

Лабораторна робота №4

Основи сучасних технологій розробки мовою Java

Система автоматизації збирання проектів Maven та модульного тестування JUnit

Назва системи **Maven** є словом з мови ідиш, сенс якого можна приблизно висловити як «збирач знання»

Зібрати на Java проект рівня «Hello, world!» можна і за допомогою командного рядка. Але чим складніше ПЗ, що розробляється, і чим більше воно використовує сторонніх бібліотек і ресурсів, тим складніше буде команда для збирання. Maven розроблений для полегшення цієї роботи.

Одна з головних особливостей системи - декларативний опис проекту. Це означає, що розробнику не потрібно приділяти увагу кожному аспекту збирання — всі необхідні параметри налаштовані за замовчуванням. Зміни потрібно вносити лише в тому обсязі, в якому програміст хоче відхилитися від стандартних налаштувань.

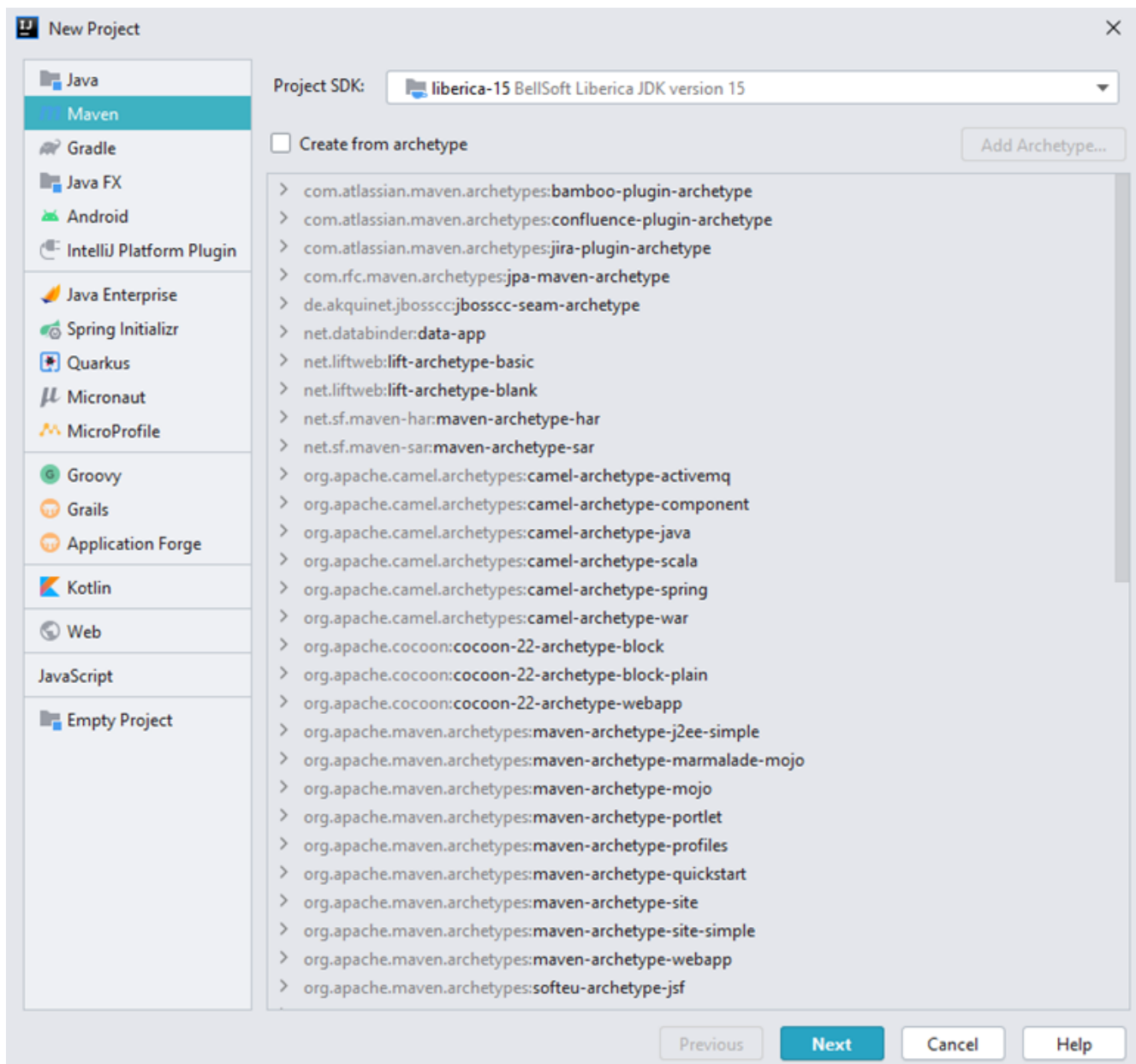
Ще одна перевага проекту — гнучке управління залежностями. Maven вміє довантажувати у свій локальний репозиторій сторонні бібліотеки, вибирати необхідну версію пакету, обробляти транзитивні залежності.

