Лабораторна робота №2 - Функції

Функцією в програмуванні називається закінчена ділянка програми, що вирішує певне завдання - зазвичай це частина великого завдання, яке вирішує програма в цілому, хоча в простих випадках написання програми зводиться до написання однієї функції. Як і у функції в математичному сенсі, у функції в програмуванні є входи (параметри), вихід (результат) і визначення, яке вказує, як розраховується значення виходу за заданим значенням входів.

Визначення простих функцій на Котліні мало відрізняється від визначення математичних функцій. Розглянемо для прикладу математичну функцію `sqr(x) = x`^{2}. На Котліні вона буде записана так:

```
fun sqr(x: Int) = x * x
// або
fun sqr(x: Double) = x * x
```

В цьому визначенні **`fun`** - це *ключове слово*, з якого починається визначення будь-якої функції в Котліні; **`fun`** є скороченням від function - функція. `sqr` - це *ім'я* функції, x - *параметр* функції, `= x * x` - *тіло* функції, яке визначає, як треба обчислити її *результат*. Им'я функції разом з її параметрами та ключовим словом **`fun`** називається її *заголовком*.
Оскільки *операція* обчислення квадрату числа в Котліні відсутня, результат обчислюється як добуток `x*x`.

Математичні функції

У бібліотеці Котліна визначена велика кількість математичних *функцій*, які призначені для виконання більш складних операцій. Для прикладу їхнього використання розглянемо задачу розв'язання квадратного рівняння `ax`^{2} `+ bx + c = 0`.

Нагадаємо, що корені квадратного рівняння шукаються за формулою `x`$_{1,2}$ `= (-b ± √d) / (2a)`, де d — дискримінант квадратного рівняння — обчислюється як `d = b`^{2}`-4ac`. Ми розв'яжемо цю задачу в спрощеному вигляді — знайти будь-який з двох можливих коренів, скажімо, той, в якому в чисельнику використовується знак плюс.

Для початку напишемо функцію для розрахунку дискримінанту (вона ще знадобиться нам у майбутньому). Для розрахунку b^2 використаємо **вже написану** вище функцію sqr(x: Double).

```
fun discriminant(a: Double, b: Double, c: Double) = sqr(b) - 4 * a * c
```

В цьому фрагменті b є *аргументом* функції `sqr`. Запис виду `sqr(b)` називається *викликом* функції `sqr`. Підкреслимо відмінність *параметру* та *аргументу* — параметр визначається **всередині** функції та має визначене ім'я, в даному випадку `x`, а аргумент передається в функцію **ззовні** та може бути як іменем змінної, так і більш складним *виразом*.
Тепер напишемо функцію для пошуку корня квадратного рівняння. Для обчислення квадратного кореня застосуємо готовую математичну функцію `sqrt(x: Double)` з математичної бібліотеки Котліна.

```
fun quadraticEquationRoot(a: Double, b: Double, c: Double) =
        (-b + sqrt(discriminant(a, b, c))) / (2 * a)
```

Тут ми, в свою чергу, використовуємо **вже написану** функцію `discriminant` для пошуку дискримінанту, і вираз `discriminant(a, b, c)`, тобто дискримінант рівняння, є *аргументом* функції `sqrt`. Це саме той випадок, коли аргумент є складним *виразом*.

Змінні в функціях

Вище ми розглянули приклади з функціями `sqr` та `discriminant` обчислення результату в яких займало один рядок коду. Проте, в програмуванні це скоріш рідкий випадок; частіше розрахунок результату функції передбачає реалізацію деякої послідовності обчислень — алгоритму. Для збереження результатів **проміжних** обчислень програмісти вигадали *змінні*.
Розглянемо, наприклад, задачу обчислення **добутку** двох коренів квадратного рівняння. Нагадаємо, що корені квадратного рівняння обчислюються як `(-b+√d)/(2a)` та `(-b-√d)/(2a)` відповідно, де `d` - дискримінант квадратного рівняння. При обчисленні добутку зручно спочатку зберегти обчислений корінь з дискримінанту у змінній `sd`, через те, що він використовується при обчисленні обох коренів. Після того треба обчислити обидва кореня `x1` та `x2` і вже потім розрахувати їхній добуток. На Котліні це записується таким чином:

```
fun quadraticRootProduct(a: Double, b: Double, c: Double): Double {
                                                /* тип обов'язковий */
    // Тіло функції у вигляді блока
    val sd = sqrt(discriminant(a, b, c))
```

```
    val x1 = (-b + sd) / (2 * a)
    val x2 = (-b - sd) / (2 * a)
    return x1 * x2 // Результат
}
```

