByValue();
    System.out.println(cval.compare(cat, chicken));      //-5
}
```

Методи `maxBy` та `minBy` функціонального інтерфейсу `BinaryOperator<T>` порівнюють два об'єкти типу `T` на основі `Comparator<T>`:

```
public static void main(String[] args) {
    Comparator<Integer> abscompare = Comparator.comparing(x->
                                                  Math.abs(x));
    BinaryOperator<Integer> bigint = BinaryOperator
                                          .maxBy(abscompare);
    BinaryOperator<Integer> smallint = BinaryOperator
                                          .minBy(abscompare);
    System.out.println(bigint.apply(2, -5));             //-5
    System.out.println(smallint.apply(2, -5));            //2
}
```

`public final class Optional<T>` - це об’єкт-контейнер, який може містити або ненульове значення або `null`. Якщо значення присутнє, метод `Optional<T> isPresent()` повертає `true`. Якщо значення відсутнє, об’єкт `Optional<T>` вважається порожнім, а `isPresent()` повертає `false`. `Optional<T>` був доданий у Java 8.

```
public final class Optional<T> {
    private static final Optional<?> EMPTY =
                                            new Optional<>();
    private final T value;
    …
}
```

Огортання робочого об’єкта типу `T` об’єктом `Optional<T>` доволяє уникнути перевірки робочого об’єкта на `null`. Об’єкти `Optional<T>` також можна використовувати всередині ланцюжків викликів методів для спрощення логіки програми.

Клас `Optional<T>` надає такі статичні методи, що використовуються для створення таких об’єктів:

```
static <T> Optional<T> of(T value);
static <T> Optional<T> ofNullable(T value);
static <T> Optional<T> empty();
```

Метод of створює об’єкт Optional<T> з ненульового аргументу. Якщо його викликати з аргументом null, він генерує NullPointerException. Якщо об'єкт, обгорнутий об’єктом Optional<T>, може бути null, слід використовувати метод ofNullable. Метод empty також може бути використаний для створення об’єкта Optional<T>, який обертає об’єкт null.

```
public static void main(String[] args) {
    Optional<String> o1 = null;
    try {
        o1 = Optional.of(null);
    } catch (NullPointerException e) {
        System.out.println(o1 + " - NullPointerException");
                                  //null –NullPointerException
    }
    Optional<String> o2 = Optional.of("Hello");
    System.out.println(o2 + " - OK");  //Optional[Hello] - OK
    Optional<String> o3 = Optional.ofNullable(null);
    System.out.println(o3 + " - OK");   //Optional.empty - OK
    Optional<String> o4 = Optional.ofNullable("Hello");
    System.out.println(o4+ " - OK");   //Optional[Hello] - OK
    Optional<String> o5 = Optional.empty();
    System.out.println(o5);                    //Optional.empty
    Optional<Object> o6 = Optional.ofNullable(o5);
    System.out.println(o6);          //Optional[Optional.empty]
}
```

Метод isPresent повертає true, якщо Optional<T> містить ненульовий об’єкт, а в протилежному випадку повертає false.

```
public boolean isPresent() {
    return value != null;
}
```

У Java 11 доданий метод isEmpty. Цей метод повертає значення true, якщо Optional<T> обертає нульовий об’єкт, та false - інакше. Цей метод дає протилежний результат порівняно з isPresent.

```
public boolean isEmpty() {
    return value == null;
}
```

Розглянемо використання цих методів:

```
public static void main(String[] args) {
    Optional<String> o5 = Optional.empty();
    if (o5.isPresent()) {
        System.out.println("o5 is non-null");
    }
    Optional<String> imNull = Optional.ofNullable(null);
    if (imNull.isEmpty()) {
        System.out.println("Empty");              //Empty
```

```
    }
}
```

Метод `get` повертає об'єкт, обгорнутий об'єктом `Optional<T>`:

```
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
```

Розглянемо використаня цього методу:

```
public static void main(String[] args) {
    Optional<String> o4 = Optional.ofNullable("Hello");
    Optional<String> o5 = Optional.empty();
    try {
        o5.get();
    } catch (NoSuchElementException e) {
        System.out.println("NoSuchElementException");
                                    //NoSuchElementException
    }
    /*Завжди використовуйте ifPresent перед тим, як викликати
      get*/
    if (o4.isPresent()) {
        System.out.println(o4.get());           //Hello
    }
    if (o5.isPresent()) {
        System.out.println(o5.get());
    } else {
        System.out.println("o5 is null");       //o5 is null
    }
    if (!o5.isEmpty()) {
        System.out.println(o5.get());
    } else {
        System.out.println("o5 is null");       //o5 is null
    }
}
```

Клас `Optional<T>` забезпечує наступні методи, які можна використовувати для створення ланцюжків виклику методів `Optional<T>`:

```
T orElse(T other);
T orElseGet(Supplier<? extends T> supplier);
Optional<T> or(Supplier<? extends Optional<? extends T> >
                                                  supplier);
<X extends Throwable> T orElseThrow(Supplier<? extends X>
                                          exceptionSupplier);
T orElseThrow(); [доданий в JAVA10]
```

Припустимо, програма визначає два рядки `t` і `u`, і програмі потрібно вибрати перший рядок, що не є `null`. Тобто потрібно виконати таку логіку:

```
String t = null;
String u = "Hello";

String s = t;
if (t == null)
    s = u;
```

Метод `T orElse(T other)` повертає значення, яке містить поточний об'єкт `Optional<T>`, а у разі, якщо він містить `null`, метод повертає значення-аргумент:

```
String s = Optional.ofNullable(t)      // Optional has null
                   .orElse(u);         // Optional has "Hello"
```

Метод `T orElseGet(Supplier<? extends T> supplier)` повертає значення, яке містить поточний об'єкт `Optional<T>`, а у разі, якщо він містить `null`, метод повертає значення, яке надає аргумент-постачальник:

```
String s2 = Optional.ofNullable(t)                    // null
                    .orElseGet(() -> u);              // "Hello"
```

Метод `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)` повертає значення, яке містить поточний об'єкт `Optional<T>`, а у разі, якщо він містить `null`, метод генерує виключення, яке генерує постачальник-аргумент. У наступному прикладі об'єкт `Optional<T>` містить `null`, тому метод `orElseThrow` видає виключення, надане його аргументом-постачальником. Виключення ловиться і відображається "`Null Optional`".

```
try {
    String s = null;
    String opt = Optional.ofNullable(s)
                         .orElseThrow(() ->
                             new Exception("Null Optional"));
    }
    catch (Exception e) {
         System.out.println(e.getMessage());
    }
```

У Java 10 був доданий перевантажений метод `T orElseThrow()`. Він повертає значення, яке містить поточний об'єкт `Optional<T>`, а у разі, якщо він містить `null`, метод генерує виключення `NoSuchElementException`. У наступному прикладі таке виключення перехоплюється і на екран виводиться повідомлення `No value present`.

```
try {
    String s = null;
    String opt = Optional.ofNullable(s)
                         .orElseThrow();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
```

```
    }
```

Методи `orElse`, `orElseGet` та `orElseThrow` повертають значення типу `T`. Метод `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)` повертає поточний об'єкт `Optional<T>` у разі, якщо його значення не дорівнює `null`, у іншому випадку метод повертає об'єкт `Optional<? extends T>`, який надає аргумент-постачальник. Метод `or` у наступному прикладі приймає постачальника `<Optional<String>>`, який пропонує користувачеві ввести рядок, якщо значення, що міститься в `Optional`, дорівнює `null`.

```
Supplier<Optional<String>> supplier = () -> {
    System.out.print("Enter a string:");
    return Optional.of((new Scanner(System.in)).nextLine());
};
String s = null;
Optional<String> os = Optional.ofNullable(s)
                              .or(supplier);
if (os.isPresent())
    System.out.println(os.get());
```

При запуску прграма видасть (введення користувачем показано жирним шрифтом):

```
Enter a string: RED
RED
```

Метод `void ifPresent(Consumer<? super T> action)` у разі, якщо "огорнене" значення поточного об'єкту `Optional<T>` не дорівнює `null`, приймає споживача і виконує його функціональний метод над своїм значенням, у іншому разі - метод нічого не робить. Часто цей метод використовується для друку об'єкта, який огорнутий `Optional<T>`. Оскільки `ifPresent` повертає `void`, цей метод повинен бути останнім, якщо він використовується в ланцюжку викликів методів `Optional`.

