Лабораторна робота №2

Основи TDD та JUnit.
Основні алгоритмічні структури мови Java

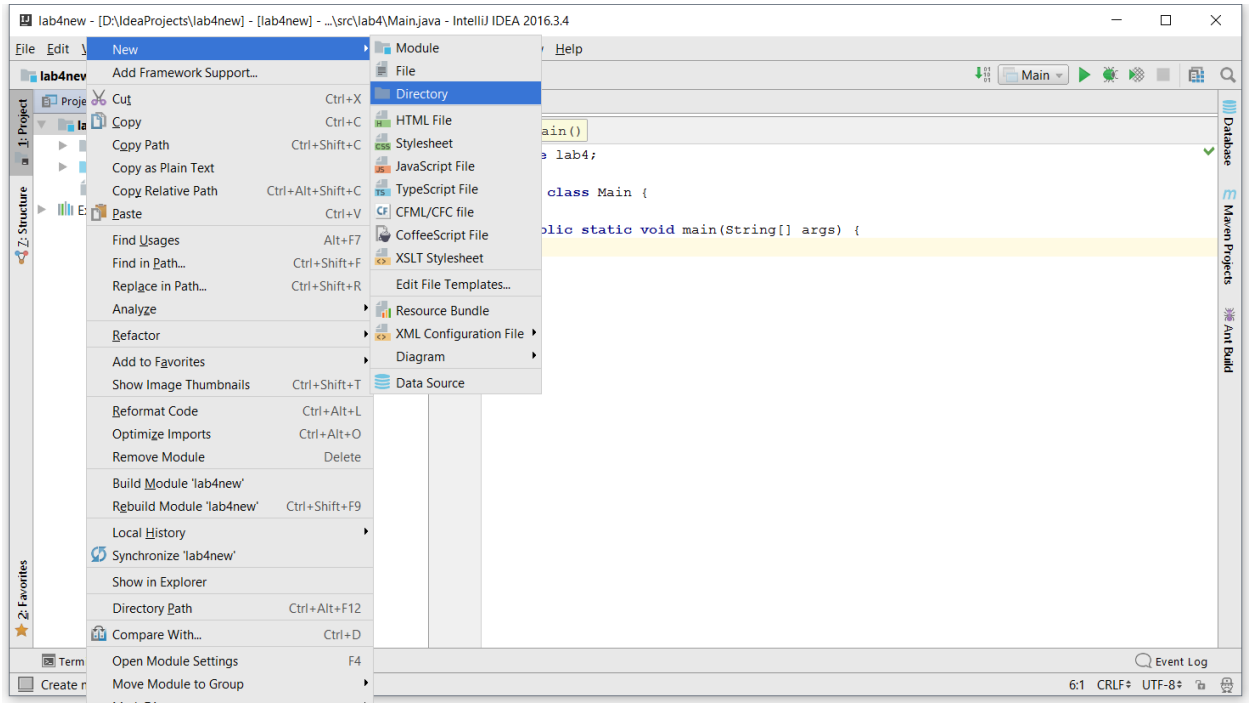
1. У середовищі IntelliJ IDEA створити новий проект, що містить один головний клас Main.
2. Створити тестовий клас для тестування головного класу програми.
3. У головному класі описати метод, що обчислює значення функції, яка задана у таблиці і у тестовому класі - тестові методи для нього. Діяти у такій послідовності: спочатку створити один тестовий метод, згенерувати метод обчислення функції. Виконати тестування та пересвідчитись, що тест працює, тобто тестування згенерованого метода повинно завершитися «помилкою». Реалізувати метод. Виконати тестування. Пересвідчившись, що тест проходить, створити ще декілька тестових методів для метода обчислення функції. Виконати тестування.
4. Розробити метод, що за вказаними значеннями кроку, початку та кінця інтервалу обчислює кількість кроків для табулювання та тестові методи для нього і виконати тестування (порядок дій см. у п.3).
5. Створити методи, що створюють масиви значень функції (y) та її аргументу (x) в усіх точках вказаного інтервалу із заданим кроком. (розмір масивів обчислити програмно за допомогою метода з п.4).

   Створити тестові методи для них і виконати тестування (порядок дій – см п.3).
6. Створити методи, які після формування масивів, знаходять номери найбільшого та найменшого елементів масиву значень функції, та методи, що обчислюють та суму та середнє арифметичне елементів масиву значень функції. Методи створювати разом з тестами та постійно виконувати тестування.
7. Створити методи виведення найбільшого та найменшого елементів масиву значень функції, вказавши їхні номери і відповідні значення аргументу.
8. Дописати у створеному класі метод main, перетворивши, таким чином, його на автономну програму. Скомпілювати і виконати програму

