ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«МЕТІНВЕСТ ПОЛІТЕХНІКА»

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА JAVA:

методичні вказівки
до виконання практичних робіт

Запоріжжя 2025

metinvest polytechnic

УДК 004.42(072)
О13

Рекомендовано Науково-методичною радою ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ «МЕТІНВЕСТ ПОЛІТЕХНІКА»
(протокол № 7 від 25.04.2025 р.)

Укладач:
Нікуліна О.М., д-р. техн. наук, професор

О13 **Об’єктно-орієнтоване програмування на Java :** методичні вказівки до виконання практичних робіт / уклад. О. М. Нікуліна. Запоріжжя : ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ «МЕТІНВЕСТ ПОЛІТЕХНІКА», 2025. 66 с.

Методичні вказівки включають теми, мету, теоретичні основи, приклади програм та завдання для кожної практичної роботи, вимоги до оформлення звіту з практичних робіт, список рекомендованої літератури.

Рекомендовано для студентів спеціальності F3 Комп’ютерні науки першого (бакалаврського) рівня освіти.

УДК 004.42(072)

ЗМІСТ

ВСТУП 4

ПРАКТИЧНА РОБОТА 1. Використання базових засобів мови 6

ПРАКТИЧНА РОБОТА 2. Робота з масивами та рядками. Створення класів 19

ПРАКТИЧНА РОБОТА 3. Використання поліморфізму. Робота з узагальненнями та колекціями 32

ПРАКТИЧНА РОБОТА 4. Робота з винятками і файлами 49

ПРАКТИЧНА РОБОТА 5. Створення програм графічного інтерфейсу користувача 56

ВИМОГИ ДО ЗВІТУ З ПРАКТИЧНОЇ РОБОТИ 61

ПОДАННЯ НА ПЕРЕВІРКУ ПРАКТИЧНОЇ РОБОТИ ТА КРИТЕРІЇ ОЦІНЮВАННЯ 63

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ 64

Додаток А ПРИКЛАД ОФОРМЛЕННЯ ТИТУЛЬНОГО ЛИСТА ПРАКТИЧНОЇ РОБОТИ 65

ВСТУП

Вказівки призначені для студентів, які вивчають мову Java на семінарах або самостійно. Класи, шаблони, успадкування, винятки, віртуальні функції, абстрактні класи розглядаються на прикладах, супроводжуваних необхідними теоретичними відомостями.

Предметом навчальної дисципліни «Об'єктно-орієнтованого програмування» є методи алгоритмізації та програмування на мовах C++ та Java. Завдання дисципліни – освоєння методів та засобів об'єктно-орієнтованого програмування у візуальному середовищах Visual Studio та IntelliJ IDEA.

Мета цих методичних вказівок – допомогти студентам засвоїти основи об'єктно-орієнтованого програмування на мові Java.

Практичні роботи передбачають вивчення розділів: побудова та об'ява класів, перевантаження, відкрите та закрите успадкування, віртуальні та абстрактні класи, винятки, використання контейнерів мови програмування Java. Для виконання завдань студент повинен в достатньому ступені володіти процедурним програмуванням на мові Java. До кожної практичної роботи в методичних вказівках наведені: мета роботи, теоретичні основи, приклади та варіанти контрольних задач за розділами курсів, що вивчаються. Варіанти робіт відповідають вимогам навчального плану та силабусу і можуть бути застосовані для здійснення контролю знань студентів зі спеціальностями F3 – «Комп'ютерні науки».

Дисципліна спрямована на отримання здобувачами наступних загальних та спеціальних (фахових) компетентностей:

ЗК1. Здатність до абстрактного мислення, аналізу та синтезу.

ЗК2. Здатність застосовувати знання у практичних ситуаціях.

ЗК3. Знання та розуміння предметної області та розуміння професійної діяльності.

ЗК6. Здатність вчитися й оволодівати сучасними знаннями.

ЗК9. Здатність працювати в команді.

СК3 Здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв'язності та нерозв'язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем.

СК7 Здатність застосовувати теоретичні та практичні основи методології та технології моделювання для дослідження характеристик

і поведінки складних об’єктів і систем, проводити обчислювальні експерименти з обробкою й аналізом результатів.

СК8. Здатність проєктувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління.

Дисципліна спрямована на отримання здобувачами наступних програмних результатів :

ПР5. Проєктувати, розробляти та аналізувати алгоритми розв’язання обчислювальних та логічних задач, оцінювати ефективність та складність алгоритмів на основі застосування формальних моделей алгоритмів та обчислюваних функцій.

ПР9. Розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук.

У результаті виконання практичних робіт здобувач вищої освіти повинен продемонструвати достатній рівень сформованості наступних результатів навчання:

- Здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв’язності та нерозв’язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем.
- Здатність застосовувати існуючі і розробляти нові алгоритми розв’язування задач у галузі комп’ютерних наук.
- Здатність розробляти і реалізовувати проекти зі створення програмного забезпечення, у тому числі в непередбачуваних умовах, за нечітких вимог та необхідності застосовувати нові стратегічні підходи, використовувати програмні інструменти для організації командної роботи над проєктом.
- Здатність проєктувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмів управління.

ПРАКТИЧНА РОБОТА 1

Використання базових засобів мови Java

Мета практичної роботи – опанувати написання програм з використанням базових засобів мови Java.

1.1 Теоретичні основи

У Java немає глобальних змінних, функцій чи процедур. Це зроблено з метою запобігання конфліктів імен. Програма складається з одного чи більше описів класів з полями (елементами даних) і методами (функціями-елементами). Класи в Java можуть мати модифікатор **`public`** Такі класи мають назву відкритих (публічних). Один із відкритих класів повинен визначати статичний метод `main()`, з якого починається виконання програми. Програмний код визначення класів слід зберігати у текстових файлах з розширенням `.java`. Ім'я класу (без розширення) повинне збігатися з іменем публічного класу, який міститься у цьому файлі. Слід пам'ятати, що великі й маленькі літери відрізняються [5].

Локальні змінні визначаються (створюються) всередині методів. Опис локальних змінних у Java здійснюється аналогічно C++. Наприклад:

```
int i = 11;
double d= 0, x;
float f;
int j, k;
```

Локальні змінні можуть бути визначені в будь-якому місці усередині тіла функції, а також у вкладеному блоці. У Java не можна у внутрішньому блоці визначати імена, вже описані в зовнішньому блоці:

```
{
    int i = 0;
    {
        int j = 1; // Змінна j визначається у внутрішньому блоці
        int i = 2; // Помилка! Змінна i визначена в зовнішньому блоці
    }
}
```

У Java не можна оголошувати змінні без їхнього створення.

Ключове слово **`final`** стосовно до імен змінних означає, що вони не можуть бути змінені.

```
final int h = 0;
```

Слово **`const`** зарезервоване, але не використовується.

Примітивні, чи базові типи поділяються на цілі типи, типи з

крапкою, що плаває, символьні і булеві. [5]

Всередині методів класів, з яких складається програма, можуть міститись твердження (оператори), які, в свою чергу, складаються з виразів. *Вираз* складається з однієї чи кількох операцій. Об'єкти операцій мають назву *операндів*. Операції бувають *унарними* (один операнд), *бінарними* (два операнда) і *тернарними* (три операнда).

У Java підтримуються практично всі стандартні арифметичні і логічні операції, а також операції порівняння мови C++. Ці операції мають такий же пріоритет і асоціативність, як і в C++.