```
public static void main(String[] args) {
    Supplier<Optional<String>> supplier = () -> {
        System.out.print("Enter a string:");
        return Optional
                    .of((new Scanner(System.in)).nextLine());
    };
    String s = null;
    Optional.ofNullable(s).or(supplier).ifPresent(x ->
                                       System.out.println(x));
    }
```

Метод `Optional<T> filter(Predicate<? super T> predicate)` повертає поточний об'єкт `Optional<T>` у разі, якщо "огорнене" значення поточного об'єкту `Optional<T>` не дорівнює `null` і якщо функціональний метод аргумента-предиката повертає `true`, у іншому разі - метод повертає `Optional.Empty`.

```
public static void main(String[] args) {
    String t = null;
    Optional<String> op = Optional.ofNullable(t).filter(x ->
                                            x.length() > 2);
    System.out.println(op);                 //Optional.empty
    Optional.of("Hello")
            .filter(x -> x.startsWith("H"))
            .filter(x -> x.length() > 2)
            .filter(x -> x.charAt(1) == 'e')
            .ifPresent(x -> System.out.println(x));   //Hello
    /*Функціональний метод першого предикату повертає false,
      тому другий і третій предикати не виконуються, і нічого
      не виводиться*/
    Optional.of("Hello")
            .filter(x -> x.startsWith("i"))
            .filter(x -> x.length() > 2)
            .filter(x -> x.charAt(1) == 'e')
            .ifPresent(x -> System.out.println(x));
                                        //нічого не виводиться
    /* Умови фільтрування можуть бути складеними за допомогою
      методів логіки інтерфейсу Predicate*/
    Predicate<String> p = x -> x.charAt(0) == 'i';
    Optional.of("Hello")
            .filter(p.or(x -> x.startsWith("H")))
            .ifPresent(x -> System.out.println(x));   //Hello
}
```

Метод `<U> Optional<U> map(Function<? super T,? extends U> mapper)` повертає об'єкт `Optional<U>`, що є результатом застосування функції-аргумента до "огорненого" поточним об'єктом `Optional<T>` значення, у разі якщо воно не дорівнює `null`, у іншому разі - метод повертає `Optional.Empty`. Наступний приклад перетворює `Optional<String>` в `Optional<Integer>`, а потім застосовує цілочисельні арифметичні операції в предикатах, що слідують далі. Оскільки функціональні методи предикатів повертають `true`, в кінці відображається значення `4`.

```
Optional.of("4")
        .map(x -> Integer.parseInt(x))          // Optional(4)
        .filter(x -> x > 2)                     // Optional(4)
        .filter(x -> (x%2) == 0)                // Optional(4)
        .ifPresent(x -> System.out.println(x)); // 4
```

Оскільки метод `ifPresent` повертає `void`, будь-які зміни в базовому об'єкті не зберігатимуться.

```
Optional<Integer> o1 = Optional.of(2);
o1.ifPresent(x -> ++x);
o1.ifPresent(x -> System.out.println(x));      // 2
```

Метод map може бути використаний для модифікації "огорненого" об'єкта, використовуйте для цього однаковий вхідний та вихідний типи функції (забезпечивши реалізацію UnaryOperator). Наступний приклад змінює базове значення в Optional<Integer> з 2 на 3. Зверніть увагу на використання префіксної форми інкременту у функції map.

```
Optional.of(2)                                  // Optional(2)
        .map(x -> ++x)                          // Optional(3)
        .ifPresent(x -> System.out.println(x)); // 3
```

Метод <U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper) подібний до методу map, за винятком того, що результат його функції "загорнутий" у Optional<? extends U>.

У наступному прикладі метод flatMap відображає рядок “4” у ціле число 4, використовуючи функцію, яка повертає Optional<Integer>.

```
Optional.of("4")                                // Optional("4")
        .flatMap(x ->
           Optional.of(Integer.parseInt(x)))    // Optional(4)
                   .ifPresent(x -> System.out.println(x));//4
```

Ланцюжок викликів методів Optional може використовуватися для відстеження споживання ресурсу. Клас Resource, визначений нижче, дозволяє користувачеві споживати ресурс, коли викликається його конструктор, а потім зменшує кількість ресурсів що залишилась.

```
class Resource {
    static int count = 3;
    public Resource(){
        count--;
        System.out.println("Resource consumed, " + count
                           + " remaining.");
    }
}
```

Ланцюжок викликів методів Oprional може бути використаний для споживання ресурсу викликом конструктора Resource як в методі of, так і в методах map. Для перевірки ресурсів, що залишились використовується метод filter. Останній виклик не виконується, оскільки кількість ресурсів дорівнює 0.

```
Optional.of(new Resource())             // Resource.count = 2
        .filter(x -> x.count > 0)
        .map(x -> new Resource())       // Resource.count = 1
        .filter(x -> x.count > 0)
        .map(x -> new Resource())       // Resource.count = 0
        .filter(x -> x.count > 0)       // Optional(null)
        .map(x -> new Resource());      // не виконується
```

Вивід програми буде такий:

```
Resource consumed, 2 remaining.
Resource consumed, 1 remaining.
```

Resource consumed, 0 remaining.

Інтерфейс `Stream<T>` забезпечує операції, які можна виконати з *низкою* (*потоком*) *значень типу* `T` (*даних*). Об'єкт, який реалізує інтерфейс `Stream<T>` (далі, для стислості та відміни від I/O потоків та потоків виконання (threads), будемо називати їх *стрімами*), схожий на об'єкт `Optional<T>` але може містити кілька значень замість одного. Стрім має багато методів, однакових за назвою та функціональністю з методами `Optional<T>`, включаючи `filter`, `map` та `flatmap`. Оскільки багато методів інтерфейса `Stream<T>` повертають об'єкти цього ж інтерфейса, можна створювати дуже функціональні ланцюжки викликів методів стрімів. Існують два типи об'єктів `Stream<T>` - *послідовний* (виконується одним потоком виконання (thread) і *паралельний* (виконується одночасно декількома потоками виконання).

Є багато способів створення стрімів. Умовно їх можна поділити на дві групи - методи самого інтерфейсу `java.util.stream.Stream<T>` та методи об'єктів, з которих можливо згенерувати об'єкт `Stream<T>` (до таких об'єктів відносяться колекції, масиви, рядки, об'єкти `Optional`, файли). Зараз розглянемо першу групу методів створення об'єктів `Stream`:

```
public static void main(String[] args) {
    /*Створення пустого послідовного стріма*/
    Stream<Integer> nums = Stream.empty();
    /*Створення послідовного упорядкованого стріма
      із значень*/
    Stream<Integer> numbers = Stream.of(1, 2, 3, 4);
    /*Створення послідовного упорядкованого стріма відповідно
      до початкового значення та унарного оператора*/
    Stream<Integer> tenIterateNums = Stream.iterate(1, x ->
                                          2 * x).limit(10);
    /*Генерування стріма відповідно до аргумента-
      постачальника*/
    Stream<Integer> tenRandomNumbers = Stream.generate(() ->
            (new Random()).nextInt(100)).limit(10);
    /*Генерування стріма за допомогою Builder*/
    Stream.builder().add("First").add("Second").build()
          .forEach(System.out::println);       //First Second
}
```

Що стосується другої групи методів створення стрімів, то будь-які колекцію, масив, рядок або об'єкт `Optinal` можна перетворити на стрім, використовуючи метод `Stream<E> stream()` інтерфейса `java.util.Collection<E>`, або `java.util.Optional<T>`, метод `stream(T[] array)` класа `java.util.Arrays`, метод `lines()` класа `java.lang.String` та інші.

Розглянемо приклад способів створення об'єктів `Stream` із інших об'ктів:

```
public static void main(String[] args) {
    /*Створення стріма з колекції*/
    List<String> list = Arrays.asList("RED", "GREEN");
    list.stream()
         .forEach(x -> System.out.print(x + " "));
    System.out.print();                    // RED GREEN
    /*Створення стріма з масиву*/
    String[] arr = {"RED", "GREEN"};
    Arrays.stream(arr)
            .forEach(x -> System.out.println(x + " "));
    System.out.print();                    // RED GREEN
    /*Створення стріма з об'єкта Optional<T>*/
    Optional.of("RED", "GREEN")
            .stream()
            .forEach(x -> System.out.println(x + " "));
    System.out.print();                    // RED GREEN
    /*Створення стріма з рядка*/
    String s = "Java supports functional progamming\n based on
functional interfaces";
        s.lines()
         .filter(x -> x.contains("Java"))
         .forEach(System.out::println); //Java supports
                                        //functional progamming
    /*Створення стріма об'єктів File*/
    Files.walk(Paths.get("E:\\1"))
            .filter(Files::isRegularFile)
            .filter(p->p.toString().endsWith("jpg"))
            .forEach(System.out::println);