Розробники також підкреслюють незалежність системи від ОС. При роботі з командного рядка параметри залежать від платформи, але Maven дозволяє не звертати уваги на цей аспект.

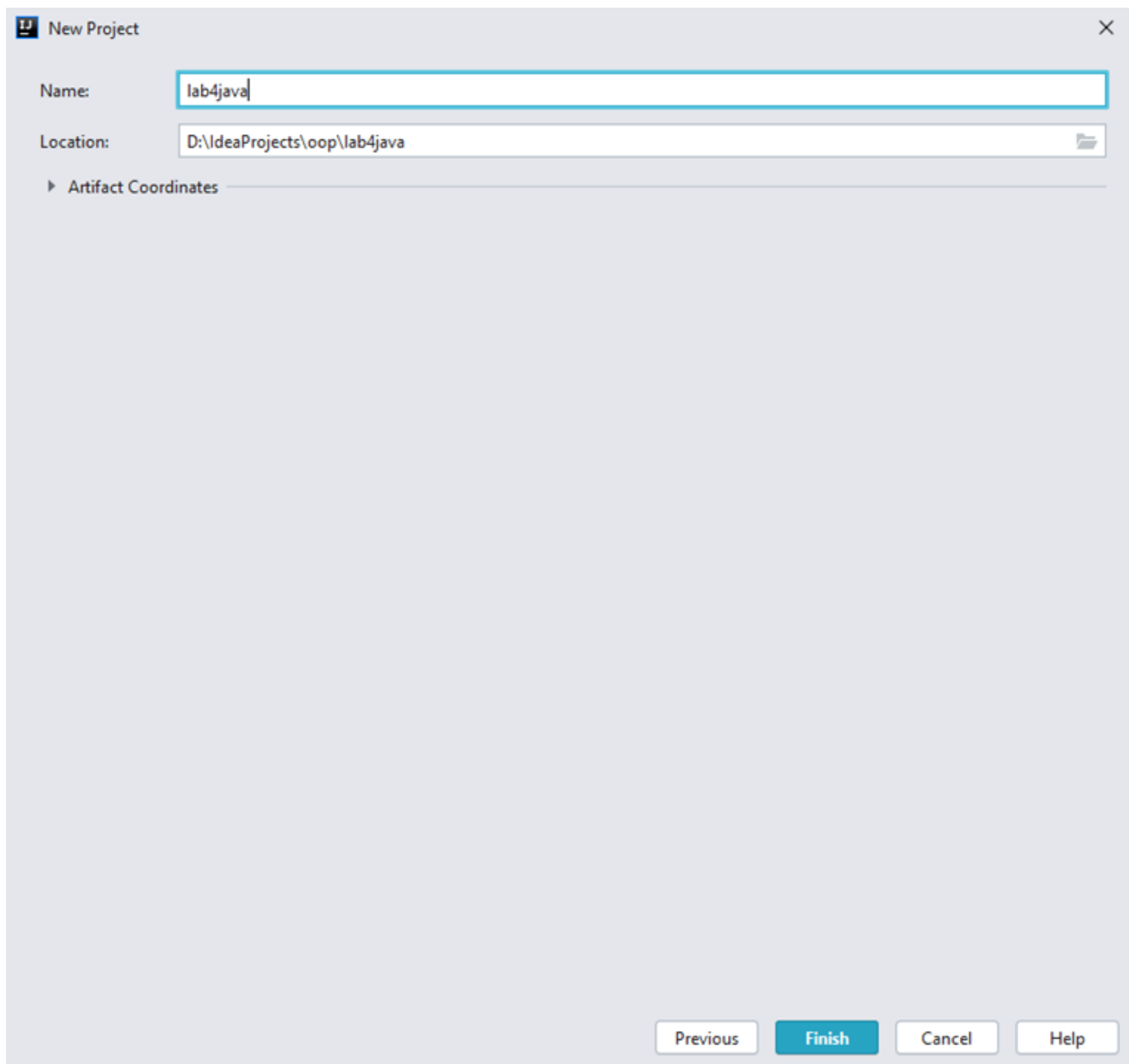
При необхідності систему збирання можна налаштувати під власні потреби, використовуючи готові плагіни та архетипи. А якщо нічого слушного не знайшлося — можна написати свої.