До *арифметичних операцій* належать +, - (бінарні й унарні), *, /, а також операція отримання залишку від ділення % (тільки до цілих). Якщо / застосовується до цілих, результатом ділення буде теж ціле, а залишок відкидається. Якщо хоча б один операнд – типу з плаваючою крапкою (дійсний), ми отримаємо дійсний результат. [5]

```
System.out.println(1 / 2);   // 0
System.out.println(1.0 / 2); // 0.5
```

До *операцій відношення* належать перевірка на рівність == і на нерівність !=, а також перевірки > (більше) >= (більше або дорівнює) < (менше) <= (менше або дорівнює). До *логічних операцій* належать логічне І (&&), АБО (||) та НІ (!). Операції відношення та логічні повертають значення типу **boolean**. Операції І та АБО можна застосовувати побітові & і |. У такому випадку завжди здійснюється повне обчислення значень обох операндів, тоді як обчислення значення другого операнду операцій && і || може не здійснюватись, якщо результат вже встановлено.

До *операцій присвоювання* відносяться операції простого і складеного присвоювання. Результатом операції простого присвоювання є значення того виразу, що присвоюється лівому операнду. Складене присвоювання можна представити в загальному вигляді в такий спосіб:

```
a op= b
```

У цьому випадку op – арифметична чи побітова операція: + - * / % | & ^ << >>. Кожна складена операція еквівалентна присвоюванню:

```
a = (a) op (b);
```

Наприклад,

```
x += 5;
```

що еквівалентно

```
x = x + 5;
```

Операція інкременту ++ забезпечує збільшення цілої змінної на одиницю. Вона має дві форми префіксну і постфіксну. Префіксна форма забезпечує збільшення змінної до того, як значення операції буде використано, а постфіксна – після. Операція декременту -- забезпечує зменшення змінної на одиницю і правила її використання аналогічні.

На відміну від C++, *операція "кома"* може бути застосована тільки в заголовках циклів, наприклад:

```
int i, j;
for (i = 0, j = 0; i < 10; i++, j += 2)
{
    System.out.println(i + " " + j);
}
```

Використання побітових операцій & (побітове І), | (побітове АБО), ^ (виключальне АБО), ~ (побітове не), << та >> (побітовий зсув) аналогічне C++. У Java існує додаткова операція беззнакового зсуву (>>>). [5]

Умовна операція (тернарна) має такий вигляд:

```
умова ? вираз1 : вираз2
```

Спочатку обчислюється значення умови. Якщо воно істинне, то обчислюється вираз1 і його значення повертається умовною операцією. Якщо значення умови хибне, то обчислюється вираз2 і повертається його значення. У наведеному нижче прикладі обчислюється мінімальне з двох чисел:

```
min = a < b ? a : b;
```

Порядок застосування унарних операцій та операцій присвоєння "справа наліво", а всіх інших операцій – "зліва направо".

Пріоритет додавання і віднімання нижче, ніж множення і ділення, пріоритет присвоєння нижче, ніж арифметичних операцій і т.д. Для зміни послідовності операцій слід використовувати дужки. [5]

У Java не допускається безпосередня робота з указівниками, отже, немає операцій розіменування, вибору елемента за вказівником і узяття адреси (*, -> та &).

У Java немає операції **sizeof**() оскільки її головне призначення в C++ – з'ясовувати розміри тих чи інших даних під час перенесення вихідного тексту на іншу платформу. У Java розміри всіх типів стандартизовані.

Твердження (або інструкція, іноді оператор, англ. statement) – найменша автономна частина мови програмування. Програма являє собою послідовність інструкцій. Більшість тверджень мови Java

аналогічна твердженням C++.

Порожня інструкція складається з однієї крапки з комою.

Інструкція-*вираз* є повний вираз, який закінчується крапкою з комою. Наприклад:

```
k = i + j + 1;
```

Складена інструкція – це послідовність тверджень, укладена у фігурні дужки. Складену інструкцію часто іменують блоком. Після фігурної дужки, яка закриває блок, крапка з комою не ставиться. Синтаксично блок може розглядатися як окрема інструкція, однак вона також має значення у визначенні видимості і часу життя ідентифікаторів. Ідентифікатор, оголошений усередині блоку, має область видимості від точки визначення до фігурної дужки, що закривається. Блоки можуть необмежено вкладатися один в одного. [5]

Інструкції вибору – умовна інструкція та перемикач. Умовна інструкція застосовується в двох видах:

```
if (вираз-умова)
    інструкція1
else
    інструкція2
```

або

```
if (умова)
    інструкція1
```

Під час виконання цієї інструкції обчислюється вираз-умова і, якщо це істина, то виконується `інструкція1` а інакше – `інструкція2`. На відміну від C++, вираз-умова може бути лише типу **`boolean`**.

Перемикач дозволяє вибрати одну з кількох можливих гілок обчислень і будується за схемою:

```
switch (цілий_вираз)
    блок
```

Блок має такий вигляд:

```
{
    case константа_1: інструкції
    case константа-2: інструкції
    ...
    default: інструкції
}
```

Виконання перемикача полягає в обчисленні керуючого виразу і переході до групи інструкцій, позначених **`case`**-міткою, значення якої дорівнює керуючому виразу. Якщо такої мітки немає, виконуються інструкції після мітки **`default`** (яка може бути відсутня). Під час виконання перемикача відбувається перехід на інструкції з обраною міткою і далі

інструкції виконуються у нормальному порядку. Для того, щоб не виконувати інструкцій, які залишилися у тілі перемикача, необхідно використовувати оператор **break**. [5]

Починаючи з Java 7, окрім цілих та символів, у конструкції **switch**() можна використовувати рядки. В цьому випадку константи варіантів (після **case**) повинні теж бути рядками, наприклад:

```
case "some text":
```

Інструкції циклу реалізовані в трьох варіантах: цикл із передумовою, цикл із постумовою і цикл із параметром.

Цикл із передумовою будується за схемою

```
while (вираз-умова)
    інструкція
```

На кожному повторенні циклу обчислюється вираз-умова і якщо значення цього виразу дорівнює **true**, виконується інструкція – тіло циклу. Цикл із постумовою будується за схемою

```
do
    інструкція
while (вираз-умова);
```

Вираз-умова обчислюється і перевіряється після кожного повторення інструкції – тіла циклу, цикл повторюється, поки умова виконується. Тіло циклу в циклі з постумовою виконується принаймні один раз.

Цикл із параметром будується за схемою:

```
for (E1; E2; E3)
    інструкція
```

де E1, E2 і E3 – вирази скалярного типу. Цикл з параметром реалізується за таким алгоритмом:

- обчислюється вираз E1 (зазвичай цей вираз виконує підготовку до початку циклу);
- обчислюється вираз E2 і якщо він дорівнює **false**, виконується перехід до наступної інструкції програми (вихід з циклу);
- якщо E2 дорівнює **true**, виконується інструкція – тіло циклу;
- обчислюється вираз E3 – виконується підготовка до повторення циклу, після чого знову виконується вираз E2. [5]

Циклічні конструкції можна вкладати одну в іншу.

У сполученні з інструкціями циклу використовуються інструкції переходу – оператор **break**, який дозволяє перервати виконання найвнутрішнішого з циклів, оператор **continue**, який перериває поточну

ітерацію найвнутрішнішого з циклів **while**, **do** або **for**. Найчастіше **break** використовують у такій конструкції:

```
if (умова_дострокового_завершення_циклу)
    break;
```

У Java після ключових слів **break** та **continue** можна розташувати мітку, що передує одному з вкладених циклів. У цьому випадку твердження стосуються не до найвнутрішнішого, а до позначеного циклу. [5] Наприклад:

```
  int a;
  . . .
  double b = 0;
label:
  for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
          if (i + j + a == 0) {
              break label;
          }
          b += 1.0 / (i + j + a);
      }
  }
```

Твердження **goto** не використовується в мові Java, але **goto** є зарезервованим словом. Це слово ніяк не може бути використано.