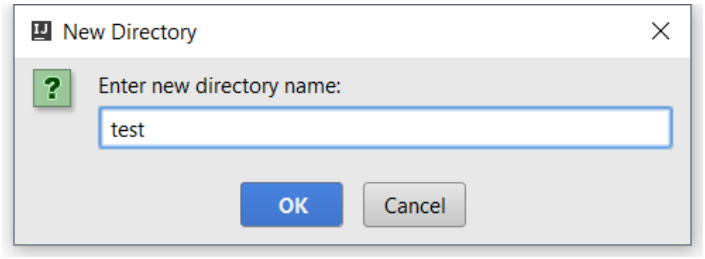
Короткі теоретичні відомості

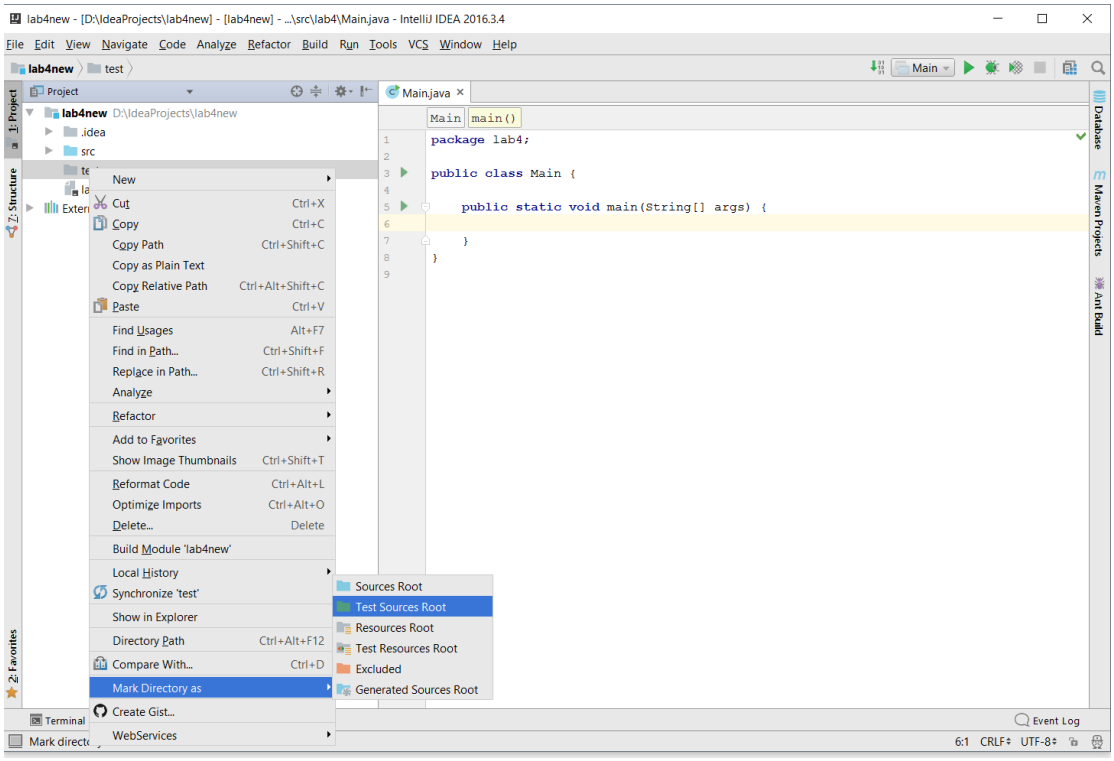
Створення класу для модульного тестування у IntelliJ IDEA

Для створення класу модульного тестування у середовищі IntelliJ IDEA треба спочатку створити каталог для розміщення тестових класів (ПКМ на проекті, вибрати New → Directory та вказати ім’я Test).