Створення проекту на основі Maven

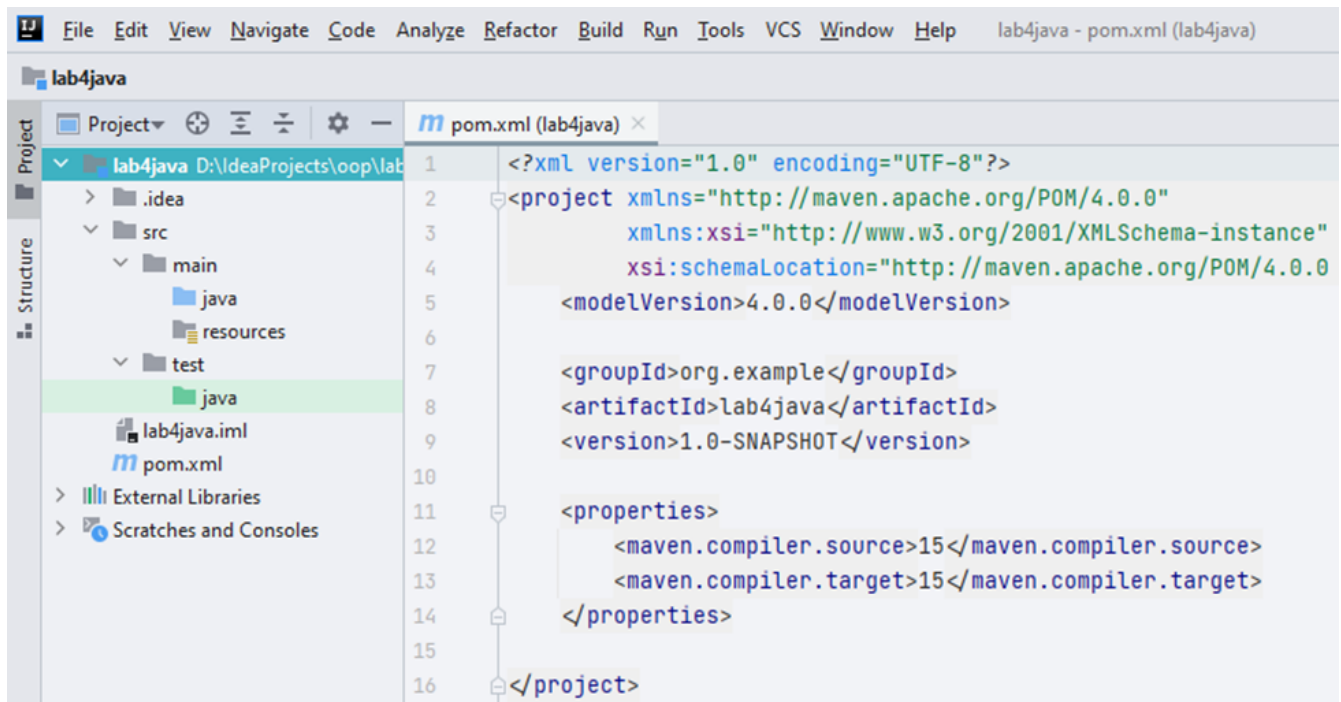
Створюємо в IntelliJ IDEA новий проект. У якості типу проекту обираємо **Maven**.