    /*Створення стріма рядків вмісту файлу*/
    String fileCOntent =Files.lines(Paths.get("c:\\fil.txt"))
                .collect(Collectors.joining(","));
    System.out.println(fileCOntent);
}
```

Методи інтерфейсу `java.util.stream.Stream<T>` умовно можна поділити на *конвейерні* та *термінальні*. Метод `void forEach(Consumer<? super T> action)` використовується для обходу стрімів аналогічно колекціям. Цей метод відноситься до термінальних методів (оскільки повертає `void`) і приймає об'єкт споживача, функціональний метод `accept(T t)` якого послідовно приймає та обробляє елементи стріма.

Метод `Stream<T> filter(Predicate<? super T> predicate)` інтерфейсу `java.util.stream.Stream<T>` видаляє з потоку елементи, які не відповідають його предикату, метод відноситься до конвейерних методів (оскільки повертає об'єкт `Stream<T>`, з якого можуть бути визваними інші методи).

Приклад використання методів:

```
public static void main(String[] args) {
    Stream.of("RED", "GREEN", "BLUE", "RED")
            .filter(x -> x.equals("YELLOW"))
            .forEach(x -> System.out.println(x));
                                        //нічого не виводиться
    Stream.of("RED", "GREEN", "BLUE", "RED") //повторне
            .filter(x -> x.equals("RED"))  //створення стріму
            .forEach(x -> System.out.print(x + " "));
                                          //RED RED
    System.out.println();
    Predicate<String> isRed = x -> x.equals("RED");
    Stream.of("RED", "GREEN", "BLUE", "RED")
            .filter(isRed.or(x -> x.indexOf("R") > -1))
            .forEach(x -> System.out.print(x + " "));
                                        //RED GREEN RED
}
```

Зверіть увагу на особливості роботи стрімів:

- об’єкт стріму не можна використовувати більше одного разу;
- обробка не почнеться доти, поки не буде викликаний термінальний оператор.

Для сортування елементів у стрімі можна використовувати такі методи інтерфейсу `java.util.stream.Stream<T>`:

```
Stream<T> sorted();         //сортування за природним порядком
Stream<T> sorted(Comparator<? super T> comparator);
```

Розглянемо приклад сортування:

```
public static void main(String[] args) {
    Stream.of("Kyle", "Jaquiline", "Jimmy")
            .sorted()
            .forEach(x -> System.out.print(x + " "));
    System.out.println();               //Jaquiline Jimmy Kyle

    Stream.of("Kyle", "Jaquiline", "Jimmy")
            /*Comparator using*/
            .sorted((x, y)
               -> removeVowels(x).compareTo(removeVowels(y)))
            .forEach(x -> System.out.print(x + " "));
    System.out.println();               //Jimmy Jaquiline Kyle
}

private static String removeVowels(String s) {
    return s.replaceAll("[aeiou]", "");
}
```

Також інтерфейс `java.util.stream.Stream` містить методи `Optional<T> min(Comparator<? super T> comparator)` та `Optional<T>`

max(Comparator<? super T> comparator), які вибирають найменший та найбільший елемент у потоці згідно з компаратором, наведеним у його аргументі, відповідно. Обидва методи повертають об'єкт Optional<T> того самого типу, що і елементи потоку. Наведемо приклад використання цих методів:

```
public class SmallestLargestElementSelection {

    private static String removeVowels(String s) {
        return s.replaceAll("[aeiou]", "");
    }

    public static void main(String[] args) {
        Stream.of("Kyle", "Jaquiline", "Jimmy")
            .min((x, y) -> removeVowels(x)
            .compareTo(removeVowels(y)))    //Optional<String>
            .ifPresent(x -> System.out.println(x));    //Jimmy

        Stream.of("Kyle", "Jaquiline", "Jimmy")
            .max((x, y) -> removeVowels(x)
            .compareTo(removeVowels(y)))    //Optional<String>
            .ifPresent(x -> System.out.println(x));     //Kyle
    }
}
```

Метод <R> Stream<R> map(Function<? super T,? extends R> mapper) інтерфейсу Stream<T> застосовує функцію mapper до кожного елементу і потім повертає стрім, в якому елементами будуть результати функції.

Метод <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) інтерфейсу Stream<T> відрізняється від методу map тим, що його функція-аргумент повертає стрім вихідних типів. Це дозволяє об'єднувати значення кількох вхідних стрімів в один вихідний (канкатенація стрімів).

Порівняємо методи map та flatMap, використовучи наступний приклад. Нехай у нас є два класи - студент та група студентів:

```
public class Student {
    String name;
    Integer id;
    Double gpa;

    public Student(String n, int i, double g){
        name = n;
        id = i;
        gpa = g;          }

    @Override
    public String toString() {
        return name + " " + id + " " + gpa;
    }
}
```

```
public class ClassForSubject {
    String subject;
    Collection<Student> students;

    public ClassForSubject(String su, Student... st) {
    subject = su;
        students = Arrays.asList(st);
    }
}
//У деякому класі
public static void main(String[] args) {

    Stream.of(new ClassForSubject("Biology",//Створюємо стрім
            new Student("Joe", 1001, 3.81), //знову
            new Student("Mary", 1002, 3.91)),
            new ClassForSubject("Physics",
            new Student("Kalpana", 1003, 3.61),
            new Student("Javier", 1004, 3.71)))
                                    //Stream<ClassForSubject>
        .map(x -> x.students)          //Collection<Student>>
        .forEach(x -> System.out.println(x)); //Print
                                //distributed by class students

    Stream.of(new ClassForSubject("Biology",
            new Student("Joe", 1001, 3.81),
            new Student("Mary", 1002, 3.91)),
        new ClassForSubject("Physics",
            new Student("Kalpana", 1003, 3.61),
            new Student("Javier", 1004, 3.71)))
                                    //Stream<ClassForSubject>
         .flatMap(x -> x.students.stream()) //Stream<Student>
         .forEach(x -> System.out.println(x)); //Print all 4
                                   //students of both classes
}
```

Метод `map` фактично повертає колекцію студентів певного класу і метод `forEach` виводить списки студентів окремо:

Output:

```
[Joe 1001 3.81, Mary 1002 3.91]
[Kalpana 1003 3.61, Javier 1004 3.71]
```

Метод `flatMap` дозволяє конкатувати потоки студентів різних класів, після нього метод `forEach` виводить студентів обох класів з одного потоку:

Output:

```
Joe 1001 3.81
Mary 1002 3.91
Kalpana 1003 3.61
Javier 1004 3.71
```

Обидва методи `map` та `flatMap` є конвейерними.

Метод `Optional<T> reduce(BinaryOperator<T> accumulator)` перетворює всі елементи стріму в один об'єкт відповідно до аргумента-бінарного оператора (наприклад, рахує суму всіх елементів, або знаходить мінімальний (максимальний) елемент).

Метод `T apply(T x, T y)` бінарного оператору приймає послідовні елементи стріму, виконує над ними операцію, та повертає єдиний результат. Після чого операція виконується над результатом і наступним елементом стріму і так далі до останнього елементу. Наприклад, добуток елементів стріму може бути отриманий так:

```
Stream.of(1, 2, 3, 4, 5)                    //Stream(Integer)
        .reduce((x, y) -> x * y)            //Optional(Integer)
        .ifPresent(x -> System.out.println(x)); //120
```

Метод `reduce` має перевантажену версію, яка приймає додатковий параметр `T identity`, який є початковим значенням для запущеного обчислення - воно поступає до методу `T apply(T x, T y)` бінарного оператору разом з першим елементом стріму на першому кроці обчислень.

```
public abstract T reduce(T identity, BinaryOperator<T>
                                                accumulator);
```

Попередній приклад з `identity=2` дасть у 2 рази більший результат:

```
Stream.of(1, 2, 3, 4, 5)                    //Stream(Integer)
        .reduce(2, (x, y) -> x * y)         //Optional(Integer)
        .ifPresent(x -> System.out.println(x)); //240
```

І є ще одна версія методу `reduce`, яка може перетворити складні об’єкти на більш прості редукції:

```
public abstract <U> U reduce(U identity,
                      BiFunction<U,? super T,U> accumulator,
                      BinaryOperator<U> combiner);
```

Для прикладу використання цієї версії методу розглянемо клас:

```
public class TwoInts {

    Integer i1;
    Integer i2;

    public TwoInts(int i1, int i2) {
        this.i1 = i1;
        this.i2 = i2;
    }
}
```

Нехай є завдання підрахувати суму полів `i2` декількох об’єктів `TwoInts`.