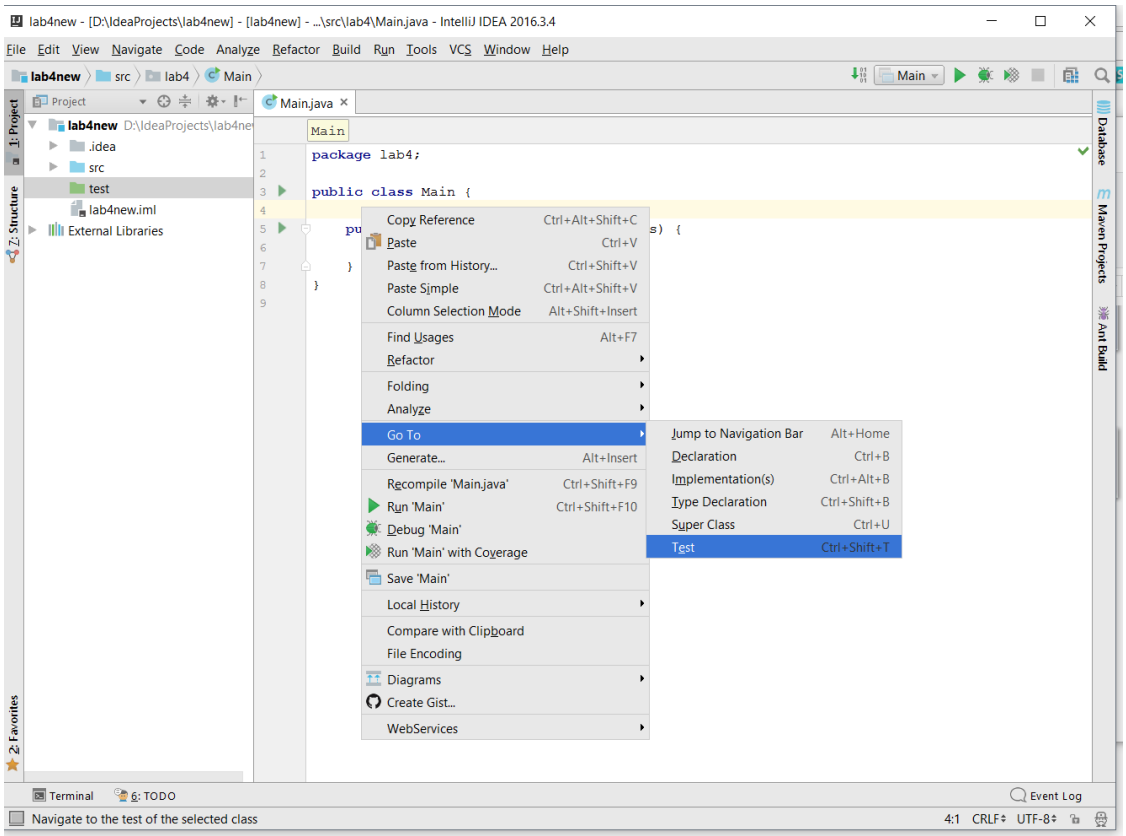


Створений каталог слід помітити, як призначений для тестів (ПКМ на ньому, вибрати Mark Directory As → Test Sources Root).

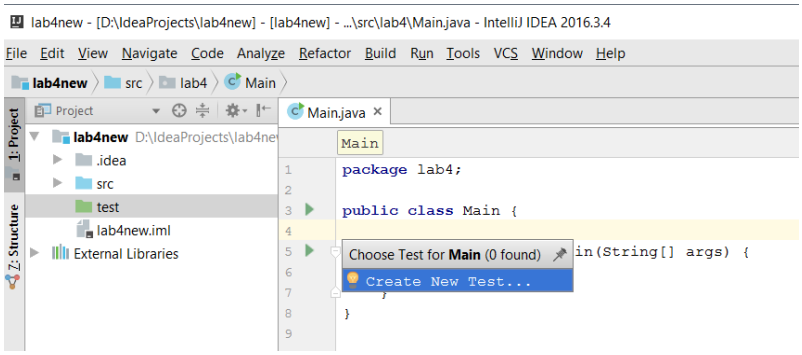




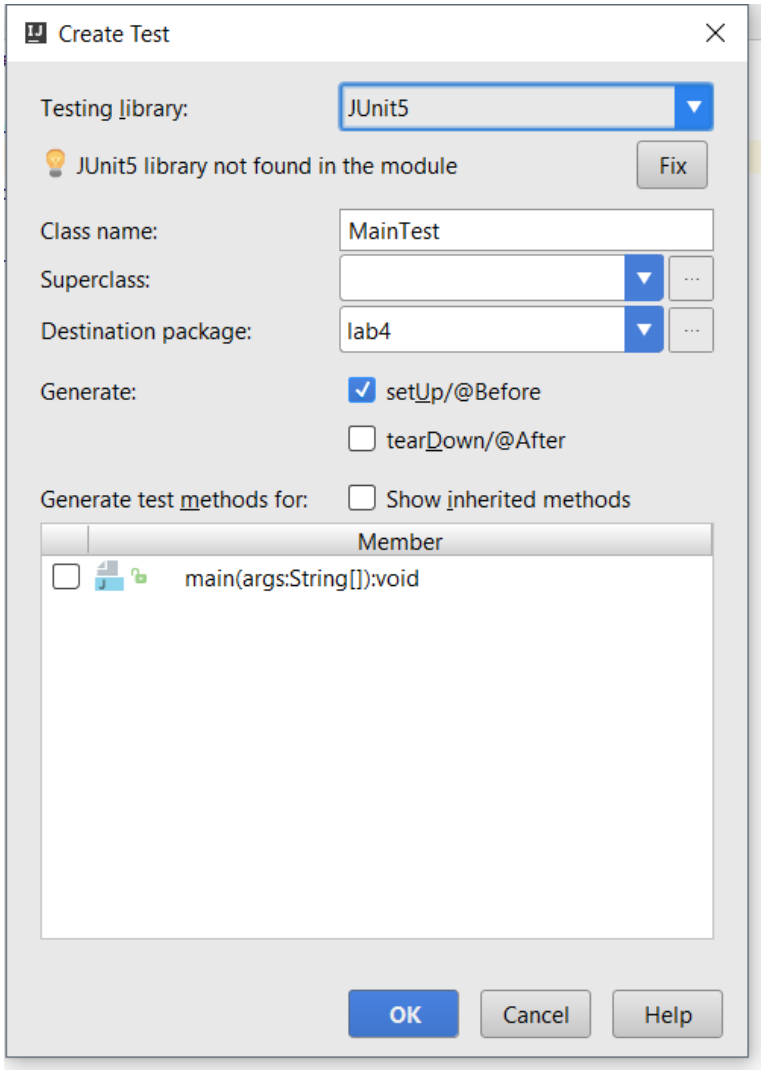
Далі, у редакторі коду натиснути ПКМ та вибрати Go To → Test (Ctrl+Shift+T).



