

Лабораторна робота №2

Основні алгоритмічні структури мови Java

1. У середовищі IntelliJ IDEA або NetBeans створити новий проект. Додати до цього проекту новий клас.
2. У створеному класі описати метод, що обчислює значення функції, яка задана у таблиці.

Створити тестовий клас для тестування головного класу програми. Додати до нього методи тестування метода, який був створений у п.2. Виконати тестування цього метода.

3. Розробити метод, що за вказаними значеннями кроку, початку та кінця інтервалу обчислює кількість кроків для табулювання.

Створити тестові методи для нього і виконати тестування.

4. 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.3). *Масиви повинні бути описані як **private** і для них потрібно створити методи доступу до їхніх елементів за номерами.*

Створити тестові методи для них і виконати тестування.

5. 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.
6. Створити тестові методи для методів з п.5 і виконати тестування
7. Вивести найбільший та найменший елементи масиву значень функції, вказавши їхні номери і відповідні значення аргументу.
8. Додати до створеного класу метод main, перетворивши, таким чином, його на автономну програму. Скомпілювати і виконати програму

Короткі теоретичні відомості

Типи масиву

Типи масиву використовуються для визначення масивів – упорядкованих наборів однотипних змінних. Ви можете визначити масив над будь-яким існуючим у мові типом, включаючи типи, визначені користувачем. Крім того, можна користатися масивами масивів чи багатовимірними масивами. Коротко говорячи, якщо ми можемо створити змінну деякого типу, виходить, ми можемо створити і масив змінних цього типу. Разом з тим, створення масивів у мові Java може показатися вам незвичним, тому що воно вимагає застосування оператора **new**.

Приклад 1. Описання масивів і виділення пам'яті для масивів

```
int[] myIntArray;           // описання масиву цілих чисел
myIntArray = new int[8];
    // створення масиву з 8 цілих чисел
MyType[] myObjectArray;
    // описання масиву об'єктів типу MyType
myObjectArray = new MyType[5];
    // створення масиву з 5 елементів типу MyType
```

Оператор **new** дає команду оболонці часу виконання виділити необхідну кількість пам'яті під масив. Як видно з цього прикладу, не треба повідомляти розмір масиву тоді ж, коли ви створюєте змінну-масив. Після того, як створили масив оператором **new**, доступ до цього масиву здійснюється точно так само, як у мовах C чи Pascal.

Приклад 2. Присвоювання значень елементам масивів

```
myIntArray[0] = 0;
myIntArray[1] = 1;
myIntArray[2] = 2;
myObjectArray[0] = new MyType();
myObjectArray[1] = new MyType();
myObjectArray[2] = new MyType();
myObjectArray[0].myDataMember = 0;
myObjectArray[1].myDataMember = 1;
myObjectArray[2].myDataMember = 2;
```

Масиви в мові Java мають три важливих переваги перед масивами в інших мовах. По-перше, програмісту не треба вказувати розмір масиву при його оголошенні. По-друге, будь-який масив у мові Java є змінною - а це значить, що його можна передати як параметр методу і використовувати як значення, що повертається методом. І по-третє, завжди легко довідатися, який розмір даного масиву в будь-який момент часу. Наприклад, так визначається розмір масиву, що був оголошений вище.

Приклад 3. Отримання довжини масиву

```
int len = myIntArray.length;  
System.out.println("Length of myIntArray=" + len);
```

Багатовимірні масиви у мові Java визначаються, як "масиви, елементами яких є масиви". Тобто двовимірний масив – це масив, елементами якого є лінійні масиви. Наприклад, так відбувається робота з двовимірним масивом.

Приклад 4. Описання та робота з двовимірним масивом

```
double[][] m;  
m = new double[3][4];  
// масив з трьох рядків, у кожному по 4 елементи  
m[1][3] = 5.4;  
// присвоювання значення елементу,  
// що знаходиться у першому рядку під номером 3  
  
double[][] z;  
z = new double[3][]; // массив з трьох рядків  
z[0] = new double[1]; // у першому рядку – один елемент  
z[1] = new double[2]; // у другому рядку – два  
z[2] = new double[3]; // у третьому – три  
  
z[2][2] = 1.5; // припустиме присвоювання  
z[0][1] = 5.3; // ПОМИЛКА! У першому рядку є лише один елемент  
// із індексом 0
```

Управляючі структури мови Java

У мові Java існують такі управляючі структури:

1. *Оголошення, вирази та інструкції (оператори).*

Як інструкції, що завершуються символом "крапка с комою", можуть бути використані вирази таких категорій:

- a. вирази присвоювання, в яких використовується базовий (=) та складений оператори присвоювання виду `оп=`;
- b. префіксні та постфіксні форми виразів з операторами інкременту (`++`) та декременту (`--`);
- c. конструкції виклику методів;
- d. вирази створення об'єктів за допомогою оператора **new**.