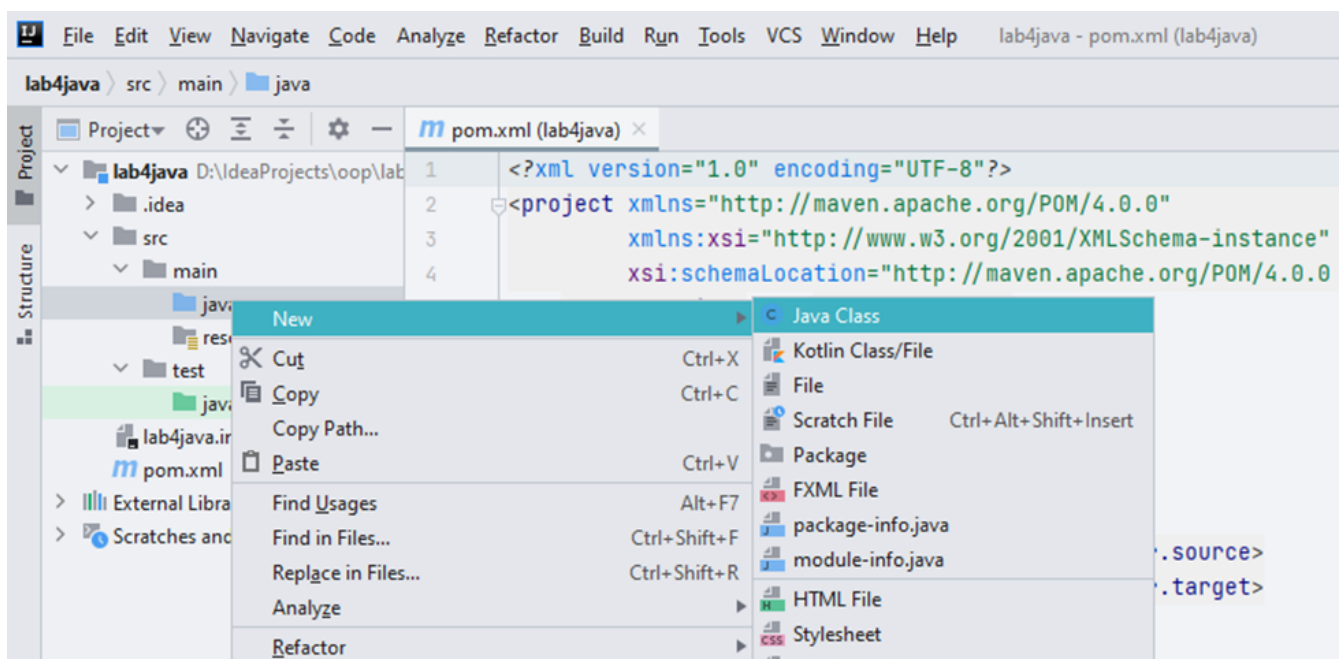
Після натискання на кнопку **Next** потрапляємо у звичне вікно вибору місця розміщення проекту:



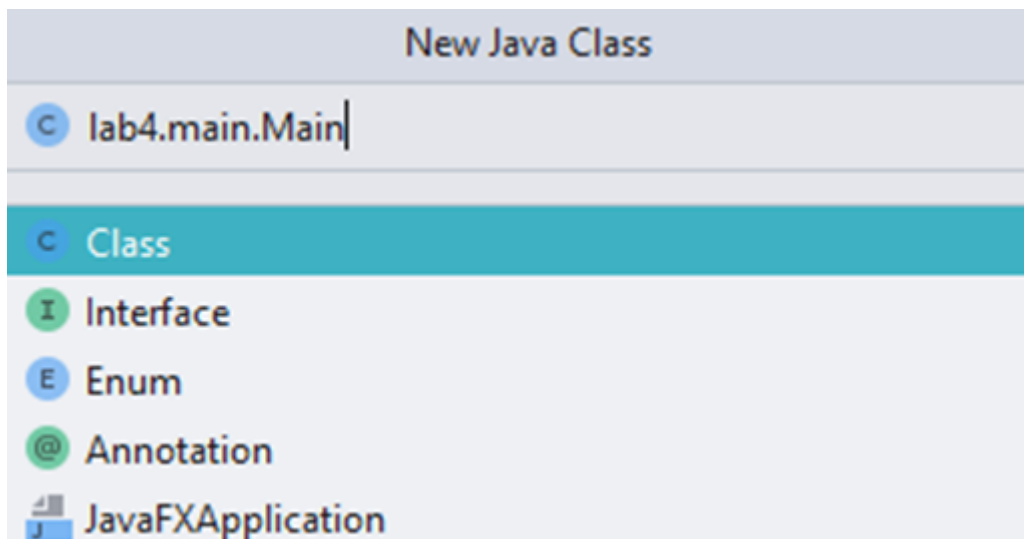
Після створення проекту відкриємо файл `pom.xml`, який містить опис початкової структури проекту.



Створимо головний клас програми. Для цього правою кнопкою миші (ПКМ) треба клацнути на вузлі java. Вибрати **New** → **Java Class**



та ввести ім'я класу. Також, можна вказати повне ім'я, включаючи всі вкладені пакети.



Тепер можна додавати будь-які методи, змінні та ін.