Вибрати Create New Test…

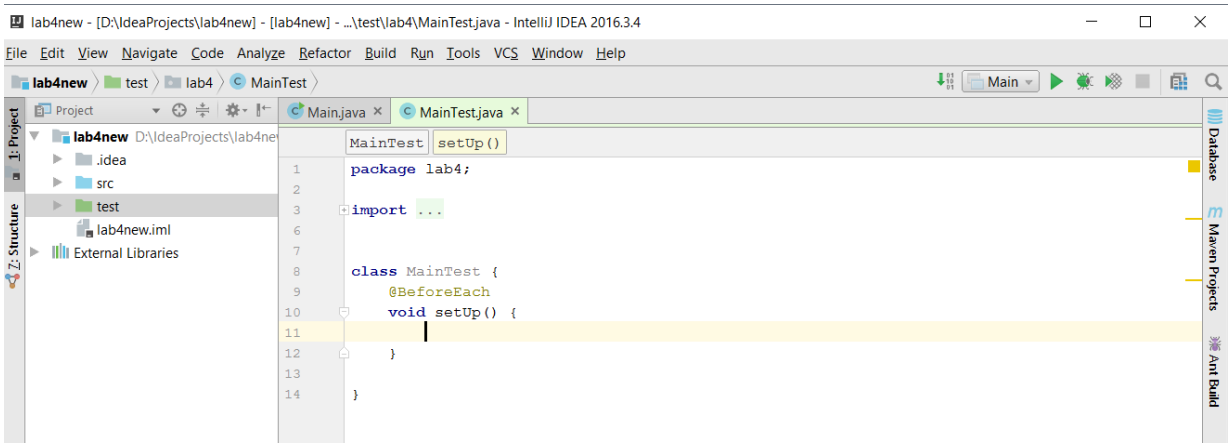


У діалоговому вікні, що відкриється, вибрати JUnit 5.

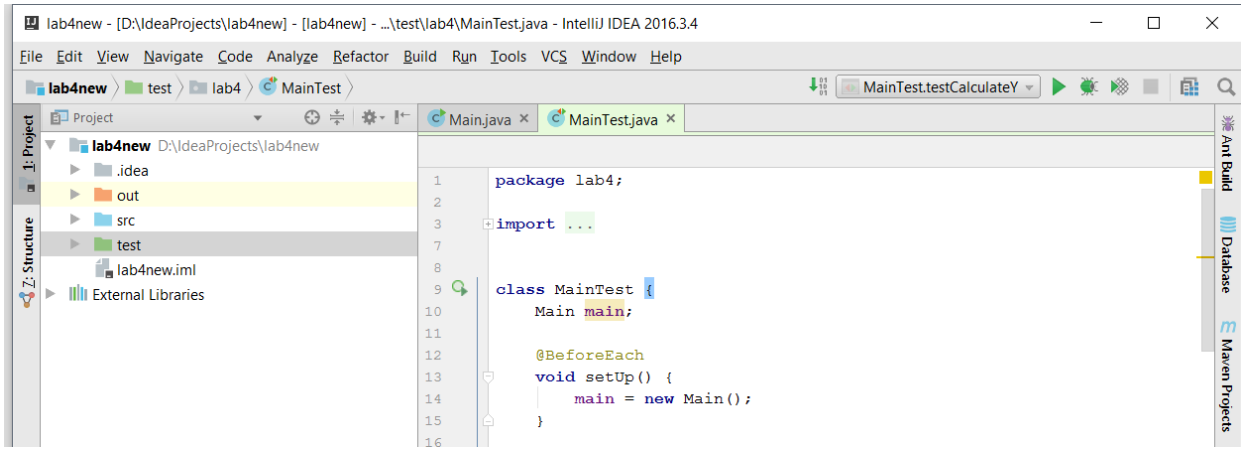


Якщо у головному класі вже описані методи, які потрібно тестувати, треба їх відмітити. Після цього натиснути кнопку Fix (щоб додати бібліотеку тестування JUnit 5 до поточного модуля проекту).