У Java немає глобальних функцій. Аналогом глобальної функції є статичний метод. Опис статичної функції у найпростішому випадку має таку структуру:

```
static тип_результату ім'я_функції(список_формальних_параметрів) тіло
```

Параметри (аргументи) функції, що вказуються в списку у визначенні функції, називаються формальними. Параметри, що вказуються під час виклику функції, називаються фактичними. Під час виклику функції виділяється пам'ять під її формальні параметри, потім кожному формальному параметру присвоюється значення фактичного параметра.

Тіло функції являє собою складений оператор (блок). Наприклад, визначення статичної функції, що обчислює суму двох цілих чисел, може бути таким:

```
static int sum(int a, int b)
{
    int c = a + b;
    return c;
}
```

Можна взагалі обійтися без змінної c:

```
static int sum(int a, int b)
{
```

```
    return a + b;
}
```

Виклик функції може бути здійснений у виразі в описі або в тілі іншої функції. Під час виклику функції вказується її ім'я і список фактичних параметрів без зазначення їхніх типів:

```
int x = 4;
int y = 5;
int z = sum(a, b);
int t = sum(1, 3);
```

Функція може бути без параметрів:

```
int zero()
{
    return 0;
}
```

Викликаючи таку функцію, також необхідно використовувати дужки:

```
System.out.println(zero());
```

Інструкція **return** у тілі функції забезпечує завершення роботи функції. Значення виразу після **return** стає значенням функції, яке ця функція повертає.

Функція може не повертати ніякого результату. Для позначення цього використовується тип **void**.

```
void hello()
{
    System.out.println("Hello!");
}
```

У цьому випадку в тілі функції **return** може бути відсутнім. Якщо інструкція **return** присутня, то після неї не повинно бути ні якого виразу. Таку функцію можна викликати тільки окремою інструкцією.

```
hello();
```

Параметри передаються до функцій за значенням, тобто значення фактичних параметрів копіюються в пам'ять, відведену для формальних параметрів. При цьому значення, з якими працює функція – це її власні локальні копії фактичних параметрів і їхня зміна на ці параметри не впливає [5]. Таким чином, під час передачі за значенням вміст фактичних параметрів не змінюється:

```
static void f(int k) {
    k++;        // k = 2;
}

public static void main(String[] args)
{
    int k = 1;
    f(k);
    System.out.println(k); // k = 1;
```

```
}
```

У Java можна перевантажувати імена функцій. Перевантаженням імені називається його використання для позначення різних операцій над різними типами.

Якщо списки формальних параметрів збігаються, але типи значень, що повертаються, різні, компілятор повідомляє про помилку. Якщо списки формальних параметрів двох функцій розрізняються числом чи параметрів їхніми типами, то ці дві функції вважаються перевантаженням однієї функції і повинні мати різні визначення.

Перевантажені функції використовуються в тих випадках, якщо кілька функцій виконує схожі дії над об'єктами різних типів і зручно дати однакові імена всім цим функціям [5].

Для того, щоб визначити, яку саме функцію варто викликати, порівнюються кількість і типи фактичних параметрів, зазначені у виклику, з кількістю і типами формальних параметрів всіх описів функцій з таким ім'ям. У результаті викликається та функція, у якої формальні параметри щонайкраще зіставилися з параметрами виклику, чи видається помилка, якщо такої функції не знайшлося.

Статичні функції в межах класу викликаються з застосуванням лише імені функції та списку фактичних параметрів. Для того, щоб викликати статичну функцію іншого класу, необхідно вказувати його ім'я і далі через точку ім'я функції. В такий спосіб можна звертатись до імен у межах пакету, а також до класів і функцій пакету `java.lang`. Цей пакет містить класи з дуже корисними функціями. Наприклад, клас `Math` надає велику кількість математичних функцій [5].

Найбільш зручним засобом уведення даних є клас `java.util.Scanner`. Клас `Scanner` надає функції для читання даних з різних джерел, наприклад, з файлів. У нашому випадку ми вказуємо на необхідність читання з клавіатури (стандартний потік введення `System.in`). Об'єкт-сканер можна застосувати для читання даних різних типів. Наприклад, функція `next()` повертає наступне прочитане значення типу `String`, `nextInt()` дозволяє отримати ціле, `nextDouble()` повертає прочитане число типу **`double`**. Є також функції `nextBoolean()`, `nextByte()`, `nextShort()`, `nextLong()`, `nextFloat()` тощо. Усі функції обумовлюють зупинку виконання програми, яке поновлюється після введення з клавіатури відповідного значення. Для можливості роботи з класом `Scanner` на початку вихідного коду слід додати:

```
import java.util.Scanner;
```

Це дозволить використовувати ім'я `Scanner` без додаткового префіксу. Спочатку треба створити об'єкт цього класу за допомогою операції **`new`** (докладно ця операція буде розглянута пізніше). Зв'язуємо об'єкт-сканер зі стандартним `System.in`. Далі можна читати дані різних типів [5]. Наприклад:

```
import java.util.Scanner;


public class ScannerTest
{
    @SuppressWarnings("resource")
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String s = scanner.next();       // читання рядка
        double d = scanner.nextDouble(); // читання дійсного числа
        int i = scanner.nextInt();       // читання цілого числа
        // ... використання введених даних
    }
}
```

Великою зручністю класу є можливість довільного розташування даних у вхідному потоці: окремі дані можна розділяти пропусками, табуляцією, або переведенням рядка. Єдиний виняток – використання функції `nextLine()`, яка читає дані до кінця рядка.

Виведення у консольне вікно здійснюється за допомогою функцій об'єкта `out` класу `System`. Функція `print()` дозволяє вивести результат без переходу на новий рядок, `println()` здійснює перехід на новий рядок після виведення.

Можна також скористатися функцією `printf()` для форматованого виведення. Використання цього методу аналогічне використанню відповідної функції мови С. Перший параметр – так званий рядок форматування. Далі можна вказати довільну кількість параметрів, значення яких слід вивести на консоль [5]. Наприклад:

```
    double d = 3.5;
    int i = 12;
    System.out.printf("%f %d%n", d, i);
```

1.2. Приклади

Приклад 1.1. Середнє арифметичне [5]

Необхідно ввести з клавіатури два числа (дійсне та ціле) і обчислити їх середнє арифметичне.

Одержуємо такий код:

```
public class Average
{
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Тепер замість коментаря всередині функції `main()` створюємо об'єкт класу `Scanner`, описуємо дві змінні, наприклад, `a` і `b`, читаємо їхні значення з клавіатури і обчислюємо значення `c`, що є їхнім середнім арифметичним.

```
Scanner s = new Scanner(System.in);
double a = s.nextDouble();
int b = s.nextInt();
double c = (a + b) / 2;
```

Для забезпечення можливості використання класу `Scanner` слід перед описом класу Average додати інструкцію імпорту (**`import java.util.Scanner;`**). Визначення (опис) змінної, як і в C++, можна поєднувати з присвоєнням початкового значення (ініціалізацією). У цьому випадку створюється змінна `s` типу `Scanner`, `b` типу **`int`**, а також змінні `a` і `c` типу **`double`**.

Після ініціалізації змінної c необхідний результат вже обчислений. Його необхідно вивести на екран (у консольне вікно) з використанням функції `println()` об'єкта `out` класу `System`.