```
Stream<TwoInts> two = Stream.of(new TwoInts(1,2),
                                          new TwoInts(8,9));
BiFunction<Integer,TwoInts,Integer> accumulator = (x,y) ->
                                                  x + y.i2;
BinaryOperator<Integer> combiner = (x,y) -> x += y;
```

```
Integer j = two.reduce(0,accumulator,combiner);
System.out.println(j);                                    //11
```

тут бінарна функція-акумулятор виконує обчислення для двох параметрів різного типу, а бінарний оператор сумує обраховані значення.

Додаткові приклади використання методу `reduce` наведені нижче:

```
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 5, 7);
    /*Пошук мінімального значення*/
    Integer min = numbers.stream()
        .reduce(Integer.MAX_VALUE, (left, right) ->
                            left < right ? left : right);
    System.out.println(min);                              //1
    /*Пошук максимального значення
      (Integer::max те ж що і (a, b) -> Integer.max(a, b))*/
    Integer max = numbers.stream()
        .reduce(Integer.MIN_VALUE, Integer::max);
    System.out.println(max);                              //7
    /*Пошук найдовшого рядка*/
    List<String> strings = Arrays.asList("aaa", "bbb", "ccc",
                                         "ddd", "ffff");
    String s = strings.stream()
        .reduce("", (left, right) ->
              left.length() > right.length() ? left : right);
    System.out.println(s);                               //ffff
}
```

Колектор - конструкція загального характеру для породження складених значень зі стрімів. Колектор можна використовувати з довільним стрімом, передавши його в якості аргументу методу `<R,A> R collect(Collector<? super T,A,R> collector)`. За допомогою колекторів можна зібрати всі елементи у список, множину або іншу колекцію, згрупувати елементи по якомусь критерію, об'єднати все в рядок і т.і. У стандартній бібліотеці є ряд готових корисних колекторів. У класі java.util.stream.Collectors дуже багато методів на всі випадки життя, ми розглянемо їх пізніше. При бажанні можна написати свій колектор, реалізувавши інтерфейс java.util.stream.Collector<T, A, R>.

Інтерфейс java.util.stream.Collector<T, A, R> є абстракцією операції редукції, яка накопичує вхідні елементи у контейнер результатів, які можуть змінюватись (*mutable container*), перетворюючи накопичений результат в остаточне представлення після обробки всіх вхідних елементів. Операції редукції можуть виконуватися як послідовно, так і паралельно.

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
```

```
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
    public static<T, R> Collector<T, R, R> of(Supplier<R>
                                                    supplier,
                        BiConsumer<R, T> accumulator,
                        BinaryOperator<R> combiner,
                        Characteristics... characteristics) {…}
    public static<T, A, R> Collector<T, A, R> of(Supplier<A>
                                                    supplier,
                        BiConsumer<A, T> accumulator,
                        BinaryOperator<A> combiner,
                        Function<A, R> finisher,
                        Characteristics... characteristics) {…}
    enum Characteristics {…}
}
```

Інтерфейс Collector має три змінні типу:

- `T` - тип вхідних елементів операції редукції;
- `A` - тип змінної акумуляції операції редукції (часто прихований як деталь реалізації);
- `R` - тип результату операції редукції.

Колектор задається чотирма функціями, які працюють разом, щоб накопичувати записи у mutable контейнері результатів, і виконувати остаточне перетворення результату:

- `Supplier<A> supplier()` – функція, що створює та повертає mutable контейнер результатів (постачальник);
- `BiConsumer<A, T> accumulator()` – функція, що виконує додання нового елемента даних у контейнер результатів (накопичувач);
- `BinaryOperator<A> combiner()` – функція, що виконує об'єднання двох часткових результатів в один (об'єднувач);
- `Function<A, R> finisher()` – функція, що виконує необов’язкове остаточне перетворення проміжного типу даних акумулятора на остаточний тип даних результату операції (завершувач).

Інтерфейс Collector<T, A, R> також містить набір характеристик в енумераторі `Characteristics`, які надають підказки, як можуть використовуватися реалізаціями функцій операції редукції:

- `CONCURRENT` - вказує, що функція акумулятора може викликатись одночасно декількома потоками виконання (thread);
- `UNORDERED` - вказує, що операція редукції не гарантує порядок елементів рівний їх вхіжному порядку;

- `IDENTITY_FINISH` - вказує на те, що функція завершувача `finisher()` є функцією ідентичності та може бути вилучена (функція ідентичності повертає значення свого параметра).

Послідовна реалізація редукції за допомогою колектора створить єдиний контейнер результатів, використовуючи функцію постачальника, і викличе функцію накопичувача один раз для кожного вхідного елемента. Паралельна реалізація буде розділяти вхідні дані, створювати контейнер результатів для кожної частини, накопичувати вміст кожної частини в підрезультаті для цього розділу, а потім використовувати функцію-об'єднувач для об'єднання підрезультатів у комбінований результат.

Роглянемо приклад створення та застосування колектора. Припустимо, програмі потрібно розташувати елементи потоку символів таким чином, щоб буквені символи з'являлися перед числовими символами. Можна створити mutable редукцію потоку символів з розташуванням у зазначеному порядку за допомогою екземпляра реаліації інтерфейсу Collector<T, A, R>.

Статичні методи інтерфейсу Collector<T, A, R> створюють mutable редукцію типу `R`, яка в цьому прикладі буде `List<Character>`. Постачальник генерує контейнери типу `List<Character>`, які використовуються для накопичення, а потім об'єднуються в остаточну колекцію.

```
public static void main(String[] args) {
    Supplier<List<Character>> supp = () ->
                                new ArrayList<Character>();
```

Акумулятор додає елементи типу `T` в контейнер. У цьому прикладі алфавітні символи додаються до початку `ArrayList`, тоді як неалфавітні символи додаються до кінця. Для наочності додані діагностичні оператори друку.

```
    BiConsumer<List<Character>, Character> acc = (x, y) -> {
        System.out.print("acc: x=" + x + " y=" + y
                                            + " result=");
        if (Character.isAlphabetic(y)) {
            x.add(0, y);
        } else {
            x.add(y);
        }
        x.forEach(z -> System.out.print(z));
        System.out.println();
    };
```

Використовуваний акумулятор має тип `BiConsumer<R, T>`. Перевантажена версія використовує `BiConsumer<A, T>`, але редукцію потрібно трансформувати до типу `R`, використовуючи функцію-завершувача.

Об’єднувач поєднує контейнери в остаточну редукцію.

```
    BinaryOperator<List<Character>> comb1 = (x, y) -> {
        x.addAll(y);
```

```
        return x;
    };
```

Об'єднувач, що використовується, має тип `BinaryOperator<R>`. Перевантажена версія використовує `BinaryOperator<A>`, але редукцію потрібно трансформувати до типу `R`, використовуючи функцію-завершувача.

Метод `collect` інтерфейсу Stream<T> використовує Collector для отримання mutable редукції елементів стріма.

```
    Stream.of('1', 'a', 'b', '2')          //Stream<Character>
        .collect(Collector.of(supp, acc, comb1))
                                           //List<Character>
        .forEach(x -> System.out.print(x + " "));
    System.out.println();
}
```

Результатом запуску програми буде:

Output:

```
acc: x=[] y=1 result=1
acc: x=[1] y=a result=a1
acc: x=[a, 1] y=b result=ba1
acc: x=[b, a, 1] y=2 result=ba12
b a 1 2
```

Акумулятор `x` спочатку порожній. Потім елемент стріму `1` розміщується у список, який формується `[1]`. Далі елемент стріму `a` розміщується на початку списку `[a1]`. Потім елемент стріму `b` розміщується на початку, формуючи список `[ba1]`. Нарешті, елемент стріму `2` розміщується в кінець списку `[ba12]`.

Також могла бути використана перевантажена версія методу `collect`, яка приймає як параметри постачальника, накопичувача та об'єднувача без створення об'єкта Collector. У цьому випадку об'єднувач має тип `BiConsumer<R, R>` замість `BinaryOperator<R>`:

```
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T>
                      accumulator, BiConsumer<R,R> combiner);
```

Наведемо приклад такого об'єднувача для задачі, що описана вище:

```
BiConsumer<List<Character>, List<Character>> comb2 = (x, y)
            -> x.addAll(y);
    Stream.of('1', 'a', 'b', '2')
            .collect(supp, acc, comb2)
            .forEach(x -> System.out.print(x + " "));
    System.out.println();
```

Mutuable редукція також може бути сформована за допомогою класу StringBuilder. У наступному прикладі накопичувач видаляє голосні з рядка, перш ніж використовувати метод `append(String)` класу StringBuilder для розміщення елементу в кінці редукції.

```
    Supplier<StringBuilder> supps = ()
            -> new StringBuilder();
```

```
    BiConsumer<StringBuilder, String> accs = (x, y)
            -> x.append(y.replaceAll("[aeiou]", ""));
    BinaryOperator<StringBuilder> combs = (x, y) -> {
        x.append(y);
        return x;
    };
    Function<StringBuilder, String> fins = x -> x.toString();
    String s = Stream.of("Joe", "Kalpana", "Christopher")
            .collect(Collector.of(supps, accs, combs, fins));
    System.out.println(s);
```