Але ми переходимо до використання фреймворку для модульного тестування JUnit

Використання JUnit

JUnit - це модульна система тестування для мови програмування Java. JUnit має важливе значення в розробці тестових розробок і є одним із сімейства модульних модулів тестування, який разом відомий як xUnit, що походить від SUnit.

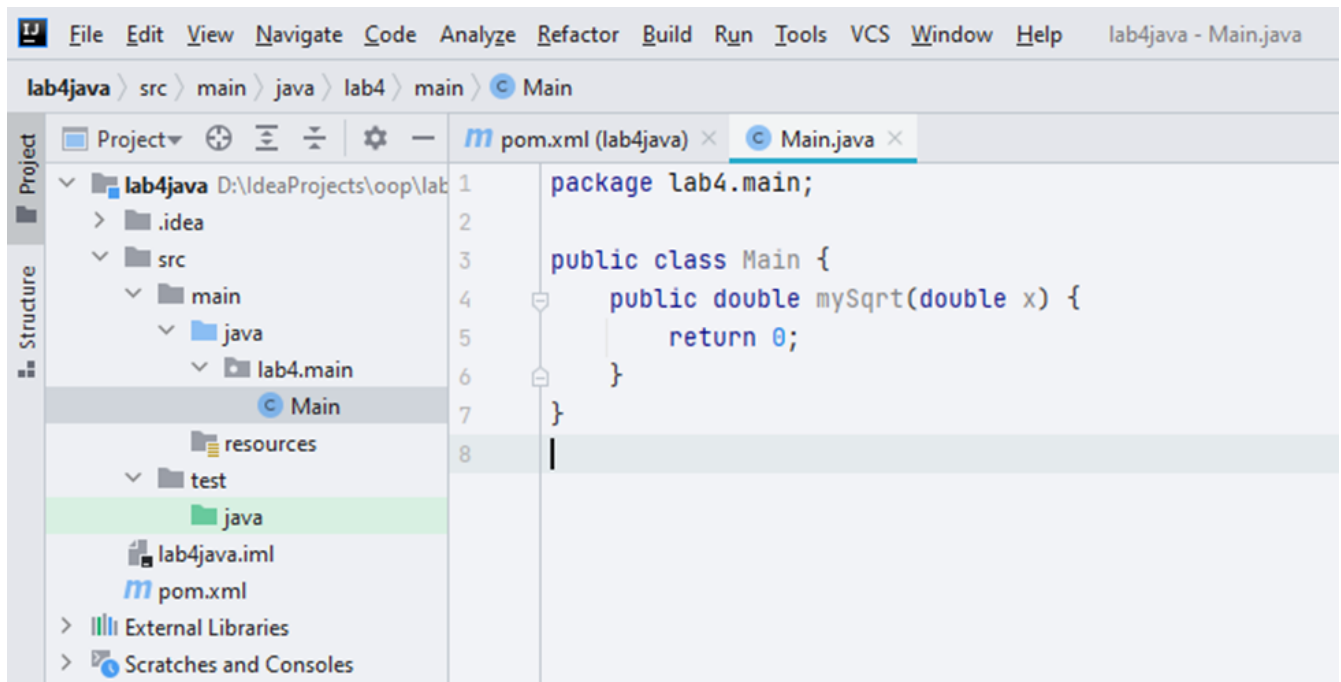
JUnit підключається як JAR під час компіляції. Остання версія фреймворку, JUnit 5, знаходиться в пакеті `org.junit.jupiter`. Попередні версії JUnit 4 та JUnit 3 знаходились у пакетах `org.junit` та `junit.framework`, відповідно.

Дослідження, проведене у 2013 р показало, що серед 10000 проектів Java, розміщених на GitHub, JUnit (у поєднанні з slf4j-api) є найбільш частою зовнішньою бібліотекою. Кожна бібліотека була використана у 30,7% проектів.

Методика використання модульного тестування із використанням JUnit лежить в основі технології TDD (Test Driven Development).