```
System.out.println("Середнє арифметичне: " + c);
```

У цьому рядку плюс означає не додавання, а зшивання рядків. До строкової константи `"Середнє арифметичне: "` пришивається рядкове представлення значення змінної `c`.

Тепер можна навести повний текст програми.

```
import java.util.Scanner;

public class Average
{
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        double a = s.nextDouble();
        int b = s.nextInt();
        double c = (a + b) / 2;
        System.out.println("Середнє арифметичне: " + c);
```

```
    }
}
```

Після запуску програми слід перейти у підвікно Console, увести в різних рядках значення a та b, після чого в консольному вікні з'явиться результат.

Приклад 1.2. [5] Припустимо, необхідно обчислити найбільший спільний дільник. Скористаємось алгоритмом Евкліда, який полягає у послідовному відніманні меншого числа з більшого та заміні більшого числа отриманою різницею. Алгоритм завершується, коли два числа збігатимуться. Програма матиме такий вигляд:

```
import java.util.Scanner;

public class GreatestCommonDivisor
{
    static int gcd(int m, int n) {
        while (m != n) {
            if (m > n)
                m -= n;
            else
                n -= m;
        }
        return m;
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int m = s.nextInt();
        int n = s.nextInt();
        System.out.println(gcd(m, n));
    }
}
```

Результат роботи можна використати для обчислення найменшого спільного кратного чисел. Для цього їхній добуток слід поділити на найбільший спільний дільник.

Приклад 1.3. Обчислення виразу [5]

Необхідно прочитати з клавіатури x та обчислити *y* за такою формулою:

$$y = \begin{cases} \sum_{i=1}^{n} (i-x)^2, x < 0 \\ \prod_{i=1}^{n} (x+i), x \geq 0 \end{cases}$$

де n дорівнює 6. Обчислення виразу слід розмістити в окремій функції. Програма може мати такий вигляд:

```
import java.util.Scanner;

public class Formula
{
    public static double f(double x) {
        final int n = 6;
        double y;
        if (x < 0) {
            y = 0;
            for (int i = 1; i <= n; i++) {
                y += (i - x) * (i - x);
            }
        }
        else {
            y = 1;
            for (int i = 1; i <= n; i++) {
                y *= (x + i);
            }
        }
        return y;
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        double x = s.nextDouble();
        double y = f(x); // Виклик функції
        System.out.println("x = " + x + "\ty = " + y);
    }
}
```

1.3. Задачі [5]

Створити консольну програму, в якій здійснюється обчислення значень функції на певному інтервалі. У програмі треба визначити значення початку інтервалу, кінця інтервалу, а також величини кроку, з яким змінюється аргумент. Відповідні значення слід прочитати з клавіатури.

Безпосередньому створенню програми повинне передувати дослідження поведінки функції на різних інтервалах.

Програма повинна містити визначення та введення необхідних даних і один великий цикл, у тілі якого здійснюється

- обчислення функції різними способами, залежно від значення аргументу

- виведення на консоль аргументу та результату на кожному кроці

циклу.

Програма повинна містити окрему статичну функцію для обчислення значення *y* залежно від значення аргументу *x*. Варіант функціональної залежності, який слід реалізувати у програмі, визначається відповідно до номеру студента у списку групи.

Таблиця 1.1 – Індивідуальні завдання

№	Функція	n	№	Функція	n
1	$y=\begin{cases}0.25x+14\sum_{i=1}^{n}\sin^{i}(0.5x-2), & x<4\\ \sqrt{x-3}, & x\ge 4\end{cases}$	9	6	$y=\begin{cases}3\cos(x-3)+2x, & x<3\\ \sum_{k=1}^{n}e^{-k(x-3)}, & x\ge 3\end{cases}$	9
2	$y=\begin{cases}\sum_{i=1}^{n}\cos((0.2x-1)^{i}), & x\le 5\\ \sqrt[3]{2x-9}+7, & x>5\end{cases}$	8	7	$y=\begin{cases}-0.2x+\sum_{k=1}^{n}ke^{0.2x-1}, & x\le 5\\ 5\cos(0.6x-3)+15, & x>5\end{cases}$	7
3	$y=\begin{cases}x+12\prod_{i=1}^{n}\sin(0.1x-0.02i+1), & x\le 20\\ -\sqrt[3]{0.5x-9}+2n+1, & x>20\end{cases}$	8	8	$y=\begin{cases}3x+6+\sum_{k=1}^{n}(1-\frac{x}{4})^{k}, & x<4\\ 4\cos(x-4)+14, & x\ge 4\end{cases}$	9
4	$y=\begin{cases}3x+\sum_{k=1}^{n}\sin^{k}(x-7), & x<7\\ 23-2e^{7-x}, & x\ge 7\end{cases}$	9	9	$y=\begin{cases}\sum_{k=1}^{n}k^{x-4}, & x<5\\ \cos(0.2x-1)+x+15, & x\ge 5\end{cases}$	8
5	$y=\begin{cases}-x+\sum_{k=1}^{n}k\cos(x-3), & x\le 3\\ 22+3e^{3-x}, & x>3\end{cases}$	9	10	$y=\begin{cases}10+\prod_{k=1}^{n}(k+e^{1-0.125x}+1), & x\le 8\\ \sin(0.5x-4)+138-x, & x>8\end{cases}$	9

ПРАКТИЧНА РОБОТА 2

Робота з масивами та рядками. Створення класів

Мета практичної роботи – опанувати написання програм з використанням масивів та рядків, створення класів.

2.1. Теоретичні основи [4]

Клас – це структурований тип даних, набір елементів даних різних типів і функцій для роботи з цими даними. Опис класу складається зі специфікаторів (наприклад, `public, final`), імені, імені базового класу, списку інтерфейсів і тіла у фігурних дужках.

Тіло класу містить поля (їм відповідають елементи даних у C++) і методи (функції-елементи в C++). Поля і методи разом іменуються елементами (членами) класу. Нижче наводиться приклад опису класу:

```
class Rectangle
{
    double width;
    double height;
    double area() {
        return width * height;
    }
}
```

Після останньої фігурної дужки, що закривається, не слід ставити крапку з комою. Методи завжди реалізуються усередині визначення класу.

Під час створення об’єкта класу поля ініціалізуються усталеними значеннями (нулями або `null` для посилань). Java допускає ініціалізацію полів початковими значеннями:

```
class Rectangle
{
    double width = 10;
    double height = 20;
    double area() {
        return width * height;
    }
}
```

Можна створити спеціальний блок ініціалізації усередині тіла класу. Такий блок виконуватиметься щораз під час створення нового об’єкта:

```
class Rectangle
{
    double width;
```

```
    double height;
    {
        width = 10;
        height = 20;
    }
    double area() {
        return width * height;
    }
}
```

Для того, щоб працювати з полями і методами класу, необхідно створити об’єкт. Для цього спочатку створюють посилання на об’єкт, а потім за допомогою операції `new` створюють сам об’єкт шляхом виклику конструктора. Ці дії можна поєднати. Після цього можна викликати методи і використовувати поля:

```
Rectangle rect = new Rectangle();
double a = rect.area();
rect.width = 15
double b = rect.area();
```

Під час виклику методів аргументи передаються за значенням.

Ключове слово `this` використовується як посилання на об’єкт, для якого викликаний метод. Усі нестатичні методи неявно отримують посилання на об’єкт для якого вони використані. Ключове слово `this` використовувати явно, наприклад, коли треба повернути з функції посилання на поточний об’єкт, або запобігти конфлікту імен.

Як і C++, Java підтримує закритий (`private`), пакетний, захищений (`protected`) і відкритий (`public`) рівні доступу. Сам клас може бути оголошений як `public`. На відміну від C++, Java вимагає окремої специфікації доступу для кожного елемента, або групи полів одного типу:

```
public class Rectangle
{
    private double width = 10;
    private double height = 20;
    {
        width = 30;
        height = 40;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getWidth() {
        return width;
    }
    public void setHeight(double height) {
        this.height = height;
```

```
    }
    public double getHeight() {
        return height;
    }
    public double area() {
        return width * height;
    }
}
```

У Java немає ключового слова `friend`, яке у C++ забезпечує доступ до закритих елементів ззовні класу.

Елементи класу без атрибутів доступу мають пакетну видимість. Такий доступ ще називають "дружнім". Всі інші класи цього пакета мають доступ до таких елементів як до відкритих. Ззовні пакета такі елементи взагалі недоступні.

Інкапсуляція (приховування даних) – одна з трьох парадигм об'єктно-орієнтованого програмування. Зміст інкапсуляції полягає у приховуванні від зовнішнього користувача деталей реалізації об'єкту. Зокрема доступ до даних (полів), які зазвичай описані з модифікатором `private`, здійснюється через відкриті функції доступу. Як правило, це так звані сеттери та геттери. Якщо поле має ім'я `name`, відповідні функції доступу мають імена `setName` та `getName`.

Екземпляр класу створюється шляхом застосування операції `new` до конструктора. Конструктор – це функція, яка здійснює ініціалізацію даних об'єкта. Ім'я конструктора збігається з ім'ям класу. Не можна вказувати типу результату конструктора. У класі може бути визначено кілька конструкторів. Якщо жоден конструктор явно не визначений, автоматично створюється усталений конструктор (без параметрів). Такий конструктор ініціалізує всі поля усталеними початковими значеннями. Після визначення принаймні одного конструктора усталений конструктор автоматично не створюється.

Один конструктор можна викликати з іншого з використанням слова `this`, після якого випливають необхідні аргументи. Спочатку здійснюється ініціалізація в місці опису, після якої значення можуть бути перевизначені в блоці ініціалізації, а потім перевизначені в конструкторі

```
public class Rectangle
{
    private double width = 10;
    private double height = 20;
    {
        width = 30;
        height = 40;
```

```
    }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle() {
        this(50, 60); // виклик іншого конструктора
    }
}
Rectangle rectangle = new Rectangle();
```

У Java немає конструкторів копіювання і деструкторів. Можна створити спеціальний метод `finalize()`, який викликається збирачем сміття перед ліквідацією об'єкта. У деяких випадках об'єкт може бути не вилучений збирачем сміття ніколи (пам'яті вистачало до кінця програми), отже метод `finalize()` може бути ніколи не викликаний.

Методи і поля можуть бути оголошені з ключовим словом `static`. Звернення до таких полів і методів може здійснюватися без створення екземпляра класу. Статичні поля є альтернативою відсутнім у Java глобальним змінним. На відміну від C++, статичні елементи даних не потрібно окремо визначати в глобальній області видимості. Статичні поля можуть бути проініціалізовані під час створення:

```
class SomeClass
{
    static double x = 10;
    static int    i = 20;
}
```

Можна створити окремий блок статичної ініціалізації:

```
class SomeClass
{
    static double x;
    static int i;
    static {
        x = 10;
        i = 20;
    }
}
```

На відміну від нестатичної ініціалізації, створення й ініціалізація статичних полів здійснюється під час першого звернення до класу (створенні екземпляра класу чи зверненні до статичних елементів). Java не створює статичних полів для класів, які не використовуються.

Статичні методи не отримують посилання на об'єкт і не можуть використовувати посилання `this`. Звернення до статичних елементів може здійснюватися як через ім'я класу, так і через посилання на об'єкт:

```
SomeClass.x = 30;
```

```
SomeClass s = new SomeClass();
s.x = 40;
```

Всередині класів можна визначати константи. Константи можуть бути двох видів – статичні й нестатичні. Статичну константу створює компілятор. За угодою її ім'я має містити лише великими літерами:

```
public static final double PI = 3.14159265;
```

Значення нестатичної константи слід визначити, причому один раз – у місці визначення, в блоці ініціалізації (тоді це значення буде однаковим для всіх екземплярів), або в конструкторі:

```
public class ConstDemo
{
    public final int one = 1;
    public final int two;
    {
        two = 2;
    }
    public final int other;
    public ConstDemo(int other) {
        this.other = other;
    }
}
```

Цілком безпечно визначати константи як `public`, оскільки компілятор не дозволить змінити їх значення.

Визначення класу може бути розміщене всередині іншого класу. В такий спосіб можуть бути створені вкладені класи, які можуть бути статичними вкладеними або внутрішніми. Вкладені класи можуть використовуватися як усередині обхопного класу, так і поза ним.

```
class Outer
{
    class Inner {
        int i;
    };
    Inner inner = new Inner();
}
class Another
{
    Outer.Inner i;
}
```

Вкладені класи можуть бути оголошені зі специфікаторами `public`, `private` або `protected`.

Локальні класи створюють всередині блоків. Існує також спеціальний різновид локальних класів – безіменні класи.

Нестатичні вкладені класи називають також *внутрішніми*. Головною відмінністю внутрішніх класів у Java є те, що об'єкти цих

класів отримують посилання на об’єкт обхопного класу. З цього факту випливає два важливих висновки: об’єкти внутрішніх класів мають прямий доступ до даних об’єкта обхопного класу та для створення об’єкта внутрішнього класу обов’язково мати в наявності об’єкт обхопного класу.

У зв’язку з цим в Java запропонований спеціальний механізм створення об’єктів внутрішніх класів. Цей механізм проілюстрований на наведеному нижче прикладі.

```
class Outer
{
    int k = 100;
    class Inner
    {
        void show() {
            System.out.println(k);
        }
    }
}
public class Test
{
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Нестатичні внутрішні класи не можуть містити статичних елементів.

Слід пам’ятати, що об’єкт внутрішнього класу автоматично не створюється. Створення об’єкта може бути передбачене в конструкторі чи у будь-якому методі обхопного класу, а також поза ним (якщо цей клас не оголошений як `private`). Можна також створити масив об’єктів внутрішнього класу. Кожен з таких об’єктів матиме доступ до посилання на обхопний об’єкт.