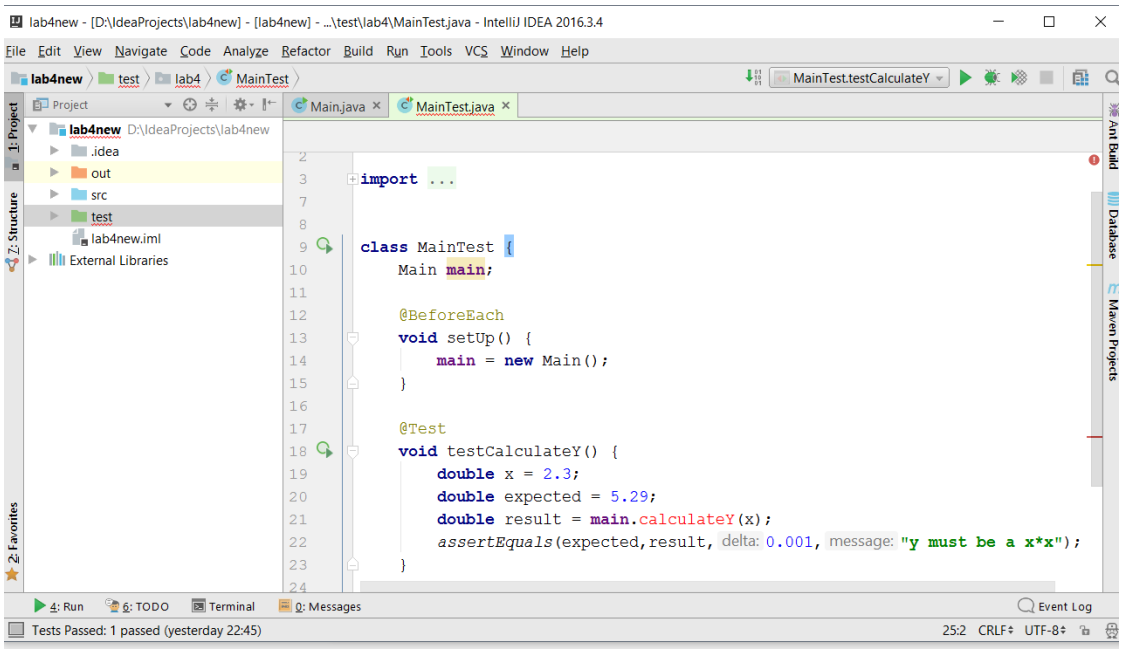
Натиснути Ok для створення тестового класу.



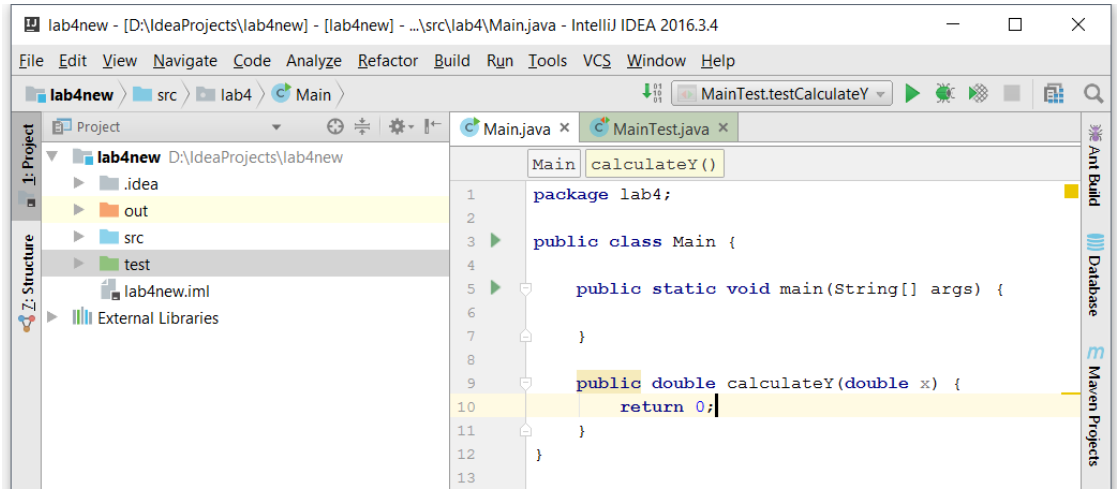
У тестовому класі, що був згенерований, додати описання головного класу, та виклик його конструктора у методі setUp.