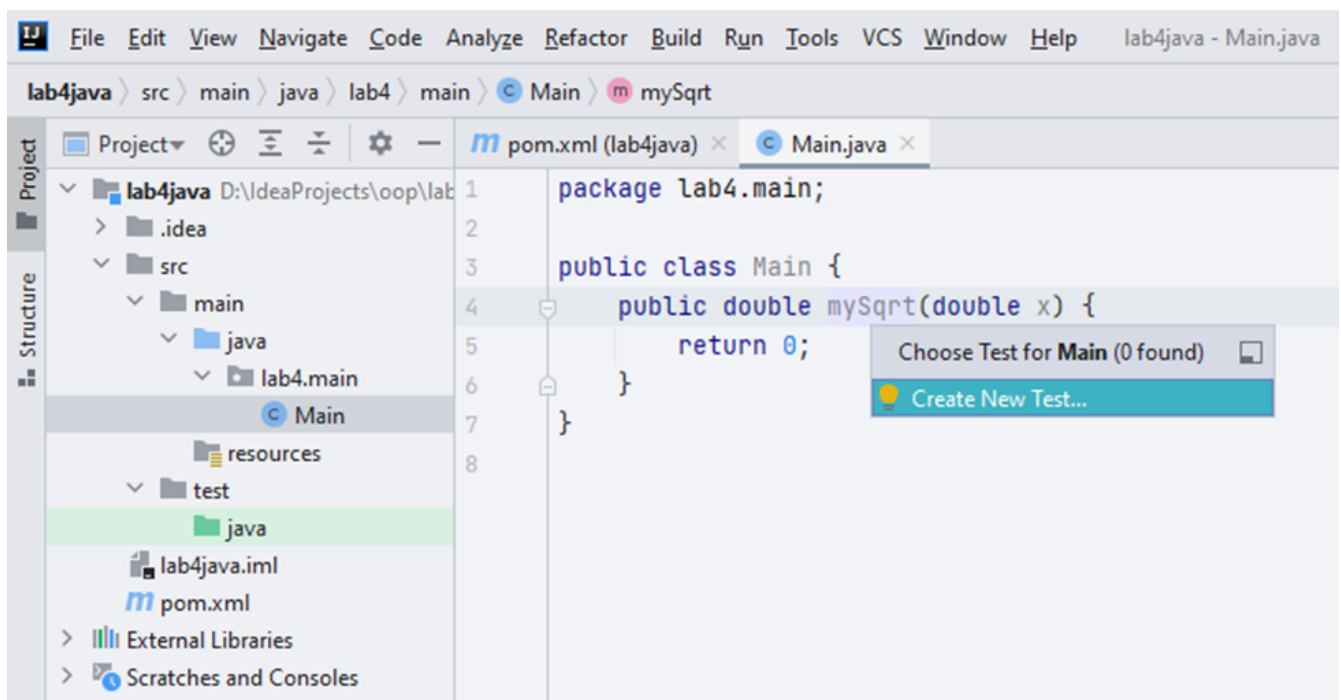
Тому, продовжимо створювати наш проєкт за допомогою TDD.

Додамо до раніше створеного класу опис нового метода. Та в операторі return цього метода, вкажемо значення "заглушку" - 0



Вибираємо в коді цей метод (встановлюємо курсор на нього). Натискаємо ПКМ та обираємо Goto → Test (або сполучення клавіш Ctrl+Shift+T)

Оскільки тесту ще немає, то з'являється меню, що пропонує його створити:



Відкриється вікно створення тесту. Якщо у ньому є повідомлення "JUnit5 library not found in the module" треба один раз натиснути кнопку Fix та відмітити пункти у вікні згідно з рисунком.

Create Test

Testing library: JUnit5

JUnit5 library not found in the module Fix

Class name: MainTest

Superclass: ...

Destination package: lab4.main ...