В цьому прикладі тіло функції записано у вигляді *блоку* в фігурних дужках, в протилежність тілу в вигляді *виразу* — як в функціях `sqr` и `discriminant` вище. Знак рівності при цьому прибирається та обов'язково вказується тип результату функції. В прикладі присутні три проміжні *змінні* — `d`, `x1`, `x2`. Визначення проміжної *змінної* в Котліні починається з *ключового слова* **val** (скорочення від value — значення), за яким іде ім'я змінної та, після знаку рівності — її значення. За бажанням можна також указати тип змінної, наприклад:

```
    // ...
    val sd: Double = sqrt(discriminant(a, b, c))
```

Якщо тип змінної не вказаний, він визначається автоматично, наприклад, в даному випадку він співпадає з типом результату функції `sqrt`.

Блок складається з так званих *операторів* (в прикладі їх чотири), що виконуються по порядку згори донизу. **Перед** використанням будь-якої змінної, її слід визначити. Наприклад, такий запис призвів би до помилки:

```
fun quadraticRootProduct(a: Double, b: Double, c: Double): Double {
    val x1 = (-b + sd) / (2 * a) // Unresolved reference: sd
    val x2 = (-b - sd) / (2 * a) // Unresolved reference: sd
    val sd = sqrt(discriminant(a, b, c))
    return x1 * x2 // Результат
}
```

Останній оператор функції, що починається з *ключового слова* **`return`**, визначає значення її результату; **`return`** перекладається з англійської як **повернути** (результат). Функція `quadraticRootProduct` в першу чергу обчислить значення змінної `sd`, використовуючи **інші функції** `discriminant` та `sqrt`. Потім відбудеться обчислення змінних `x1` та `x2` і лише в кінці — обчислення результату в операторі **`return`**.

Для порівняння, наведемо запис тієї ж функції, що не використовує змінні:

```
fun quadraticRootProduct(a: Double, b: Double, c: Double) =
        ((-b + sqrt(discriminant(a, b, c))) / (2 * a)) *
        ((-b - sqrt(discriminant(a, b, c))) / (2 * a))
```

Хоча і записана в один рядок, така функція є набагато менш зрозумілою, при її написанні легко заплутатись при розстановці дужок. Крім того, в ній

відбувається двократне обчислення кореня з дискримінанту, чого слід уникати.

Функція println та рядкові шаблони

Розглянемо приклад — функція, що розв'язує квадратне рівняння та демонструє розв'язки користувачеві.

```
fun solveQuadraticEquation(a: Double, b: Double, c: Double) {
                                                    /* no result */
    val sd = sqrt(discriminant(a, b, c))
    val x1 = (-b + sd) / (2 * a)
    val x2 = (-b - sd) / (2 * a)
    // Виведення на екран значень x1 та x2
    println(x1)
    println(x2)
    // Виведення на екран рядку вигляду x1 = 3.0 x2 = 2.0
    println("x1 = $x1 x2 = $x2")
    // Виведення на екран добутку коренів
    println("x1 * x2 = ${x1 * x2}")
}
```

Ми підійшли до такої важливої частини програмування, як взаємодія з користувачем та взагалі з зовнішнім для програми світом. Зверніть увагу — тепер функції, які ми використовуємо, починають відрізнятися від чисто математичних, оскільки у них з'являються *побічні ефекти* (side effects). Функція в програмуванні в загальному випадку не зводиться *лише* до залежності між параметрами та результатом.