Виконання програми виведе:

Output:

```
JKlpnChrstphr
```

Клас java.util.stream.Collectors містить статичні методи, які створюють об'єкти Collector<T, A, R>, що вирішують різні проблеми. У наступному прикладі використовується метод `toList` для накопичення елементів стріму у списку в оригінальному порядку

```
public static <T> Collector<T, ?, List<T>> toList()
```

Приклад використання методу:

```
List<Character> list = Stream.of('1', 'a', 'b', '2')
      .collect(Collectors.toList());
System.out.print(list);   //[1, a, b, 2]
```

Також існує метод `toSet` для накопичення елементів стріму у множині.

```
public static <T> Collector<T, ?, Set<T>> toSet()
```

Приклад:

```
Set<Character> set = Stream.of('1', 'a', '1', 'a')
            .collect(Collectors.toSet());
System.out.println(set);                    //[1, a]
```

і метод `toCollection`, у якому тип колекції задається фабрикою-постачальником:

```
public static <T, C extends Collection<T>> Collector<T, ?, C>
                  toCollection(Supplier<C> collectionFactory)
```

Наприклад,

```
Deque<Integer> deque = Stream.of(1, 2, 3, 4, 5)
          .collect(Collectors.toCollection(ArrayDeque::new));
System.out.println(deque);       //[1, 2, 3, 4, 5]
```

Також існує декілька перевантажених версій метода `toMap`, наприклад:

```
public static <T, K, U> Collector<T, ?, Map<K,U>>
          toMap(Function<? super T, ? extends K> keyMapper,
                Function<? super T, ? extends U> valueMapper)
```

Приклад використання:

```
Map<Integer, String> map1 = Stream.of(1, 2, 3)
```

```
    .collect(Collectors.toMap(
        Function.identity(),
        i -> String.format("%d", i * 2)
    ));
map1.forEach((k, v) -> System.out.print(k + "=" + v + " "));
                                                  // 1=2 2=4 3=6
```

У прикладі метод `Function.identity()` передає у якості результату значення поточного елементу стріму, але у якості функції `keyMapper` може виступати будь-яка функція.

Метод `collectingAndThen` збирає елементи потоку за допомогою об'єкта `Collector`, а потім перетворює результат на інший тип за допомогою функції завершувача:

```
static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(
       Collector<T,A,R> downstream, Function<R,RR> finisher);
```

Наведемо приклад його використання:

```
Supplier<List<Character>> supp = () ->
                                    new ArrayList<Character>();
BiConsumer<List<Character>, Character> accc2 = (x, y) -> {
    if (Character.isAlphabetic(y)) {
        x.add(0, y);
    } else {
        x.add(y);
    }
};
BinaryOperator<List<Character>> comb1 = (x, y) -> {
    x.addAll(y);
    return x;
};
Function<List<Character>, String> fins2 = x -> {
    String t = "";
    for (Character c : x) {
        t += c;
    }
    return t;
};
String t = Stream.of('1', 'a', 'b', '2')  //Stream<Character>
    .collect(Collectors.collectingAndThen(Collector.of(supp,
             accc2, comb1),                  //List<Character>
             fins2));                        //String
System.out.println(t);                       //ba12
```

Метод `reducing` збирає елементи стріму в об’єкт Optional<T>. Він приймає BinaryOperator<T>.

```
static <T> Collector<T,?,Optional<T>>
```

```
                                    reducing(BinaryOperator<T> op)
```

Приклад використання метода `reducing`:

```
Stream.of(1, 2, 3, 4, 5)                    //Stream<Integer>
      .collect(Collectors.reducing((x, y) -> x *= y))
                                            //Optional<Integer>
      .ifPresent(System.out::println);      //120
```

Метод `joining` об'єднує стрім рядків в один об'єкт String.

```
static Collector<CharSequence,?,String> joining()
```

Приклад використання метода `joining`:

```
String s = Stream.of("RED", "GREEN", "BLUE")// Stream<String>
        .collect(Collectors.joining());     // String
System.out.println(s);                      //REDGREENB
```

Можливе об’єднання рядків з вказівкою роздільника:

```
String s1 = Stream.of("RED", "GREEN", "BLUE")
                                            // Stream<String>
        .collect(Collectors.joining(","));  // String
System.out.println(s1);                     //RED,GREEN,BLUE
```

Метод `groupingBy` впорядковує результат за ключовим значенням. Отриманий контейнер - це об’єкт-реалізація Map, значення якого є переліком елементів стріму. Повинна бути надана функція, яка перетворює елемент стріму у значення ключа.

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
                  Function<? super T,? extends K> classifier)
```

Наведемо приклад використання методу `groupingBy`. Нехай є клас `Car`:

```
class Car {
    String manu;
    String model;
    int mpg;
    public Car(String ma, String mo, int mp) {
        manu = ma;
        model = mo;
        mpg = mp;
    }
    public String toString() {
        return manu + " " + model + " gets " + mpg + " mpg";
    }
}
```

Можн виконати угрупування по фірмі-виробнику автомобілів:

```
Stream.of(new Car("Buick", "Regal", 25),
          new Car("Hyundai", "Elantra", 27),
          new Car("Buick", "Skylark", 26),
          new Car("Hyundai", "Accent", 30))     //Stream<Car>
      .collect(Collectors.groupingBy(x -> x.manu))
```

```
                                        //Map<String,List<Car>>
      .forEach((x, y) -> System.out.println(x + ": " + y));
/*Buick: [Buick Regal gets 25 mpg, Buick Skylark gets 26 mpg]
  Hyundai: [Hyundai Elantra gets 27 mpg, Hyundai Accent gets
30 mpg]*/
```

Перевантажена версія методу `groupingBy` приймає об'єкт Collector як другий аргумент. Як другий аргумент часто використовується метод `mapping`, що перетворює елемент потоку у нове значення, а потім генерує новий список з новими значеннями елементів:

```
static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,
  ? extends U> mapper, Collector(<? super U,A,R> downstream);
```

У такому разі:

```
Stream.of(new Car("Buick", "Regal", 25),
          new Car("Hyundai", "Elantra", 27),
          new Car("Buick", "Skylark", 26),
          new Car("Hyundai", "Accent", 30))     //Stream<Car>
    .collect(Collectors.groupingBy(x -> x.manu,
                                   //Map<String,List<Integer>>
Collectors.mapping(x -> x.mpg, Collectors.toList())))
                                                //List<Integer>>
    .forEach((x, y) -> System.out.println(x + ": " + y));
/*Buick: [25, 26]
  Hyundai: [27, 30]*/
```

Метод `partitioningBy` групує елементи за критерієм, зберігаючи результат в Map, розділяє редукцію на два компоненти на основі критерію проходження/відмови. Ключ `Map` має тип `Boolean`, і передбачається використання предикату, який реалізує критерій проходження/відмови.

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
                                  Predicate<? super T> predicate)
```

Розглянемо приклад використання методу `partitioningBy`:

```
Stream.of(new Car("Buick", "Regal", 25),
          new Car("Hyundai", "Elantra", 27),
          new Car("Buick", "Skylark", 26),
          new Car("Hyundai", "Accent", 30))     //Stream<Car>
        .collect(Collectors.partitioningBy(x -> x.mpg >= 30))
                                   //Map<Boolean,List<Car>>
        .forEach((x, y) -> System.out.println(x + ": " + y));
    /*false: [Buick Regal gets 25 mpg, Hyundai Elantra gets
      27 mpg, Buick Skylark gets 26 mpg]
      true: [Hyundai Accent gets 30 mpg]*/
```

Клас Collectors має методи, які створюють суму як mutable редукцію. Ці методи приймають функцію, яка перетворює елемент потоку в `int`, `long` або `double`, які підсумовуються.

```
static <T> Collector<T, ? , Integer> summingInt(ToIntFunction<?
```

```
                                                    super T> mapper)
```

Приклад використання методу `summingInt`:

```
Integer sum = Stream.of(new Car("Buick", "Regal", 25),
                        new Car("Hyundai", "Elantra", 27),
                        new Car("Buick", "Skylark", 26),
                        new Car("Hyundai", "Accent", 30))
                    .collect(Collectors.summingInt(x -> x.mpg));
    System.out.println("sum of mpg = " + sum);
                                                //sum of mpg = 108
```

Також є метод розрахунку середнього:

```
public static <T> Collector<T, ?, Double>
                  averagingInt(ToIntFunction<? super T> mapper)
```

та виведення статистики:

```
public static <T> Collector<T, ?, IntSummaryStatistics>
                summarizingInt(ToIntFunction<? super T> mapper)
```