Generate: ☒ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

Member
<input checked="" type="checkbox"/> m g mySqrt(x:double):double

? OK Cancel

Буде створений тестовий клас MainTest, що містить текст подібний до наступного рисунка:

```
package lab4.main;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class MainTest {

    @BeforeEach
    void setUp() {
    }

    @Test
    void testMySqrt() {
    }

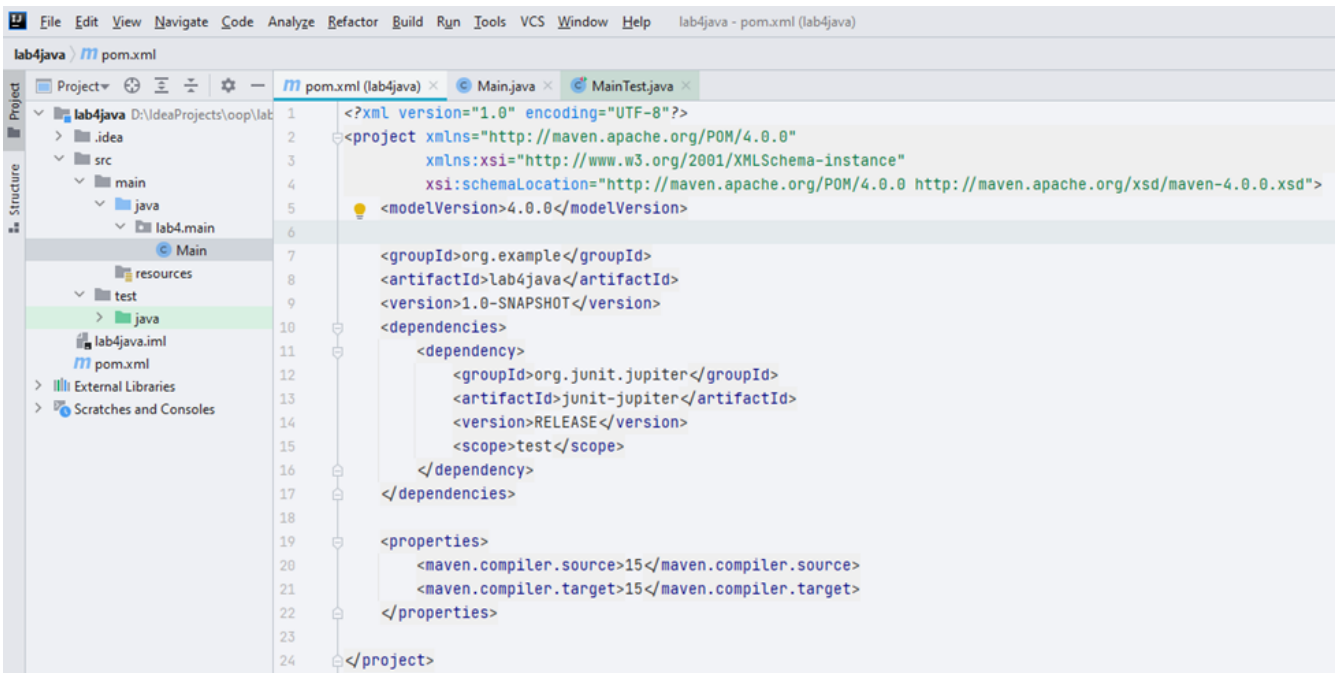
}
```

Можливо, текст у файлі буде трохи інакший, в такому випадку його можна виправити, щоб

він виглядав, як на рисунку вище.

```
@org.junit.jupiter.api.Test
void mySqrt() {
}
}
```

Переглянувши файл pom.xml, можна побачити, що в ньому з'явився розділ dependencies, де підключена одна "залежність" (термін Maven): **junit-jupiter**



Прив'яжемо тестовий клас до основного, для цього додамо опис основного класу:

```
Main main;
```

У методі setUp треба виконати початкову ініціалізацію об'єкта main (створюємо новий об'єкт). Цей метод буде викликатися перед виконанням кожного тестового методу.

```
main = new Main();
```



```

package lab4.main;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class MainTest {

    Main main;

    @BeforeEach
    void setUp() {
        main = new Main();
    }

    @Test
    void testMySqrt() {
        double x = 2.0;
        double expected = 1.4142;
        double result = main.mySqrt(x);
        assertEquals(expected, result, delta: 1e-4);
    }
}

```

Розробимо логіку тестування:

- вкажемо значення аргументу функції `double x = 2.0;`
- вкажемо, значення функції, що очікується як результат `double expected = 1.4142;`
- викликаємо функцію, що тестується, та зберігаємо її результат `double result = main.mySqrt(x);`
- порівнюємо результат з очікуваним. Для порівняння дробних чисел треба вказувати припустиму похибку порівняння. Оскільки у нас очікуване значення вказане з точністю 4 знаки, то вкажемо похибку 10^{-4} `assertEquals(expected, result, 1e-4);`

Одразу ж виконаємо тестування, та пересвідчимося, що тест "не проходить"

```

17      @Test
18      void testMySqrt() {
19          double x = 2.0;
20          double expected = 1.4142;
21          double result = main.mySqrt(x);
22          assertEquals(expected, result, delta: 1e-4);
23      }
24  }

```

"C:\Program Files\BellSoft\LibericaJDK-15-Full\bin\java.exe" ...

org.opentest4j.AssertionFailedError:

Expected :1.4142

Actual :0.0

[<Click to see difference>](#)