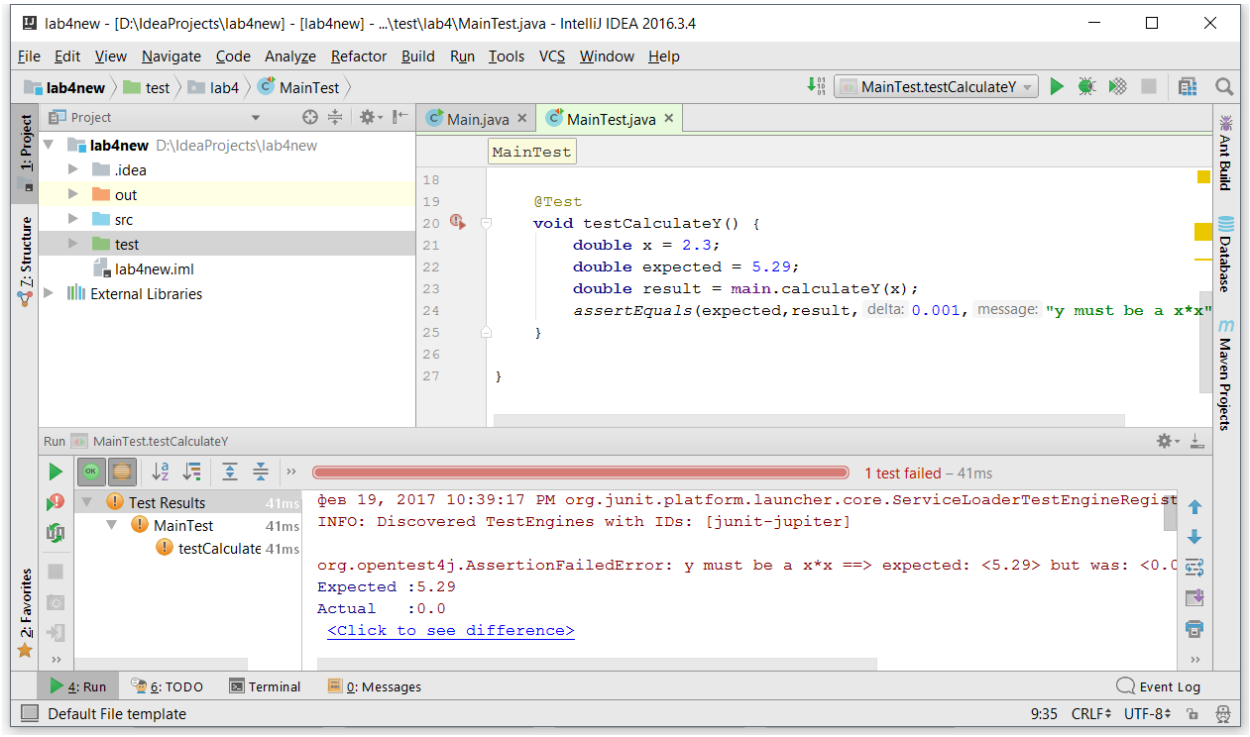
У тестовому класі описати метод для тестування майбутнього методу з основного класу:



Створити метод у основному класі натиснувши Alt+Enter:

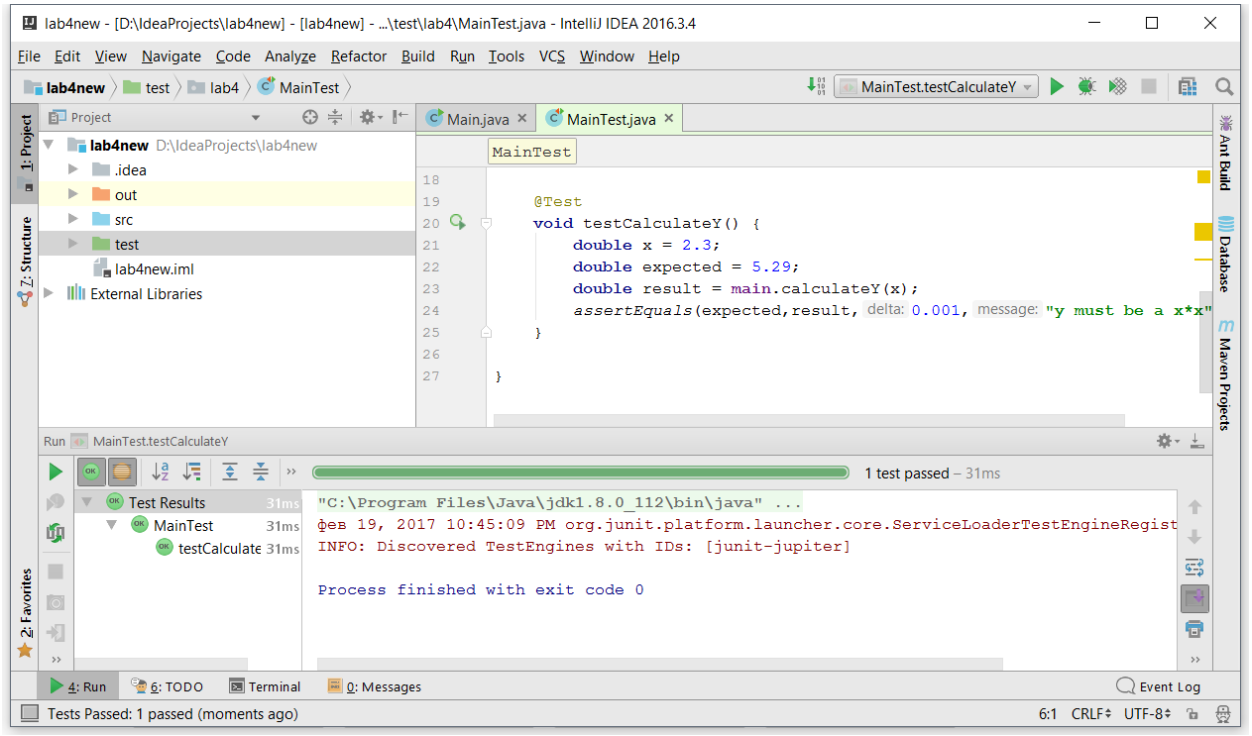


Повернутись до тестового файлу, та виконати тестування. Пересвідчитись, що тест завершиться помилкою.



Дописати створений у основному класі метод, та повторити тестування.

Пересвідчитись, що метод написаний правильно та тест завершується успіхом.



Для інших методів створювати тести аналогічно.

Управляючі структури мови Java, які відсутні у мові C++

У мові Java існують управляючі структури, аналогічні до тих, що є у C++, але тут розглянемо ті, що відсутні у C++:

1. ***switch*.**

   Конструкція **switch** у Java як і у C++ дозволяє передавати управління тому чи іншому блоку коду, що позначений іменованою міткою, в залежності від значення виразу. Загальний синтаксис **switch** можна представити таким чином:

```
switch (Вираз) {
        case n: Інструкції
        case m: Інструкції
        ...
        default: Інструкції
 }
```

   Тіло **switch**, відоме як блок перемикачів, містить набори інструкцій, яким передують мітки, що починаються з службового слова **case.** Кожній мітці **case**

ставиться у відповідність константа. Якщо значення виразу співпадає зі значенням деякої мітки, управління буде передано першій інструкції, що йде після цієї мітки. Якщо співпадінь не знайдено, виконуються інструкції блоку **default**. Якщо ж мітка **default** відсутня, виконання **switch** завершується. При передаванні управління відповідній мітці виконуються всі наступні за нею інструкції, навіть ті, що мають свої власні мітки **case.** Якщо треба вийти з блоку **switch**, треба використати інструкцію **break**. На відміну від C++, у Java у якості виразу та міток перемикачів, дозволяється використовувати не тільки цілі числа, а й рядки.

2. for (each)

Починаючи з версії Java 5 у мові Java з'явилась нова конструкція, призначена для виконання ітерації по масиву або колекції. Вона виглядає так:

```
for (<тип елементу> <формальне ім'я> : <масив>)
    Інструкція
```

3. Мітки

Інструкції програми можуть бути позначені мітками (labels). Мітка являє собою змістовне ім'я, що дозволяє посилатися на відповідну інструкцію:

```
Мітка: Інструкція
```

Звертатися до мітки дозволено тільки за допомогою команд **break** та **continue** (вони розглядатимуться далі).

4. break

Інструкція **break** застосовується для завершення виконання коду будь-якого блоку. Існують дві форми інструкції – безіменна:

```
break;
```

та іменована:

```
break мітка;
```

Безіменна команда **break** перериває виконання коду конструкцій **switch, for, while** або **do** і може використовуватися лише всередині цих конструкцій.

Команда **break** у іменованій формі може перервати виконання будь-якої інструкції, що помічена відповідною міткою.

Команда **break** найчастіше використовується для примусового виходу з тіла циклу. А для виходу із вкладеного циклу чи блоку, достатньо позначити міткою зовнішній блок і вказати її в інструкції **break** як показано в наступному прикладі:

Приклад. Використання поміченого **break**

```
private float[][] matrix;
    public boolean workOnFlag(float flag) {
        int y, x;
        boolean found = false;
        search:
        for (y = 0; y < matrix.length; y++) {
            for (x = 0; x < matrix[y].length; x++) {
                if (matrix[y][x] == flag) {
                    found = true;
                    break search;
                }
            }
        }
        if (!found) {
            return false;
        }
        // А тут знайдене значення matrix[y][x]
        // деяким чином обробляється
        return true;
    }
```

Відмітимо, що іменована інструкція **break** – це зовсім не те ж саме, що й сумнозвісна команда **goto**. Інструкція **goto** дозволяє ”стрибати” по коду без жодних обмежень, переплутуючи порядок обчислень і збиваючи читача з глузду. Команди же **break** і **continue,** що посилаються на мітку, дозволяють лише акуратно залишити відповідний блок і забезпечити його повторення, при цьому потік обчислень залишається цілком очевидним.