Приклад:

```
Object stat
            = Stream.of(new Car("Buick", "Regal", 25),
                        new Car("Hyundai", "Elantra", 27),
                        new Car("Buick", "Skylark", 26),
                        new Car("Hyundai", "Accent", 30))
                .collect(Collectors.summarizingInt(x -> x.mpg));
        System.out.println("Statistics: " + stat);
                //Statistics: IntSummaryStatistics{count=4,
                //sum=108, min=25, average=27,000000, max=30}
```

Також є метод, що підраховує кількість елементів стріму:

```
public static <T> Collector<T, ?, Long> counting()
```

Наприклад,

```
Long count = Stream.of(new Car("Buick", "Regal", 25),
                new Car("Hyundai", "Elantra", 27),
                new Car("Buick", "Skylark", 26),
                new Car("Hyundai", "Accent", 30))
                .collect(Collectors.counting());
        System.out.println(count);                          //4
```

Інтерфейс `Stream.Builder` - це підінтерфейс `Consumer`. За його допомогою можна інтерактивно будувати потік. Для створення `Stream.Builder` використовується статичний метод `builder` інтерфейсу `Stream`.

```
public static void main(String[] args) {
    Stream.Builder<String> bld = Stream.builder();
        bld.accept("RED");
        bld.accept("GREEN");
        bld.accept("BLUE");
        Stream<String> st = bld.build();    //build immutable
                                            //stream
```

```
        st.forEach(x -> System.out.print(x + " "));
                                              //RED GREEN BLUE
        System.out.println();
        try {
            bld.accept("YELLOW");      //can not add elements
                                       //to builted stream
        } catch (IllegalStateException e) {
            System.out.println("IllegalStateException");
                                       //llegalStateException
        }
    /* Оскільки метод add повертає об'єкт Stream.Builder,
      його можна використовувати в ланцюжку методів*/
        Stream.builder()
                .add("RED")
                .add("GREEN")
                .add("BLUE")
                .build()
                .forEach(x -> System.out.print(x + " "));
                                         //RED GREEN BLUE
}
```

Метод `Stream<T> peek(Consumer<? super T> action)` інтерфейсу `Stream` виконує дію над кожним елементом стріму і при цьому повертає стрім з елементами вихідного стріму. Служить для того, щоб передати елемент куди-небудь, не пориваючи при цьому ланцюжок операторів, або для відлагодження.

```
Stream.of(1, 2, 3, 4)
      .peek(x -> System.out.print(x + " "))          //1 2 3 4
      .reduce((x, y) -> x += y)
      .ifPresent(x -> System.out.println(x));             //10
```

В Java 9 до інтерфейсу Stream<T> були додані методи

```
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

Перший повертає елементи стріму до тих пір, поки вони задовольняють умові, тобто функція-предикат повертає true. Метод працює аналогічно методу інтрфейса Stream `Stream<T> limit(long maxSize)`, тільки не з числом, а з умовою. Другий пропускає елементи до тих пір, поки вони задовольняють умові, а потім повертає решту стріму. Якщо предикат повернув для першого елемента `false`, то жодного елемента не буде пропущено. Метод подібний методу інтерфейсу Stream `Stream<T> skip(long n)`, тільки працює за умовою. Наведемо приклади використання методів:

```
Stream.of(1, 2, 3, 4, 2, 5)
      .takeWhile(x -> x < 3)
      .forEach(x -> System.out.print(x + " "));    //1 2
    System.out.println();
```

```
Stream.of(1, 2, 3, 4, 2, 5)
      .dropWhile(x -> x < 3)
      .forEach(x -> System.out.print(x + " "));  //3, 4, 2, 5
    System.out.println();
```

Java API надає неузагальнені інтерфейси `IntStream`, `LongStream` та `DoubleStream`, які підтримують стріми `Integer`, `Long` та `Double`, відповідно.

Наприклад, їх методи можуть генерувати стріми:

```
IntStream.range(0, 10)
         .forEach(x -> System.out.print(x + " "));
    System.out.println();      //0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Дозволяють простіше використовувати перетворення типів:

```
IntStream ints = Stream.of(new Car("Buick", "Regal", 25),
                           new Car("Hyundai", "Elantra", 27),
                           new Car("Buick", "Skylark", 26),
                           new Car("Hyundai", "Accent", 30))
                                                  //Stream<Car>
                           .mapToInt(x -> x.mpg);  //IntStream
```

Дозволяють виконувати перетворення в інші типи (`Double`, `Long`, `Object`):

```
String s2 = Stream.of(2, 0, 1, 3, 2)
    .collect(Collectors.flatMapping(
      x -> IntStream.range(0, x).mapToObj(Integer::toString),
        Collectors.joining(", ")
    ));    // 0, 1, 0, 0, 1, 2, 0, 1
```

Методи `max` та `min` обчислюють найбільший та найменший елементи у потоці відповідно. Методи повертають об'єкти `OptionalInt`, `OptionalLong` та `OptionalDouble`, відповідно.

```
IntStream intsm = Stream.of(new Car("Buick", "Regal", 25),
                           new Car("Hyundai", "Elantra", 27),
                           new Car("Buick", "Skylark", 26),
                           new Car("Hyundai", "Accent", 30))
                                                  //Stream<Car>
                           .mapToInt(x -> x.mpg);  //IntStream
    intsm.max()
         .ifPresent(x -> System.out.println(x));          //30

LongStream.of(1, 2, 3, 4)
          .min()
          .ifPresent(x -> System.out.println(x));          //1
```

Метод `average` обчислює середнє значення елементів потоку і повертає `OptionalDouble`:

```
DoubleStream.of(1.1, 2.2, 3.3, 4.4)
        .average()
        .ifPresent(x -> System.out.println(x));         //2.75
```

Метод `sum` обчислює суму елементів потоку і повертає її int, double або long значення у залежності від типу стріму:

```
int sum = IntStream.of(1, 2, 3, 4)
        .sum();
System.out.println(sum);                                    //10
```

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. При вирішенні завдань використовуйте функціональні методи інтерфейсів та класів.

V	Завдання	V	Завдання
1	Використайте метод `forEachRemaining` для створення рядка, що містить розділений комами список моделей зі списку автомобілів, `List<Car> cars` в розділі 1. Теоретичні відомості.	12	Для класу `Car` з 1 сторінки розділу 1. Теоретичні відомості: - створіть `Map<String, Car>`, використовуйте методи інтерфейсу `Map`, щоб зробити наступне: - додайте Hyundai Excel із серійним номером “S123”; - додайте Buick Skylark із серійним номером “S456” - додайте Toyota Prius із серійним номером “S789” - змініть марку запису “S123” на “Chevy” - змініть модель запису “S123” на “Vega” - видаліть Toyota. Роздрукуйте отриманий `Map`.
2	Напишіть програму, яка використовує реалізацію інтерфейсу `PrimitiveIterator.OfDouble` для обчислення та друку поточної суми елементів (суми елементів після кожної ітерації) у масиві `double`.	13	Словник реалізований як `Map`, що містить наступні записи: **key**: automobile **value**: ground-powered vehicle with wheels; **key**: boat **value**: vehicle that travels on water; **key**: airplane **value**: powered vehicle that flies. Використовуючи методи `Map`, перекладіть визначення на іспанську мову таким чином: automobile: vehiculo tierra-accionado con las ruedas boat: vehiculo que viaja en el agua airplane: vehiculo motorizado que vuela
3	Для списку з 16 цілих, згенерованого у діапазоні від 1 до 10 випадковим чином виконайте зведення до кубу елементів двох частин списку, використовуючи методи інтерфейсу `Splitera-tor<T>`. Роздрукуйте результуючі значення елементів.	14	База даних складається з `Map` із записами, ключем яких є назва книги, а значення яких є об’єктом, що складається з автора книги, ціни та кількості доступних копій. База даних спочатку містить такі книги: Назва Автор Ціна В наявності

V	Завдання	V	Завдання
			Мобі-Дік Герман Мелвілл $19.99 25 *Дзен програмування на Java* Java Дж. Гуру $15.99 5 *Франкенштейн* Мери Шелли $12.99 10 Використовуючи методи `Map`, виконайте такі зміни бази даних: - видаліть книгу "Франкенштейн"; - додайте книгу "Дракула" Брема Стокера за 14,99$ із 13 примірниками; - змініть ціну програми "Дзен програмування на Java" на 13,99$; - змініть кількість доступних копій "Мобі-Дік" на 24. Роздрукуйте `Map`.
4	Створіть клас `MyDouble`, який інкапсулює масив `double` та реалізує `Iterable<Double>`. Продемонструйте обхід масиву в основній програмі.	15	Напишіть компаратор, який порівнює два цілих числа на основі їх природного впорядкування, але розглядає всі парні цілі числа як рівні
5	Створіть клас, який виконує обхід символів рядка. Використовуйте ланцюжок методів, щоб перетворити кожен символ у рядку у верхній регістр, а потім надрукувати символ.	16	Даний клас: `class A implements Comparable<A> { String s1; String s2; int i; public A(String a, String b, int c) { s1 = a; s2 = b; i = c; } @Override public String toString() { return s1 + " " + s2 + " " + i; } @Override public int compareTo(A a) { return s1.compareTo(a.s1); } }` напишіть такі компаратори: - компаратор, який порівнює два об'єкти `A`; - компаратор, який порівнює два об'єкти `A` за їх зворотним упорядкуванням; - компаратор, який порівнює два об'єкти `A` за зворотним упорядкуванням поля `s2`; компаратор, який порівнює два об'єкти `A` за природним упорядкуванням поля `i`. Продемонструйте компаратори в основній програмі.