Функція `println(p)` визначена в стандартній бібліотеці мови Котлін та не потребують підключення будь-яких пакетів. Її параметр p може мати будь-який тип — так, виклик `println(x1)` виведе на окремий рядок *консолі* значення змінної x1. Найчастіше, проте, p є рядком, наприклад, "x1 = $x1 x2 = $x2". В цьому рядку присутні рядкові шаблони $x1 та $x2, що складаються з символу **$** та імені змінної (параметра). Замість них програма автоматично підставить значення відповідних змінних. Рядковий шаблон дозволяє також підставити значення складного виразу, як, наприклад, тут: "x1 * x2 = ${x1 * x2}". В цьому випадку вираз записується в фігурних дужках, щоб програма мала можливість відслідкувати його початок та кінець.
Зверніть увагу, що тип результату функції solveQuadraticEquation не вказаний. Це означає, що функція **не має** результату (в математичному сенсі). Такі

функції зустрічаються доволі часто, один з прикладів — сама функція println, та їхній реальний результат зводиться до їхніх побічних ефектів — наприклад, виведення на консоль.
Залишилось визначити — що саме є *консоль*? У звичній нам операційній системі Windows *консоль* — це вікно або його частина, яку програма використовує для виведення текстової інформації. В Intellij IDEA це вікно можна відкрити послідовністю команд View → Tool windows → Run. При запуску програми з операційної системи, вона сама відкриє так зване "вікно терміналу", яке буде використовуватися програмою для виведення текстової інформації.

Тестові функції

Тестові функції — особливий вид функцій, які призначені для перевірки правильності роботи інших функцій. Оскільки людині властиво помилятися, програмісти винайшли чимало способів, як можна проконтролювати правильність програми, як своєї власної, так і написаної іншими людьми. Тестові функції є одним з таких способів. Розглянемо приклад:

```
// Дозвіл на використання короткого імені анотації org.junit.jupiter.api.Test
import org.junit.jupiter.api.Test
// Дозвіл на використання короткого імені для функції
org.junit.jupiter.api.Assertions.assertEquals
import org.junit.jupiter.api.Assertions.assertEquals

// Класс Tests, наявність класу обов'язкова для бібліотеки JUnit
class Tests {

    // ...

    // Тестова функція
    @Test
    fun testSqr() {
        assertEquals(0, sqr(0))  // Перевірити, що квадрат нуля це 0
        assertEquals(4, sqr(2))  // Перевірити, що квадрат двух це 4
        assertEquals(9, sqr(-3)) // Перевірити, що квадрат -3 це 9
    }
}
```

Написання тестових функцій вимагає підключення до програми однієї з бібліотек автоматичного тестування, наприклад, бібліотеки **JUnit** Більшість класів цієї бібліотеки знаходяться в пакеті org.junit для версії JUnit 4.x або в пакеті org.junit.jupiter.api для версії JUnit 5.x.
`@Test` - це так звана *анотація*, тобто, позначка, яка використовується для надання функції `testSqr` додаткового сенсу. В даному випадку, анотація

показує, що функція `testSqr` - тестова. Функція assertEquals призначена для порівняння результату виклику деякої іншої функції, наприклад, `sqr`, з тим, що очікується. У наведеному прикладі вона викликається тричі.
Тестових функцій в проекті може бути багато, будь-яка з них запускається так само, як і головна функція - натисканням зеленого трикутника зліва від заголовка функції. Тестові функції виконуються за тими ж принципами, що і будь-які інші, але виклики `assertEquals` відбуваються особливим чином:

- *Якщо перевірка показала збіг результату з очікуваним, функція не робить нічого;*
- *В іншому випадку виконання тестової функції завершується і в IDEA з'явиться повідомлення, виділене червоним кольором, про невдале завершення тестової функції.*

Якщо тестова функція завершила роботу і результати всіх перевірок збіглися з очікуваними, тестова функція вважається завершеною успішно.

Нарешті, що ж таке `class Tests`? За правилами бібліотеки JUnit, всі тестові функції повинні бути присутніми всередині деякого *класу*. Про те, для чого потрібні класи, ми поговоримо пізніше. В даному прикладі для цієї мети був створений клас з ім'ям `Tests` (ім'я може бути довільним), і тестова функція була записана в ньому. Зелений трикутник навпроти імені класу дозволяє одночасно запустити всі тестові функції в даному класі.
Будь-яка написана програма або функція **завжди** вимагає перевірки. Ця вимога тим важливіше, чим складніше програма або функція. Тестові функції дозволяють довести правильність роботи перевіряється функції, принаймні, для деяких значень її аргументів.