Внутрішні класи можуть мати свої базові класи. В такий спосіб за допомогою внутрішніх класів можна змоделювати відсутній у Java механізм множинного спадкування:

```
class FirstBase
{
    int a = 1;
}
class SecondBase
{
    int b = 2;
```

```
}
class Outer extends FirstBase {
    int c = 3;
    class Inner extends SecondBase {
        void show() {
            System.out.println(a);
            System.out.println(b);
            System.out.println(c);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Наведений приклад має суто теоретичний сенс, оскільки множинне успадкування класів незалежно від способів його реалізації є небезпечним з точки зору можливого конфлікту імен.

До внутрішніх класів також відносяться локальні. До таких класів не можна звернутися ззовні блоку, у якому вони визначені. Локальні класи найчастіше поміщають у тіло функції:

```
void f() {
    class Local {
        int j;
    }
    Local l = new Local();
    l.j = 100;
    System.out.println(l.j);
}
```

Можна також розміщати локальні класи всередині окремих блоків.

Безіменний клас може реалізовувати певний інтерфейс, перекривати абстрактні функції базового класу чи розширювати його. Для створення об’єкта безіменного класу здійснюється виклик конструктора базового класу, або вказується ім’я інтерфейсу з круглими дужками, після чого розташовують тіло безіменного класу:

```
new Object() {
    // Додавання нового методу:
    void hello() {
        System.out.println("Привіт!");
    }
}.hello();
System.out.println(new Object() {
    // Перевизначення методу:
```

```
    @Override public String toString() {
        return "Це безіменний клас.";
    }
});
```

Безіменні класи не можуть бути абстрактними. Безіменний клас завжди є внутрішнім класом; він не може бути статичним. Безіменні класи автоматично є фінальними (`final`). У наведеному нижче прикладі безіменний клас створюється для визначення способу сортування масиву рядків:

```
void sortByABC(String[] a) {
     Arrays.sort(a, new Comparator<String>() {
        public int compare(String s1, String s2) {
             return (s1).compareTo(s2);
         }
     });
}
```

У безіменних класів не може бути явних конструкторів. Разом з тим, завжди створюється усталений безіменний конструктор. Якщо в базового класу немає конструктора без параметрів, необхідні параметри конструктора вказуються в дужках під час створення об’єкта. Для того, щоб безіменні класи мали доступ до локальних елементів зовнішніх блоків, ці елементи повинні бути описані як `final`.

Статичні вкладені класи мають доступ тільки до статичних елементів обхопних класів. Об’єкти таких класів можуть бути створені без створення об’єктів обхопних класів:

```
class Outer
{
    int k = 100;
    static int m = 200;
    static class Inner {
        void show() {
            // k недоступно
            System.out.println(m);
        }
    }
}
public class Test
{
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.show();
    }
}
```

Статичні вкладені класи можуть містити свої статичні елементи, у тому числі свої вкладені статичні і нестатичні класи.

Класи можна створювати всередині інтерфейсів. Такі класи автоматично є статичними. Усередині класів також можна створювати інтерфейси, що є також статичними.

Під композицією класів розуміють створення нових класів з використанням об’єктів інших класів як полів. Java не дозволяє розміщення об’єктів усередині інших об’єктів, можна тільки описувати посилання. Композиція класів у цьому випадку припускає створення об’єктів безпосередньо (у тілі класу при оголошенні полів) чи в конструкторах.

```
class X
{
}
class Y
{
}
class Z
{
    X x = new X();
    Y y;
    Z() {
        y = new Y();
    }
}
```

Можна також створити внутрішній об’єкт безпосередньо перед його першим використанням. Відношення, що моделюється композицією, часто називають відношенням "has-a".

Агрегування – це різновид композиції, який передбачає, що сутність (екземпляр) міститься в іншій сутності або не може бути створена та існувати без сутності, яка її охоплює. При цьому сутність, що охоплює, може існувати без внутрішньої, тобто час життя зовнішньої та внутрішньої сутностей може не збігатися. Більш строге трактування композиції (власне композиція) передбачає, що час життя зовнішньої та внутрішньої сутностей збігається. На рівні Java агрегування передбачає можливість створення внутрішнього об’єкта перед його використанням, тоді як строга композиція передбачає створення внутрішнього об’єкта в тілі класу, в блоці ініціалізації або в конструкторі.

2.2. Приклади

Приклад 2.1. Сума елементів масиву [5].

Наведена нижче програма знаходить суму елементів масиву дійсних чисел.

```
public class SumOfElements
{
    public static void main(String[] args) {
        double[] a = {1, 2, 1, 2.5, 1};
        double sum = 0;
        for (int i = 0; i < a.length; i++) {
            sum += a[i];
        }
        System.out.println("Sum is " + sum);
    }
}
```

Другий варіант тієї ж програми:

```
public class SumOfElements
{
    public static void main(String[] args) {
        double a[] = {1, 2, 1, 2.5, 1};
        double sum = 0;
        for (double x : a) {
            sum += x;
        }
        System.out.println("Sum is " + sum);
    }
}
```

Третій варіант можна реалізувати за допомогою рекурсії:

```
public class SumWithRecursion
{
    static double sum(double[] a, int n) {
        if (n <= 0) {
            return 0;
        }
        n--;
        return a[n] + sum(a, n);
    }

    static double sum(double[] a) {
        return sum(a, a.length);
    }

    public static void main(String[] args) {
        double[] a = { 1, 2, 1, 2.5, 1 };
        System.out.println("Sum is " + sum(a));
    }
}
```

Приклад 2.2. Обчислення факторіалів [5]

Припустимо, необхідно розробити функцію обчислення факторіалів (від 0 до 20 включно) з використанням допоміжного масиву (статичного поля). Під час першого виклику функції масив заповнюється

до необхідного числа. Під час наступних викликів число або повертається з масиву, або обчислюється з використанням останнього числа, що зберігається у масиві, з подальшим заповненням масиву.

Необхідно також здійснити тестування функції для різних чисел, що вводяться у довільному порядку. Програма матиме такий вигляд:

```
public class Factorial
{
    private static long[] f = new long[30];
    private static int last = 0;

    public static long factorial(int n) {
        f[0] = 1;
        if (n > last) {
            for (int i = last + 1; i <= n; i++) {
                f[i] = i * f[i - 1];
            }
            last = n;
        }
        return f[n];
    }

    public static void main(String[] args) {
        System.out.println(factorial(5));
        System.out.println(factorial(1));
        System.out.println(factorial(3));
        System.out.println(factorial(6));
        System.out.println(factorial(20));
    }
}
```

Приклад 2.3. Точка на площині [5]

Припустимо, необхідно реалізувати програму для роботи з точками на площині. Точка задається парою дійсних чисел. Клас для представлення точки повинен реалізовувати такі методи:

обчислення відстані від точки до початку координат;

обчислення відстані між двома точками.

Друга функція може бути реалізована як статичний метод. Програма може бути такою:

```
public class Point
{
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double distance() {
```