V	Завдання	V	Завдання
6	Створіть клас, який буде виконувати ітерацію масиву `int` у зворотному порядку. Продемонструйте цю функціональніть в основній програмі	17	Орієнтуючись на клас `A` з завдання 16, виконайте багаторівневе сортування наступного списку об'єктів `A`: спочатку за полем `s1`, потім за зворотним упорядкуванням поля `s2`, а потім за полем `i`: Sally Jones 3 Sally Seashell 4 Harry Jones 1 Harry Homeowner 5 Harry Homeowner 2 Виведіть список на екран
7	Для 5 студентів розрахуйте та побудуйте їх рейтинг за оцінками з 6 дисциплін, які задайте випадково - від 3 до 5. Побудуйте `Map`, який у якості ключа отримує ПІБ студента, а у якості значення - середній бал. Відсортуйте `Map` за зменшенням середнього балу.	18	Створіть гру «камень, ножниці та папір», яка використовує компаратор для порівняння вибору гравця з вибором комп'ютера згідно з наступними правилами: — камень розбиває ножниці; — ножниці ріжуть папір; — папір огортає камінь; — усі інші поєднання - це нічия. Напишіть `Supplier`, який запропонує користувачеві зробити його вибір, інший `Supplier`, який випадково генерує вибір комп'ютера, і третій `Supplier`, який запропонує користувачеві грати знову. Зберігайте рахунок переможця кожного раунду. Надрукуйте остаточний рахунок та загального переможця в кінці гри.
8	Розробіть список з 5 об'єктів класу Student, що містить ПІБ та стать студента. За допомогою методу `removeIf` отримайте окремі списки студентів чоловічої та жіночої статі.	19	Колода гральних карт спочатку відсортована за масттю, а потім за номіналом. Іншими словами, першою картою є чирвова двійка, за якою йдуть чирвова трійка і так до чирвового туза, потім трефова двійка, потім трефова трійка і так далі тощо. Напишіть компаратор, який сортує колоду за номіналом і потім за мастю. Іншими словами, першою картою повинна бути чирвова двійка, за якою трефова двійка, потім пікова двійка, потім бубнова двійка, потім чирвова трійув і так далі
9	За допомогою метода `setAll` класу `Arrays` заповніть кожен елемент масиву `int` значенням, що дорівнює довжині масиву мінус індекс елемента. Потім за допомогою методу `removeIf`	20	Створіть `Supplier`, який повертає об'єкт `Optional`. `Supplier` пропонує користувачеві вибрати число від 1 до 5 або ввести "Q" для виходу. Об'єкт `Optional` повинен містити ціле число, коли вибрано число, і містити `null`,

V	Завдання	V	Завдання
	видаліть елемент, значення якого дорівнює 1. Роздрукуйте отриманий масив.		коли введено “Q”. Викликайте об’єкт Supplier в циклі, який друкує вміст Optional, якщо він заповнений. Цикл припиняється, коли Optional має значення null. Продемонструйте в основній програмі.
10	За допомогою метода setAll класу Arrays заповніть кожен елемент масиву int з 10 елементів випадковими значеннями від 1 ло 5. Створіть об’єкт-список з згенерованого масиву. Виконайте заміну значень елементів списку на квадрти їх значень. Розрукуйте список-результат.	21	Використовуючи ланцюжок викліків методів Optional, зробіть наступне: - створіть Optional, що містить рядок "Optionals є опціональними"; - змініть вміст Optional на "Optionals є необов’язковими"; - перевірте, чи довжина рядка, що міститься в Optional, перевищує 20 символів - роздрукуйте вміст Optional.
11	Створіть Map<Integer, Integer>. Встановіть ключі та відповідні значення ключі у однакові значеня від 1 до 5. Використовуючи метод compute інтерфейсу Map, помножте на 3 значення всіх записів на карті, ключі яких містять непарні значення. Роздрукуйте отримані записи.	22	Служба внутрішніх доходів дозволяє вказати ваш дохід як найманого працівника та/або як самозайнятої особи. Використовуючи Optional та функціональні інтерфейси, напишіть програму, яка дозволяє користувачеві вводити свій дохід як нйманому працівнику або як самозайнятій особі, а потім друкує отриманий дохід.

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. Вкажіть функціональні інтерфейси, які були задіяні при виконанні завдання та поясніть їх роботу.
4. 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 стрім, що містить імена “Роберт”, “Шеріл”, “Рейчел”, “Хосе” та “Ріта”. Видаліть усі імена, які не починаються з літери Р. Відсортуйте решту елементів за зворотним упорядкуванням, а потім роздрукуйте результати.	12	Створіть стрім, що містить 5 рядків та перевірте, чи унікальні вони.
2	Створіть стрім, що містить цифри від 1 до 5. Додайте по 4 до кожного елемента стріма. Потім створіть редукцію, яка додає елементи з непарними	13	За допомогою методу String.chars() отримайте стрім, у якому обчисліть значення хеш-коду для рядка з латинських символів за

V	Завдання	V	Завдання
	значеннями і подвоює парні значення елементів. Надрукуйте результат редукції.		стандартним алгоритмом: `h = 31 * h + (v & 0xff)`, де `v` - значення байту символа. Перевірте правильність розрахунку порвнянням з результатом методу `hashcode()` для рядка.
3	Створіть стрім символів `'1' 'a' '5' 'v'` та зберіть (collect) їх у `Map`, ключі якої є символами, а значення - цілочисельним значенням кожного символу (наприклад, `'1' - 49`, а `'a' – 97`). Роздрукуйте отриманий `Map`.	14	Для списку рядків довільної кількості розробіть використовуючи стрім метод, що розбиває вихідний список на списки-частини з кількістю елементів, вказаній як параметр метода (останній частина-список може бути неповним). Перевірте роботу розробленого метода.
4	Створіть стрім з `10` елементів що містить випадкові цілі від `1` до `5` та підрахуйте суму та середнє непарних значень елементів.	15	Користуючись методом `Files.lines()`, що повертає стрім рядків, виведіть на екран рядки, що містить тільки цифри через символ двокрапки. Перевірте роботу метода.
5	Для абзацу тексту з декількома реченнями за допомогою перетворення у стрім виконайте заміну першого слова кожного речення на теж слово великими літерами.	16	Розробіть програму, використовуючи `IntStream`, яка буде генерувати послідовність простих чисел у заданому двома параметрами діапазоні. Перевірте роботу програми.
6	Для довільного текстового файлу з допомогою стріму виберіть рядки, де є слова з великої літери та виконайте заміну порядку літер у таких словах на зворотній. Роздрукуйте отриманий текст.	17	За допомогою стрімів заповніть випадковими цілими числами в діапазоні від 1 до 5 три масиви з двома елементами кожний. Виведіть послідовність унікальних значень з цих масивів.
7	За допомогою стріму складіть список моделей автомобілів, розділених комами, за алфавітом за такими даними: Make Model MPG Buick Regal 25 Hyundai Elantra 27 Buick Skylark 26 Hyundai Accent 30	18	За допомогою стріму для списку фраз з двох-трьох слів виведіть відсортований список унікальних слів, які використовуються у фразах.
8	Розробіть за допомогою стріму метод, який генерує випадковим чином 10 чисел `Double` у диапазоні від 1 до 10 (не включаючи 10) та повертає суму їх дробових частин. Перевірте роботу метода.	19	За допомогою стріму для списку об'єктів класу `Person`, що містить ПІБ та стать за допомогою колектора угрупуйте персони у два списки за статтю. Роздрукуйте результат
9	Користуючись методом `Files.walk`, що повертає стрім об'єктів `Path`, отримайте список `List<String>` імен файлів з заданим як параметр розширенням та каталогом, у якому містяться файли.	20	Розробіть за допомогою стріму метод, у якому список рядкових представлень чисел обробляється і виводиться максимальне число.
10	Для списку об'єктів класу `Student`, який містить ПІБ студента та його	21	Для списку об'ктів-покупок виконайте за допомогою стріму розділення списку

V	Завдання	V	Завдання
	середній бал за допомогою стріму виконайте обчислення середнього балу групи студентів у списку, а також виведіть студента з максимальним та мінімальним середнім балом.		покупок на списки покупок продовольчих та непродовольчих товарів. Роздрукуйте результуючі списки.
11	Створіть стрім з 5 довільник рядків та за допомогою компаратора отримайте список рядків з максимальною довжиною	22	Є два списки - список унікальних цілих чисел та список довільних букв. Виконайте за допомогою стріму та методу `Collectors.partitioningBy` поділ першого списку на парні та непарні числа, а букв - на голосні та приголосні.