Поряд з тестовими функціями, може бути використано і *ручне* тестування. Ручне тестування передбачає виведення результатів функції на консоль і ручну перевірку їх з очікуваними. Для ручного тестування може бути використана головна функція, наприклад:

```
fun main() {
    println("sqr(0) = ${sqr(0)}")
    println("sqr(4) = ${sqr(4)}")
}
```

В нормальному випадку ми повинні побачити на консолі рядки

```
sqr(0) = 0

sqr(4) = 16
```

Ручне тестування є набагато більш трудомістким і вимагає від програміста або тестувальника набагато більшої уваги. Тому в сучасному програмуванні рекомендується починати перевірку функцій зі створення тестових функцій, які запускаються кожен раз при зміні програми і дозволяють помітити чи з'явилися помилки. Ручне тестування виконується значно рідше, зазвичай перед випуском нової *версії* програми. Але це зовсім інша історія…

Завдання

Відкрийте файл `Main.kt`

Знайдіть у ньому описи заголовків функцій для свого варіанту.

Наприклад, для 1 варіанту - це будуть функції

```
fun var1calcR(a: Double, b: Double, x: Double) : Double = TODO()
fun var1calcS(a: Double, b: Double, x: Double) : Double = TODO()
```

Замість `TODO()` опишіть реалізацію цих функцій.

Перейдіть до класу тестових функцій `MainKtTest` та виконайте тестування ваших функцій за допомогою відповідних функцій цього класу.

Якщо функції тестування показали, що у Вас є помилки - виправіть їх (свої помилки, не функції!) та повторіть тестування.

Додайте власні функції для тестування розроблених функцій з іншими аргументами та виконайте тестування.

У головній функції програми реалізуйте введення початкових даних, виклик створених функцій, та виведення результатів.

Відобразіть цей процес на блок-схемі.

Оформіть звіт

Варіант	Розрахункові формули	Значення вхідних даних
1	$R = x^2(x+1)/b - \sin^2(x+a); s = \sqrt{\frac{xb}{a}} + \cos^2(x+b)^3$	a=0.7 b=0.05 x=0.5
2	$f = \sqrt[3]{mtgt + \lvert csint \rvert}; z = mcos(btsint) + c$	m=2;c=-1 t=1.2 b=0.7
3	$y = btg^2 x - \frac{a}{\sin^2(x/a)}; d = ae^{-\sqrt{a}}\cos(bx/a)$	a=3.2 b=17.5 x=-4.8
4	$s = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}; f = x(\sin x^3 + \cos^2 y)$	x=0.335 y=0.025
5	$s = x^3 tg^2(x+b)^2 + \frac{a}{\sqrt{x+b}}; Q = \frac{bx^2 - a}{e^{ax} - 1}$	a=16.5 b=3.4 x=0.61
6	$y = e^{-bt}\sin(at+b) - \sqrt{\lvert bt + a \rvert}; s = bsin(at^2 cos2t) - 1$	a=-0.5 b=1.7 t=0.44
7	$y = \sin^3(x^2 + a)^2 - \sqrt{\frac{x}{b}}; z = \frac{x^2}{a} + \cos(x+b)^3$	a=1.1 b=0.004 x=0.2
8	$a = \frac{2\cos(x - \pi/6)}{1/2 + \sin^2 y}; b = 1 + \frac{z^2}{3 + z^2/5}$	x=1.426 y=-1.220 z=3.5
9	$w = \sqrt{x^2 + b} - b^2 \sin^3(x+a)/x; y = \cos^2 x^3 - \frac{x}{\sqrt{a^2 + b^2}}$	a=1.5 b=15.5 x=-2.8
10	$c = \left\lvert x^{y/x} - \sqrt[3]{y/x} \right\rvert; f = (y-x)\frac{y - z/(y-x)}{1 + (y-x)^2}$	x=1.825 y=18.225 z=-3.298