5. ***continue***. Команда **continue** застосовується лише у контексті циклічних конструкцій і передає управління на кінець тіла циклу. В ситуації з **while** і **do** це призводить до виконання перевірки умови циклу, а при використанні в тілі

for інструкція **continue** провокує передавання управління секції змін значень змінних циклу.

Як і **break**, команда **continue** дозволяє використання в двох формах – без імені: **continue;** і іменованій: **continue** мітка. Команда **continue** у формі без імені мітки передає управління в кінець поточного циклу, а іменована – в кінець циклу, позначеного відповідною міткою. Мітка повинна відноситися до циклічного виразу.

6. ***goto***. У мові Java НЕМАЄ інструкції **goto**, що має змогу передавати управління довільному фрагменту коду, хоча у споріднених мовах аналогічні засоби передбачені. Всі засоби, що були розглянуті раніше, дозволяють створювати зрозумілий і надійний код, а також обходитися без допомоги **goto**.

7. Для обробки виключень, тобто ситуацій, що могли б привести до краху програми (наприклад, ділення на нуль, помилка введення-виведення) використовують конструкцію **try…catch…finally…** Обробка виключень у Java спирається в основному, на конструкції C++ (хоча ідейно більше схожа на Object Pascal). У місці, де виникла проблема, ви, можливо, ще не знаєте що з нею робити, проте знаєте, що просто ігнорувати її не можна – треба зупинитись і передати управління блоку обробки.

Варіанти завдань

№	Функція	Умова	Вхідні дані	Діапазон та крок зміни аргумента	Номери елементів, для тестування
1	$y=\begin{cases} ax^2 \ln x \\ 1 \\ e^{ax}\cos bx \end{cases}$	$0.7 < x \le 1.4$ $x \le 0.7$ $x > 1.4$	$a=-0.5$ $b=2$	$x \in [0;3]$ $\Delta x = 0.004$	175, 350, 750
2	$y=\begin{cases} px^2 - 7/x^2 \\ ax^3 + 7\sqrt{x} \\ \lg(x+7\sqrt{x}) \end{cases}$	$x < 1.7$ $x = 1.7$ $x > 1.7$	$a=1.5$ $p=4$	$x \in [0.8;2]$ $\Delta x = 0.005$	0, 180, 240
3	$y=\begin{cases} ax^2 + bx + c \\ a/x + \sqrt{x^2+1} \\ (a+bx)/\sqrt{x^2+1} \end{cases}$	$x < 1.4$ $x = 1.4$ $x > 1.4$	$a=2.8$ $b=-0.3$ $c=4$	$x \in [0;2]$ $\Delta x = 0.002$	0, 700, 1000
4	$y=\begin{cases} px^2 - 7/x^2 \\ ax^3 + 7\sqrt{x} \\ \ln(x+7\sqrt{\lvert x+a \rvert}) \end{cases}$	$x < 1.3$ $x = 1.3$ $x > 1.3$	$a=1.65$ $p=7.2$	$x \in [0.7;2]$ $\Delta x = 0.005$	0, 120, 260
5	$y=\begin{cases} 1.5a\cos^2 x \\ (x-2)^2 + 6a \\ 3a \cdot tgx \end{cases}$	$x \le 0.3$ $0.3 < x \le 2.3$ $x > 2.3$	$a=2.3$	$x \in [0.2;2.8]$ $\Delta x = 0.002$	50, 1050, 1300
6	$y=\begin{cases} x\sqrt{x-a} \\ x\sin ax \\ e^{-ax}\cos ax \end{cases}$	$x > a$ $x = a$ $x < a$	$a=2.4$	$x \in [1;5]$ $\Delta x = 0.01$	0, 140, 400
7	$y=\begin{cases} bx - tg\,bx \\ bx + \lg bx \end{cases}$	$bx \le 0.45$ $bx > 0.45$	$b=1.5$	$x \in [0.1;1]$ $\Delta x = 0.001$	0, 200, 900
8	$y=\begin{cases} \sin x \lg x \\ \cos^2 x \end{cases}$	$x > 3.4$ $x \le 3.4$		$x \in [2;5]$ $\Delta x = 0.005$	0, 280, 600
9	$y=\begin{cases} \lg(x+1) \\ \sin^2\sqrt{ax} \end{cases}$	$x > 1.2$ $x \le 1.2$	$a=20.3$	$x \in [0.5;2]$ $\Delta x = 0.005$	0, 140, 300
10	$y=\begin{cases} (\ln^3 x + x^2)/\sqrt{x+t} \\ \cos x + t\sin^2 x \end{cases}$	$x \le 0.9$ $x > 0.9$	$t=2.2$	$x \in [0.2;2]$ $\Delta x = 0.004$	0, 175, 450