5. Наведіть код програми у звіт. У висновку вкажіть особливості роботи зі стрімами.

3. Контрольні питання

1. Який метод доданий до інтерфейсу `Iterator<E>` в Java 8? Наведіть приклад його використання.
2. Для яких примітивних типів може бути використаний інтерфейс `PrimitiveIterator<T, T_CONS>`? Приведіть приклад його використання для масиву цілих чисел.
3. Які Ви знаєти спеціалізовані примітивні ітератори, що дозволяють обходити масиви примітивних типів? Приведіть приклад використання такого ітератору для обходу масиву цілих чисел.
4. Для чого служить інтерфейс `Spliterator<T>`? Опишіть його методи та приведіть приклад його використання.
5. До якого інтерфейсу Java 8 був доданий метод `default void forEach(Consumer<? super T> action)`? Приведіть приклад його використання.
6. Опишіть метод `forEach(BiConsumer<? super K,? super V> action)` інтерфейсу `Map<K, V>` та приведіть приклад його використання.
7. Опишіть метод `removeIf(Predicate<? super E> filter)` інтерфейсу `Collection<E>` та приведіть приклад його використання.
8. Опишіть методи `setAll`, додані в Java 8 до утилітарного класу Arrays. Наведіть приклади їх використання для заповнення масивів.
9. Опишіть методи `replaceAll` інтерфейсів `List<E>` і `Map<K, V>` та приведіть приклади їх використання.
10. Опишіть методи `parallelPrefix` класу `java.utils.Arrays` т приведіть приклад їх використання.
11. Опишіть методи, додані до інтерфейсу `Map<K, V>`, які дозволяють виконувати обчислення на записах, та приведіть приклад їх використання.
12. Опишіть методи, додані до інтерфейсу `Comparator<T>`, які використовують функціональні інтерфейси, та приведіть приклади їх використання.

13. Опишіть спеціалізації методів порівняння інтерфейсу `Comparator<T>` та приведіть приклад їх використання.
14. Опишіть методи інтерфейсу `Comparator<T>`, які дозволяють проектувати ланцюжки викликів методів компараторів та приведіть приклад їх використання.
15. Опишіть спеціалізації методів інтерфейсу `Comparator<T>`, які дозволяють проектувати ланцюжки викликів методів компараторів та приведіть приклад їх використання.
16. Опишіть методи `maxBy` та `minBy` функціонального інтерфейсу `BinaryOperator<T>`, які порівнюють два об'єкти типу `T` на основі `Comparator<T>` та наведіть приклад їх використання.
17. Що собою являє клас `Optional<T>`, які переваги його використання? Опишіть на прикладах методи створення об'єкту `Optional<T>`.
18. Опишіть на прикладах методи класу `Optional<T>` для перевірки та отримання об'єктів, які він огортає.
19. Опишіть на прикладах методи класу `Optional<T>` для створення ланцюжків виклику методів `Optional<T>`.
20. Опишіть використання методу `filter` класу `Optional<T>` та приведіть приклад його використання.
21. Опишіть використання методу `map` класу `Optional<T>` та приведіть приклад його використання.
22. Опишіть використання методу `flatMap` класу `Optional<T>` та приведіть приклад його використання. У чому відмінність цього методу від методу `map`?
23. Назвіть призначення інтерфейсу `Stream<T>` та дайте стислу характеристику його методам. Які типи об'єктів `Stream<T>` існують?
24. Опишіть використання методів створення об'єктів `Stream<T>` та наведіть приклади їх використання.
25. Опишіть способи створення об'єктів `Stream<T>` із інших об'ктів та наведіть приклади їх використання. Які особливості роботи таких об'єктів Ви знате?
26. Назвіть відомі Вам конвейерні методи інтерфейсу `Stream<T>`. Наведіть приклад використання одного з них.
27. Назвіть відомі Вам термінальні методи інтерфейсу `Stream<T>`. Наведіть приклад використання одного з них.
28. Опишіть метод інтерфейсу `Stream<T>`, що забезпечує фільтрацію елементів стріма та наведіть приклад його використання.
29. Опишіть методи інтерфейсу `Stream<T>`, що забезпечують сортування елементів стріма та наведіть приклади їх використання.
30. Опишіть методи інтерфейсу `Stream<T>`, що повертають елемент з мінімальним та з максимальним значенням та наведіть приклади їх використання. Який тип повертають ці методи?

31. Опишіть методи `map` та `flatMap` інтерфейсу `Stream<T>` та наведіть приклади їх використання. Чим різняться ці методи?
32. Що так редукція об'єкту `Stream<T>`? Опишіть перевантажені методи `reduce` інтерфейсу `Stream<T>`, як працює функціональний метод його параметра-акумулятора? Коли корисною є версія методу з параметром-об'єднувачем (combiner)? Наведіть приклад використання одного з методів.
33. Що таке колектор і як такий об'єкт відноситься до інтерфейсу `Stream<T>`? Приведіть приклад використання методу `collect`.
34. Опишіть інтерфейс Collector<T, A, R> та його методи і енумератор. Чи можна вносити зміни вмісту контейнера результатів, який отриманий за допомогою колектора? Наведіть приклад створення та застосування користувацького колектора.
35. Опишіть метод `collect` інтерфейсу Stream<T>, який приймає як параметри постачальника, накопичувача та об'єднувача без створення об'єкта Collector. Наведіть приклад використання такого методу.
36. Наведіть приклад створення та використання колектора з вхідним типом StringBuilder, який виконує об'єднання рядків з попередньою їх модифікацією та перетворює об'єднаний об'єкт StringBuilder на об'єкт String.
37. Надайте характеристику класу java.util.stream.Collectors, назовіть його основні методи та навіедіть приклад використання одного з них.
38. Наведіть приклади методів класу java.util.stream.Collectors, які накопичують елементи стріму у списку, у множині, у довільній колекції та у карті (Map). Як працює метод `Function.identity()` при використанні у методі `Collectors.toMap`?
39. Опишіть метод `collectingAndThen` класу java.util.stream.Collectors. Наведіть приклад використання такого методу.
40. Опишіть метод `reducing` класу java.util.stream.Collectors. Наведіть приклад використання такого методу.
41. Опишіть метод `joining` класу java.util.stream.Collectors. Наведіть приклад використання такого методу.
42. Опишіть метод `groupingBy` класу java.util.stream.Collectors. Наведіть приклад використання такого методу.
43. Опишіть метод `partitioningBy` класу java.util.stream.Collectors. Наведіть приклад використання такого методу.
44. Опишіть методи `summingT`, `averagingT`, `summarizingT` (де `T` - `Int` або `Double` або `Long`) класу java.util.stream.Collectors. Наведіть приклади використання цих методів.
45. Опишіть інтерфейс `Stream.Builder` та продемонструйте його можливості для побудови стрімів. Чи можлива модифікація створеного засобами цього інтерфейсу стріму?
46. Опишіть методи `peek`, `takeWhile`, `dropWhile` інтерфейсу java.util.stream.Stream<T>. Наведіть приклади використання цих методів.

47. Опишіть неузагальнені інтерфейси `IntStream`, `LongStream` та `DoubleStream`, які підтримують стріми `Integer`, `Long` та `Double`, відповідно. На прикладі одного з інтерфейсів наведіть використання його методів для створення стріму, перетворення в інший примітивний та в об'єктний типи, пошуку максимального та мінімального елементу, обрахунку суми та середнього елементів стріму.

4. Литература

1. Шилдт Г. Java. Полное руководство, 10-е изд. / Пер. с англ. Москва : Вильямс, 2019. 1488 с.
2. Lecessi R. Functional Interfaces in Java: Fundamentals and Examples, APress, 2019. 415 с.
3. Уорбэртон Р. Лямбда-выражения в Java 8. Функциональное программирование – в массы / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2014. – 192с.
4. Функциональные интерфейсы в Java 8. URL: https://javanerd.ru/основы-java/функциональные-интерфейсы-в-java-8/. (дата звернення 30.08.2020).
5. Полное руководство по Java 8 Stream API в картинках и примерах. URL: https://annimon.com/article/2778#top (дата звернення 30.08.2020).