Оголошення використовуються для описання (локальних) змінних та ініціалізації їх початковими значеннями.

2. *Блоки.*

Фігурні дужки, { та }, використовуються для поєднання декількох виразів у блок (він може бути і порожнім). Блок дозволяється використовувати в будь-якому місці коду, де передбачено застосування інструкції, оскільки блок, як такий, є складеною інструкцією.

3. *if ... else .*

Найбільш поширеною формою управляючих структур, що використовуються для зміни порядку обчислень в залежності від значення логічного виразу, є конструкція **if ... else**, синтаксис якої виглядає таким чином:

```
if (ЛогічнийВираз) Інструкція1  
  
else Інструкція2
```

Спочатку виконується перевірка значення логічного виразу. Якщо результат дорівнює `true`, виконується Інструкція1, інакше (і при наявності необов'язкового елементу **else**) – Інструкція2. Одна конструкція **if ... else** може бути вкладена всередину елементу **else** іншої з метою перевірки послідовності логічних умов. Також така конструкція може бути вкладена замість Інструкції1, але тоді треба пам'ятати, що елемент **else** завжди буде відноситися до найближчого **if**, яке не "закрито" своїм **else**.

4. *switch*.

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

```
switch (ЦілочисельнийВираз) {  
    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 ціла константа. Якщо значення цілочисельного виразу співпадає з зі значенням деякої мітки, управління буде передано першій інструкції, що йде після цієї мітки. Якщо співпадань 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**.

5. *while* та *do ... while*.

Синтаксис циклічної конструкції **while** виглядає так:

```
while (ЛогічнийВираз)
    Інструкція
```

Спочатку виконується перевірка значення логічного виразу. Якщо результат дорівнює **true**, виконується Інструкція (як інструкція може бути використаний блок), після чого логічний вираз перевіряється знов і процес повторюється до тих пір, поки в результаті перевірки не буде отримано значення **false**, – в цьому випадку управління буде передане першій інструкції коду, що іде за межами конструкції **while**.

Іноді виникає необхідність забезпечити циклічне виконання блоку інструкцій якнайменше один раз. З цією метою використовується конструкція **do ... while**, синтаксис якої може бути описаний таким чином:

```
do
    Інструкція
while (ЛогічнийВираз)
```

В цьому випадку перевірка істинності логічного виразу виконується після виконання тіла циклу. Цикл повторюється до тих пір, поки в результаті перевірки виразу не буде отримане значення **false**. В якості тіла циклу **do ... while** найчастіше використовується блок інструкцій.

6. *for*.

Вираз **for** застосовується для організації циклічного переходу по значеннях із вказаного діапазону і в загальному випадку виглядає таким чином:

```
for ( СекціяІніціалізації;
      ЛогічнийВираз;
      СекціяЗмін) Інструкція
```

Секція ініціалізації дозволяє ініціалізувати (можливо, з попереднім об'явленням) змінні циклу і виконується лише один раз. Логічний вираз піддається перевірці, та її результат, якщо він дорівнює **true**, дає підставу для виконання тіла циклу, після чого обчислюються вирази секції змін, і

логічний вираз перевіряється у черговий раз. Цикл повторюється доти, поки результатом перевірки не стане значення false. Кожна з двох частин конструкції **for** – СекціяІніціалізації та СекціяЗмін – може бути представлена списком виразів, відокремлених символом ”,”(кома). Вирази у подібних списках обчислюються зліва направо.

Усі секції заголовка циклу **for** є необов’язковими. Якщо СекціяІніціалізації опускається, на її місці залишається лише символ крапки з комою. Якщо ж із заголовка виключається ЛогічнийВираз, то в якості логічної умови мається на увазі літерал true. Виключення усіх трьох секцій призводить до того, що цикл стає ”нескінченим”:

```
for ( ; ; )  
    Інструкція
```

Вихід з такого циклу повинен бути забезпечений за допомогою інших засобів, таких як команда break (її ми розглянемо далі) або інструкції викидання виключення.

Цикли for, у відповідності до прийнятої угоди, застосовуються тільки в тих випадках, коли необхідно забезпечити ”проходження” по значеннях із визначених діапазонів. Якщо виразі ініціалізації та зміни змінних застосовуються не за призначенням і не пов’язані з логічною умовою циклу, подібні дії можна кваліфікувати не інакше як порушення угоди і ознака порочного стилю програмування.

7. *for (each)* .

Починаючи з версії Java5 у мові Java з’явилась нова конструкція, призначена для виконання ітерації по масиву або колекції. Вона виглядає так:

```
for (<тип елемента> <формальне ім’я> : <масив, або колекція>)  
    Інструкція
```

8. 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).

9. *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 2.4. 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.

10. 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.

11. return. Інструкція **return** завершує виконання методу і передає управління до коду-ініціатору. Якщо метод не повертає значень, інструкція виглядає просто: **return**. Якщо ж в оголошенні методу вказано тип параметра, що повертається, до складу команди **return** повинен бути включеним вираз, який може бути присвоєний змінній оголошеного типу. Наприклад, якщо метод повертає значення типу

double, в інструкції **return** дозволяється використовувати вирази, що відносяться до типів **double**, **float** або будь-якого цілих типів.

12. **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**.
13. 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.

Модульне тестування з використанням JUnit у середовищі NetBeans

Розглянемо, як створюються та виконуються тести на прикладі програми, аналогічної до тієї, що була складена у лабораторній роботі №1. Проте, на практиці, для того, щоб виконувати наступні кроки треба створити головний клас програми і додати до нього хоча б один метод.

Нехай у нас є проект *Example1*, що складається з пакету **example1** в якому міститься єдиний клас Main (у файлі Main.java).

```
package example1;

/**
 * @author Eugeny
 */
public class Main {

    /**
     * Метод, що обчислює значення у за формулою з завдання
     * @param a перший параметр
     * @param b другий параметр
     * @param x третій параметр
     * @return обчислене значення у
     */
    public double calcY(double a, double b, double x) {
        return (Math.pow(a, 2 * x) + Math.pow(b, -x) *
            Math.cos(a + b) * x) / (x + 1);
    }

    /**
     * Метод, що обчислює значення r за формулою з завдання
     * @param a перший параметр
     * @param b другий параметр
     * @param x третій параметр
     * @return обчислене значення r
     */
    public double calcR(double a, double b, double x) {
        return Math.sqrt(x * x + b - b * b *
            Math.pow(Math.sin(x + a), 3) / x);
    }

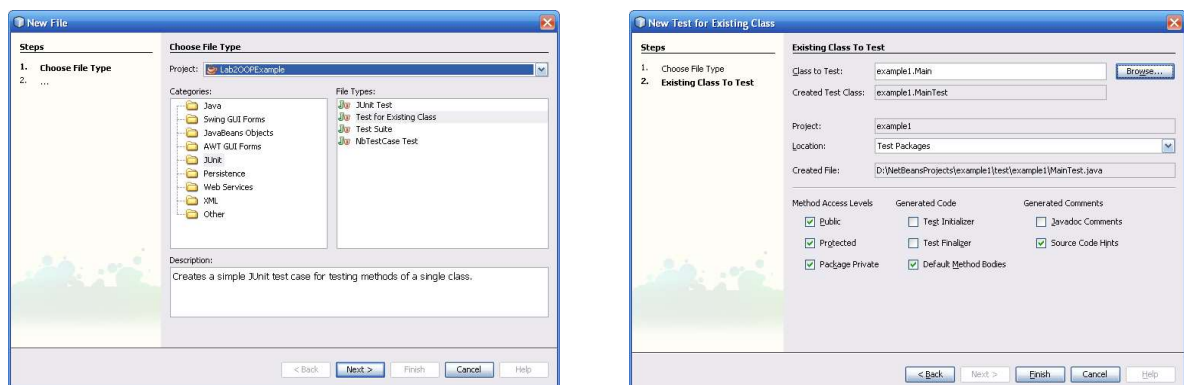
    /**
     * метод присвоює a,b,x значення з завдання
     * та обчислює значення змінних r та y - calcR() і calcY()
     * Після того виводяться початкові та обчислені значення
     */
}
```

