**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»**

**ім. Ігоря Сікорського»**

Кафедра системного проектування

ЛАБОРАТОРНА РОБОТА №4

з дисципліни

" **ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ**"

на тему: «Розробка і реалізація паралельного алгоритму з використанням механізмів асинхронних обчислень»

Виконав:

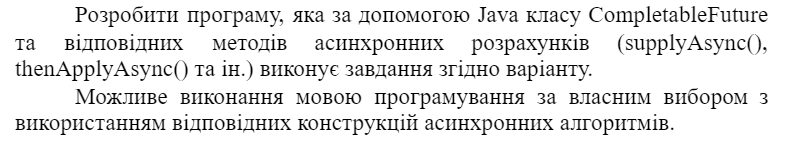
Студент ІІІ курсу

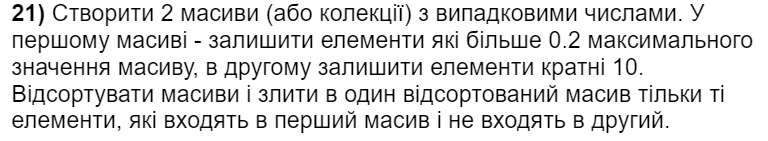
групи ДА-81

Нікітін Б. Д.

Київ-2021

# Завдання





# Опис виконання

В лабораторній я використовуючи Java class CompatableFuture виконав власне завдання.

Коли ми викликаємо метод supplyAsync, то Future виконається у ForkJoinPool.commonPool(). За допомогою методу thenApplyAsync ми задаємо функцію, що виконається після успішного завершення нашого Future.  
І в кінці за допомогою методу thenCombine, після успішного виконання 2 Future ми обробляємо їх заданою функцією.

# Результат роботи



Результат вірний.

# Програмна реалізація

import java.util.\*;  
import java.util.concurrent.CompletableFuture;  
import java.util.concurrent.ExecutionException;  
import java.util.stream.Collectors;  
  
public class Main {  
  
 public static void main(String[] args) {  
 final List<Integer> a1 = Arrays.*asList*(0, 9, 6, 2, 5, 1, 7, 4, 3, 8, 10, 20); //first  
 final List<Integer> a2 = Arrays.*asList*(8, 7, 0, 1, 3, 6, 9, 2, 5, 4, 10); //second  
 CompletableFuture<List<Integer>> firstFuture, secondFuture, resultFuture;  
 firstFuture = CompletableFuture  
 .*supplyAsync*( // Возвращает новый объект CompletableFuture, который асинхронно завершается задачей, выполняемой в данном исполнителе, со значением, полученным путем вызова данного поставщика.  
 () -> a1.stream()  
 .parallel()  
 .filter(element -> element > 0.2 \* Collections.*max*(a1)) // фильтруем наш List с условием > 0.2 максимального элемента  
 .collect(Collectors.*toList*()))  
 .thenApplyAsync(first -> { // Возвращает новый объект CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента предоставленной функции.  
 Collections.*sort*(first); // сортируем наш List  
 return first;  
 });  
 secondFuture = CompletableFuture  
 .*supplyAsync*( // Возвращает новый объект CompletableFuture, который асинхронно завершается задачей, выполняемой в данном исполнителе, со значением, полученным путем вызова данного поставщика.  
 () -> a2.stream()  
 .parallel()  
 .filter(element -> (element % 10) == 0) // фильтруем наш List с условием кратности 10  
 .collect(Collectors.*toList*()))  
 .thenApplyAsync(second -> { // Возвращает новый объект CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента предоставленной функции.  
 Collections.*sort*(second); // сортируем наш List  
 return second;  
 });  
 resultFuture = firstFuture  
 .thenCombine(secondFuture, // Возвращает новый объект CompletionStage, который при нормальном завершении этого и другого заданного этапа выполняется с двумя результатами в качестве аргументов предоставленной функции.  
 (first, second) -> {  
 List<Integer> a3 = new ArrayList<>(first);  
 return a3.stream()  
 .parallel()  
 .filter(element -> !second.contains(element)) // фильтруем наш List с условием отсутствия этого элемента в List second  
 .collect(Collectors.*toList*());  
 });  
 try {  
 System.*out*.println("Result: " + resultFuture.get()); /\*blocks until future completes\*/  
 } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
}