<5 internal calls>

at lab4.main.MainTest.testMySqrt(MainTest.java:22) <31 internal calls>

at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal calls>

at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <23 internal calls>

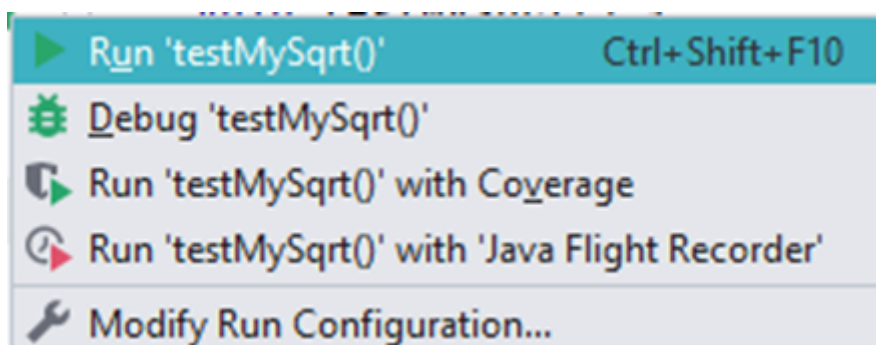
виправимо нашу функцію так, щоб вона правильно обчислювала значення згідно завдання:

```

public double mySqrt(double x) {
    return Math.sqrt(x);
}

```

Знову виконуємо тест



Бачимо, що на цей раз все добре:



Аналогічно додаємо та тестуємо інші функції. Після успішного проходження всіх тестів створюємо методи, що викликають вже протестовані та, таким чином реалізують програмну логіку.

Управляючі структури мови Java

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 зі службового слова **case**. Кожній мітці **case** ставиться у відповідність константа. Якщо значення виразу збігається із значенням деякої мітки, управління буде передано першій інструкції, що йде після цієї мітки. Якщо збігів 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 елемента> <формальна змінна> : <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 знайдено 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 ж **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 ставитися 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**.

- 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 аргументу	Номери елементів, для тестування
1	$y = \begin{cases} ax^2 \ln x \\ 1 \\ e^{ax} \cos bx \end{cases}$	$0.7 < x \leq 1.4$ $x \leq 0.7$ $x > 1.4$	$a = -0.5$ $b = 2$	$x \in [0; 3]$ $\Delta x = 0.004$	175, 350, 750
2	$y = \begin{cases} \pi x^2 - 7/x^2 \\ ax^3 + 7\sqrt{x} \\ \lg(x + 7\sqrt{x}) \end{cases}$	$x < 1.7$ $x = 1.7$ $x > 1.7$	$a = 1.5$	$x \in [0.8; 2]$ $\Delta x = 0.005$	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	$y = \begin{cases} \pi x^2 - 7/x^2 \\ ax^3 + 7\sqrt{x} \\ \ln(x + 7\sqrt{ x + a }) \end{cases}$	$x < 1.3$ $x = 1.3$ $x > 1.3$	$a = 1.65$	$x \in [0.7; 2]$ $\Delta x = 0.005$	0, 120, 260
5	$y = \begin{cases} 1.5a \cos^2 x \\ (x - 2)^2 + 6a \\ 3a \cdot \operatorname{tg} x \end{cases}$	$x \leq 0.3$ $0.3 < x \leq 2.3$ $x > 2.3$	$a = 2.3$	$x \in [0.2; 2.8]$ $\Delta x = 0.002$	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	$y = \begin{cases} bx - \operatorname{tg} bx \\ bx + \lg bx \end{cases}$	$bx \leq 0.45$ $bx > 0.45$	$b = 1.5$	$x \in [0.1; 1]$ $\Delta x = 0.001$	0, 200, 900
8	$y = \begin{cases} \sin x \lg x \\ \cos^2 x \end{cases}$	$x > 3.4$ $x \leq 3.4$		$x \in [2; 5]$ $\Delta x = 0.005$	0, 280, 600
9	$y = \begin{cases} \lg(x + 1) \\ \sin^2 \sqrt{ax} \end{cases}$	$x > 1.2$ $x \leq 1.2$	$a = 20.3$	$x \in [0.5; 2]$ $\Delta x = 0.005$	0, 140, 300
10	$y = \begin{cases} (\ln^3 x + x^2) / \sqrt{x + t} \\ \cos x + t \sin^2 x \end{cases}$	$x \leq 0.9$ $x > 0.9$	$t = 2.2$	$x \in [0.2; 2]$ $\Delta x = 0.004$	0, 175, 450