```

public void printResults() {
    double a = 0.3;
    double b = 0.9;
    double x = 0.61;
    double r = calcY(a, b, x);
    double y = calcR(a, b, x);
    System.out.printf("x=%5.3f y=%5.3f z=%5.3f\n", a, b, x);
    System.out.printf("y=%8.4f\n", y);
    System.out.printf("r=%8.4f\n", r);
}
/**
 * @param args аргументи командного рядка
 */
public static void main(String[] args) {
    Main program = new Main();
    program.printResults();
}
}

```

Для створення класу модульного тестування у середовищі NetBeans оберіть пункт New File... у меню File. У вікні, що відкриється, обрати категорію **JUnit** і в цій категорії тип файлу Test for Existing Class. Для продовження треба натиснути кнопку "Next >". Відкриється наступне вікно.



Укажіть в ньому, для якого класу створюються тестовий клас (**Class to Test:**). Для обрання потрібного класу натисніть кнопку **Browse...** та оберіть потрібний клас. В нашому випадку це клас *example1.Main*. Після налаштування створюваного тестового класу треба натиснути "Finish". Якщо Ви робите це перший раз, то буде запропоновано вибрати версію бібліотеки модульного тестування JUnit 3.x або JUnit 4.x. Оберемо JUnit 4.x.

Буде створено клас (у нашому випадку це буде клас `MainTest`), який буде розміщено у розділі `Test Packages`.

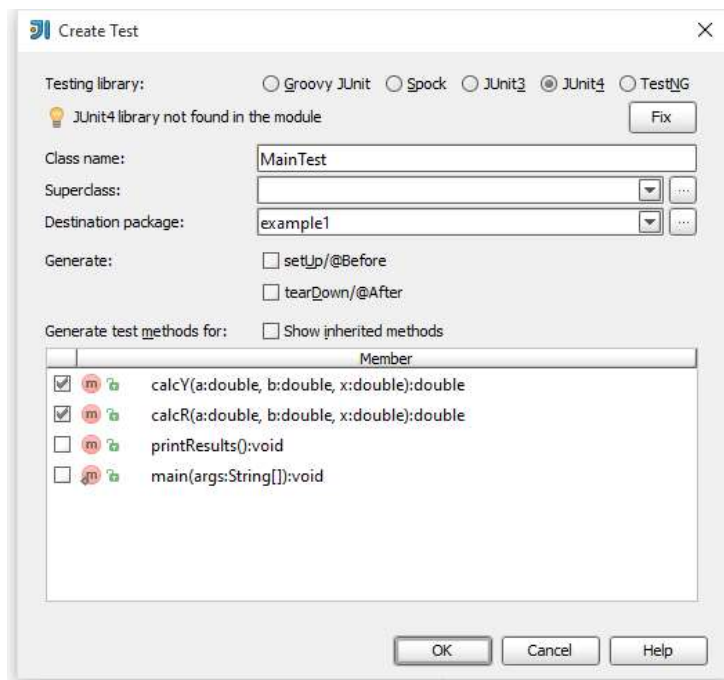
При створенні тестового класу, середовище `NetBeans` формує шаблони тестових методів, їх треба відредагувати таким чином:

- Метод для тестування метода `main()` (він буде називатися `testMain()`) можна (або навіть, треба) видалити.
- Оскільки метод `printResults()` лише викликає інші методи і виводить результати їх виконання, то його можна не тестувати (тобто метод `testPrintResults()` теж можна видалити).
- На початку тестового класу додайте описання константи `EPS` – точності для порівнянь.
- Методи `testCalcY()` та `testCalcR()` треба змінити так, щоб за їх допомогою можна було тестувати відповідні методи головного класу програми. Спочатку треба видалити останні два рядки кожного метода – вони були додані туди тільки для того, щоб пересвідчитися, що тести насправді писалися людиною, а не були використані шаблони.
- У кожному з методів впишіть початкові значення, що треба присвоїти змінним, які передаються цим методам (в нашому випадку – змінні `a`, `b`, `x`). Ці значення треба підібрати таким чином, щоб можна було протестувати різні випадки в процесі обчислення функцій.
- Змінній **`expResult`** треба присвоїти значення, що очікується у результаті виконання метода, що буде тестуватися (його треба обчислити вручну, або за допомогою калькулятора).
- Після виклику метода, що тестується, змінній **`result`** буде присвоєне значення, яке обчислить метод.

- Значення змінних **expResult** та **result** треба порівняти за допомогою методу `assertEquals(arg1,arg2,eps)`, де `arg1`, `arg2` – значення, що порівнюються, `eps` – похибка порівняння. Цей метод перевіряє, чи виконується нерівність $|arg1 - arg2| < eps$, тобто якщо `eps` – мале число, то можна вважати, що `arg1` та `arg2` приблизно рівні.
- Для кожного з методів класу, що тестується, можна (і треба) створити декілька тестових методів (якщо ж в методі, який тестується, є оператори розгалуження, то потрібно створити стільки тестів, скільки є різних шляхів виконання метода, а крім того, **ОБОВ'ЯЗКОВО ТРЕБА** перевірити умови на межі переходу в умовному операторі).
- Виконати тести можна обравши в головному меню NetBeans у пункті Run підпункт Test "<Ім'я проекту>" (в нашому прикладі – Test "example1").
- Після виконання тесту у вікні тестування будуть показані результати: **passed** (зеленим кольором) означає, що відповідний тест "пройшов", **FAILED** (червоним кольором) означає, що тест "не пройшов" і треба перевірити, чому так сталося, та знайти помилку. Якщо хоча б один тест не пройшов, то вважається, що проект в цілому не пройшов тестування і не може здаватися, як завершений продукт. Треба виправляти помилки до тих пір, поки ВСІ тести будуть "проходити".
- Відмітимо, що тестовий клас найчастіше виходить більшим, ніж той клас, який він тестує, проте тестовий клас складається з простих методів і тому його розроблювати нескладно.

Модульне тестування з використанням JUnit у середовищі IntelliJ IDEA

Для створення класу модульного тестування у середовищі IntelliJ IDEA треба спочатку створити каталог для розміщення тестових класів (ПКМ на проєкті, 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 JUnit4 та відмітити методи, які потрібно тестувати та натиснути кнопку Fix (щоб додати бібліотеку тестування JUnit 4 до поточного модуля проєкту). Натиснути Ok для завершення створення тестового класу.



На початку тестового класу додайте описання константи EPS – точності для порівнянь.

Методи testCalcY() та testCalcR() треба змінити так, щоб за їх допомогою можна було тестувати відповідні методи головного класу програми.

У кожному з методів впишіть початкові значення, що треба присвоїти змінним, які передаються цим методам (в нашому випадку – змінні a, b, x). Ці значення треба підібрати таким чином, щоб можна було протестувати різні випадки в процесі обчислення функцій.

Змінній `expResult` треба присвоїти значення, що очікується у результаті виконання метода, що буде тестуватися (його треба обчислити вручну, або за допомогою калькулятора).

Після виклику метода, що тестується, змінній `result` буде присвоєне значення, яке обчислить метод.

Значення змінних `expResult` та `result` треба порівняти за допомогою методу `assertEquals(arg1,arg2,eps)`, де `arg1`, `arg2` – значення, що порівнюються, `eps` – похибка порівняння. Цей метод перевіряє, чи виконується нерівність $|arg1 - arg2| < eps$, тобто якщо `eps` – мале число, то можна вважати, що `arg1` та `arg2` приблизно рівні.

Для кожного з методів класу, що тестується, можна (і треба) створити декілька тестових методів (якщо ж в методі, який тестується, є оператори розгалуження, то потрібно створити стільки тестів, скільки є різних шляхів виконання метода, а крім того, **ОБОВ'ЯЗКОВО ТРЕБА** перевірити умови на межі переходу в умовному операторі)

Виконати тест можна вибравши у дереві проекту створений клас `MainTest`. Натиснути ПКМ та вибрати `Run MainTest`.

Приклад тестового класу:

```
package example1;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Eugeny
 */
public class MainTest {

    public static final double EPS = 1e-6;
    // припустима точність порівнянь
```

```

@Test // Тест загального випадку, очікуємо результат 1.0
public void testCalcY() {
    System.out.println("calcY (1.0, -1.0, 0.0)");
    double a = 1.0;
    double b = -1.0;
    double x = 0.0;
    Main instance = new Main();
    double expResult = 1.0;
    double result = instance.calcY(a, b, x);
    assertEquals(expResult, result, EPS);
}

@Test // Тест окремого випадку, очікуємо результат "+INFINITY"
public void testCalcY0() {
    System.out.println("calcY (1.0, -1.0, 0.0)");
    double a = 1.0;
    double b = -1.0;
    double x = -1.0;
    Main instance = new Main();
    double expResult = Double.POSITIVE_INFINITY;
    double result = instance.calcY(a, b, x);
    assertEquals(expResult, result, EPS);
}

@Test // Тест загального випадку, очікуємо результат 2.0
public void testCalcR() {
    System.out.println("calcR (0.0, 0.0, 2.0)");
    double a = 0.0;
    double b = 0.0;
    double x = 2.0;
    Main instance = new Main();
    double expResult = 2.0;
    double result = instance.calcR(a, b, x);
    assertEquals(expResult, result, EPS);
}

@Test // Тест окремого випадку, очікуємо результат "NaN", 0/0
public void testCalcR0() {
    System.out.println("calcR (0.0, 0.0, 0.0)");
    double a = 0.0;
    double b = 0.0;
    double x = 0.0;
    Main instance = new Main();
    double expResult = Double.NaN;
    double result = instance.calcR(a, b, x);
    assertEquals(expResult, result, EPS);
}
}

```

Варіанти завдань

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