```
        return Math.sqrt(x * x + y * y);
    }
    public static double distance(Point p1, Point p2) {
        return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x) +
                         (p1.y - p2.y) * (p1.y - p2.y));
    }
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        System.out.println(p1.distance());
        Point p2 = new Point(4, 5);
        System.out.println(distance(p1, p2));
    }
}
```

2.3. Задачі [5]

Спроектувати та реалізувати два класи відповідно до індивідуального завдання. У першому з класів повинен бути описаний масив елементів другого класу. Класи повинні мати конструктори, приватні поля та відкриті методи, зокрема методи доступу (сеттери та геттери).

Слід окремо здійснити тестування кожного з класів, після чого окремими методами першого класу реалізувати основне завдання. У функції `main()` першого з класів створити необхідний об'єкт та викликати для нього методи, які реалізують основне завдання. Вивести результати у консольне вікно.

Варіант завдання, який слід реалізувати у програмі, визначається залежно від номеру студента у списку групи.

№№	Перший клас		Другий клас		Основне завдання: знайти та вивести такі дані
	Сутність	Обов'язкові поля	Сутність	Обов'язкові поля	
1	Погода	Сезон, коментар	День	Дата, температура, коментар	Середня температура, день з максимальною температурою, день з найдовшим коментарем
2	Навчальний курс	Назва, наявність іспиту	Практичне заняття	Дата, тема, кількість студентів	Середня кількість студентів, заняття з максимальною кількістю студентів, список тем з певним словом у назві
3	Трамвайна зупинка	Назва, список номерів маршрутів	Година	Кількість пасажирів, коментар	Загальна кількість пасажирів, година з найменшою кількістю пасажирів, найдовший коментар

4	Навчальний курс	Назва, прізвище викладача	Лекція	Дата, тема, кількість студентів	Лекція з мінімальною кількістю студентів, список тем з певним словом у назві, остання літера у прізвищі викладача
5	Погода	Рік, коментар	Вимір температури	Дата, температура, коментар	Виміри з мінімальною температурою, з найбільшою кількістю слів у коментарі до виміру, останнє слово коментаря до погоди
6	Конференція	Назва, місце проведення	Засідання	Дата, тема, кількість учасників	Середня кількість учасників на засіданні, засідання з найбільшою кількістю учасників, довжина назви
7	Виставка	Назва, прізвище художника	День	Кількість відвідувачів, коментар	Сумарна кількість відвідувачів, день з найменшою кількістю відвідувачів, список коментарів з певним словом
8	Станція метрополітену	Назва, рік відкриття	Година	Кількість пасажирів, коментар	Сумарна кількість пасажирів, години з найменшою кількістю пасажирів та найбільшою кількістю слів у коментарі
9	Лікар	Прізвище, фах	Прийом	День, зміна, кількість відвідувачів	Загальна кількість відвідувачів, прийом з мінімальною кількістю відвідувачів, довжина прізвища
10	Музичний гурт	Назва, прізвище керівника	Гастрольна поїздка	Місто, рік, кількість концертів	Гастрольна поїздка з максимальною кількістю концертів, список гастрольних поїздок у певне місто, остання літера в прізвищі керівника

ПРАКТИЧНА РОБОТА 3
Використання поліморфізму. Робота з узагальненнями та колекціями

Мета практичної роботи – опанувати написання програм узагальненнями та колекціями.

3.1. Теоретичні основи [4]

Механізм успадкування полягає в породженні похідних класів від базових. Якщо один клас (похідний) є нащадком іншого (базового), то спадкоємець має можливість безпосередньо користуватися неприватними даними і функціями, визначеними в базовому класі. Відносини між класами і підкласами (нащадками) називаються *ієрархією спадкування* класів.

На відміну від C++, у Java дозволяється тільки одиничне успадкування – клас може мати тільки один базовий клас. Успадкування завжди відкрите. У Java також немає захищеного і закритого успадкування. Успадкування має такий синтаксис:

```
class DerivedClass extends BaseClass
{
    // тіло класу
}
```

Функції похідного класу мають доступ тільки до елементів, описаних у як **`public`** і **`protected`**. Члени класу, оголошені як захищені, можуть використовуватися класами-нащадками, а також у межах пакета. Закриті (**`private`**) члени класу недоступні навіть для його нащадків.

Усі класи Java безпосередньо чи опосередненo походять від класу `java.lang.Object`. Цей клас надає набір корисних методів, таких як `toString()` для отримання даних будь-якого об’єкта у вигляді рядка тощо. Базовий клас `Object` не вказують явно.

Клас успадковує всі елементи базового класу, крім конструкторів. До початку виконання конструктора похідного класу викликається конструктор базового класу.

Ключове слово **`super`** використовують для доступу до елементів базового класу з похідного класу, зокрема:

– для виклику перекритого методу базового класу;

– для передачі параметрів конструктору базового класу.

Наприклад:

```
class BaseClass
{
    int i, j;
    BaseClass(int i, int j) {
        this.i = i;
        this.j = j;
    }
}
class DerivedClass extends BaseClass
{
    int k;
    DerivedClass(int i, int j, int k) {
        super(i, j);
        this.k = k;
    }
}
```

Доступ до базового класу з використанням **super** дозволений тільки в конструкторах і нестатичних методах.

Класи можуть бути визначені з модифікатором **final** (фінальний). Фінальні класи не можуть використовуватися як базові. Методи з модифікатором **final** не можуть бути перевизначені. Наприклад:

```
final class A
{
    void f() { }
}
class B
{
    final void g() { }
}
class C extends A
{ // Помилка! Не можна успадкувати від A
}
class D extends B
{
    void g() { }    // Помилка! g() не можна перекрити
}
```

Посилання на похідний клас неявно приводяться до посилання на базовий клас. Об’єкти похідних класів завжди можна використовувати там, де потрібен об’єкт базового класу.

```
class Base
{
    static void f(Base b) { }
}
class Derived extends Base
{
    public static void main(String[] args) {
```

```
        Base b;
        b = new Derived(); // Неявне приведення
        Derived d = new Derived();
        f(d);              // Неявне приведення
    }
}
```

Зворотне приведення необхідно робити явно:

```
Base b = new Base();
Derived d = (Derived) b;
```

Поліморфізм часу виконання – це властивість класів, згідно з якою поведінка об'єктів класу може визначатися не на етапі компіляції, а на етапі виконання. Класи, що надають ідентичний інтерфейс, але реалізовані під конкретні специфічні вимоги, мають назву *поліморфних класів*.

У мовах об'єктно-орієнтованого програмування пізнє зв'язування реалізоване через механізм віртуальних функцій. *Віртуальна функція* (віртуальний метод, `virtual method`) – це функція, визначена в базовому класі, та перевизначена (перекрита) у похідних, так, що конкретна реалізація функції для виклику визначатиметься під час виконання програми. Вибір реалізації віртуальної функції залежить від реального (а не оголошеного під час опису) типу об'єкта. Оскільки посилання на базовий тип може містити адресу об'єкта будь-якого похідного типу, поведінка раніше створених класів може бути змінена пізніше шляхом перевизначення віртуальних методів. Перевизначення передбачає відтворення імені, списку параметрів та специфікатора доступу. Фактично поліморфними є класи, які містять віртуальні функції.

У Java всі методи є віртуальними, за винятком конструкторів, статичних (`static`), фінальних (`final`) і закритих (`private`) методів. На відміну від C++, слово `virtual` не використовується.

Починаючи з Java 5, перед перевизначеними віртуальними методами розміщують директиву `@Override`, яка дозволяє компілятору здійснити додаткову перевірку синтаксису – відповідність сигнатури нової функції сигнатурі перекритої функції базового класу. Використання `@Override` є бажаним, але не обов'язковим.

Усі класи Java є поліморфними, оскільки таким є клас `java.lang.Object`. Зокрема, завдяки поліморфізму кожен клас може визначити свою віртуальну функцію `toString()`, яка буде викликана для автоматичного отримання даних про об'єкт у вигляді рядку.

Іноді класи створюються для представлення абстрактних

концепцій, а не для створення екземплярів. Такі концепції можуть бути представлені абстрактними класами. У Java для цього використовується ключове слово `abstract` перед визначенням класу

```
abstract class SomeConcept
{
    . . .
}
```

Абстрактний клас може містити абстрактні методи, такі, для яких не приводиться реалізація. Такі методи не мають тіла функції. Їхнє оголошення аналогічне оголошенню функцій-елементів у C++, але оголошенню повинне передувати ключове слово `abstract`.

Абстрактні методи аналогічні суто віртуальним функціям у C++.

Від абстрактного класу не вимагають обов'язкової наявності абстрактних методів. Але кожен клас, у якому є хоч один абстрактний метод, чи хоча б один абстрактний метод базового класу не був визначений, повинен бути оголошений як абстрактний.

У Java використовується поняття інтерфейсів. Інтерфейс може розглядатися як чисто абстрактний клас, але на відміну від абстрактних класів інтерфейс ніколи не містить даних, тільки методи. Ці методи усталено вважаються публічними:

```
interface SomeFunctions
{
    void f();
    int g(int x);
}
```

Кожен клас може бути створений тільки від одного базового класу, але при цьому реалізовувати один чи кілька інтерфейсів. Клас, що реалізує інтерфейс, повинен забезпечити реалізацію всіх методів, оголошених в інтерфейсі. В іншому випадку такий клас буде абстрактним і повинен бути оголошений зі специфікатором `abstract`.

Інтерфейси формально можуть містити поля, але вони є фінальними і статичними (константами часу компіляції). Вони повинні бути ініціалізовані під час створення. Інтерфейси не можуть містити конструкторів, оскільки немає ніяких даних окрім статичних констант.

Для того, щоб указати, що клас реалізує інтерфейс, ім'я інтерфейсу вказують у списку реалізованих інтерфейсів. Такий список розташовують у заголовку класу після ключового слова `implements`. Методи, визначені в інтерфейсі, є абстрактними і відкритими. У класі, що реалізує інтерфейс, такі методи повинні бути оголошені як `public`:

```
interface SomeFunctions
```

```
{
    void f();
    int g(int x);
}
class SomeClass implements SomeFunctions
{
    @Override
    public void f() {
    }
    @Override
    public int g(int x) {
        return x;
    }
}
```

Інтерфейс може мати кілька базових інтерфейсів. Множинне успадкування інтерфейсів є безпечним з точки зору дублювання даних і конфліктів імен.

Клас може реалізувати кілька інтерфейсів. Це – більш розповсюджений шлях, ніж створення похідного інтерфейсу:

Дуже часто в програмі створюють посилання на інтерфейс, яке ініціалізують об’єктом класу. Такий підхід є належною практикою, оскільки він дозволяє легко замінити одну реалізацію інтерфейсу іншою.

Інтерфейси не походять від класу `java.lang.Object`. Не можна створювати новий об’єкт типу інтерфейсу. Навіть для порожнього інтерфейсу треба створити клас, який його реалізує. JDK надає велику кількість стандартних інтерфейсів.

Парадигма *узагальненого програмування* передбачає опис правил зберігання даних і алгоритмів у загальному вигляді незалежно від конкретних типів даних. Конкретні типи даних, над якими виконуються дії, специфікуються пізніше. Механізми розділення структур даних і алгоритмів, а також формування абстрактних описів вимог до даних, визначаються по-різному в різних мовах програмування.

Для реалізації узагальненого програмування в Java використовуються узагальнення – спеціальна мовна конструкція, яка з’явилася у синтаксисі мови починаючи з версії Java 5.

Узагальнення – конструкція, що включає в себе параметр класу або функції, який містить додаткову інформацію про тип елементів та інших даних. Цей параметр беруть у кутові дужки. Узагальнення надають можливість створення та використання структур даних, безпечних з точки зору типів. Класи, опис яких містить такий параметр,

мають назву *узагальнених*. Під час створення об’єкта узагальненого типу у кутових дужках вказують імена реальних типів. Можна використовувати тільки типи-посилання.

```
public class Pair<T>
{
  T first, second;
  public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
  public static void main(String[] args) {
   Pair<String> p = new Pair<String>("Прізвище", "Ім\’я");
   String s = p.first; // Отримуємо рядок без приведення типів
   Pair<Integer> p1 = new Pair<Integer>(1, 2);
   int i = p1.second;
    }
}
```

Примітка: Java версії 7 і вище дозволяє не повторювати фактичний параметр узагальнення після імені конструктора. Наприклад:

```
Pair<Integer> p1 = new Pair<>(1, 2);
```

Якщо ми намагаємось додати до пари дані різних типів, компілятор згенерує помилку. Помилковою є також спроба явно перетворити тип:

```
Pair<String> p = new Pair<String>("1", "2");
Integer i = (Integer) p.second; // Помилка компіляції
```

Тип даних з параметром у кутових дужках (наприклад, `Pair<String>`) має назву *параметризованого типу*.

Узагальнення по зовнішньому представленню і використанню аналогічні шаблонам C++. Але на відміну від шаблонів C++, існує не декілька різних типів `Pair`, а один. Фактично у полях класу зберігаються посилання на `Object`. Інформація про тип параметрів використовується компілятором для контролю та автоматичного приведення типів у вихідному тексті.

Окрім узагальнених класів, можна створювати *узагальнені інтерфейси*. Параметр може бути використаний в описі функцій, оголошених в інтерфейсі. Під час їх реалізації замість параметра узагальнення використовують деякий тип-посилання. Наприклад:

```
interface Function<T>
{
    T func(T x);
}
class DoubleFunc implements Function<Double>
{
```

```
    @Override
    public Double func(Double x) {
        return x * 1.5;
    }
}
class IntFunc implements Function<Integer>
{
    @Override
    public Integer func(Integer x) {
        return x % 2;
    }
}
```

Java також дозволяє створювати *узагальнені функції* всередині як узагальнених, так і звичайних (неузагальнених) класів:

```
public class ArrayPrinter
{
    public static<T> void printArray(T[] a) {
        for (T x : a) {
            System.out.print(x + "\t");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        String[] as = {"First", "Second", "Third"};
        printArray(as);
        Integer[] ai = {1, 2, 4, 8};
        printArray(ai);
    }
}
```

Як видно з прикладу, виклик узагальненої функції не вимагає явного визначення типу. Іноді таке визначення необхідне, наприклад, коли в функції немає параметрів узагальненого типу. Якщо це статична функція, необхідно явно вказувати її клас. Наприклад:

```
public class TypeConverter
{
    public static <T>T convert(Object object) {
        return (T) object;
    }
    public static void main(String[] args) {
        Object o = "Some Text";
        String s = TypeConverter.<String>convert(o);
        System.out.println(s);
    }
}
```

Рекомендованими іменами формальних параметрів є імена з однієї великої літери. Узагальнення може мати два і більше параметрів.

Над даними типу параметру узагальнення можна здійснювати

тільки дії, дозволені для об’єктів класу `Object`. Іноді для розширення функціональності бажаною є конкретизація типу. Наприклад, ми хочемо викликати методи, оголошені у певному класі або інтерфейсі. Тоді можна застосувати такий синтаксис опису параметру: `<T extends SomeBaseType>` або `<T extends FirstType & SecondType>` тощо. Слово `extends` використовують як для класів, так і для інтерфейсів.

Синтаксис узагальнень передбачає використання так званих масок (wildcard, символ ‘?’). Маска застосовується, наприклад, для опису посилань на поки невідомий тип. Використання масок робить узагальнені класи та функції більш сумісними. Маска надає альтернативний спосіб створення узагальнених функцій. Такі функції самі по собі не є узагальненими, але містять аргументи узагальнених типів.

3.2. Приклади

Приклад 3.1. Ієрархія об'єктів реального світу [5]

Припустимо, необхідно розробити ієрархію класів "Регіон" – "Населений район" – "Країна". Окремі класи цієї ієрархії можуть стати базовими для інших класів (наприклад "Незаселений острів", "Національний парк", "Адміністративний район", "Автономна республіка" і т.д.). Ієрархію класів можна доповнити класами "Місто" і "Острів". Доцільно в кожен клас додати конструктор, який ініціалізує усі поля. Можна також створити масив посилань на різні об'єкти ієрархії і для кожного об'єкта вивести на екран рядок даних про нього.

Для того, щоб одержати рядкове представлення об'єкта, необхідно перекрити функцію `toString()`.

Можна запропонувати таку ієрархію класів.

```
import java.util.*;

// Ієрархія класів
class Region
{
    private String name;
    private double area;

    public Region(String name, double area) {
        this.name = name;
        this.area = area;
    }

    public String getName() {
        return name;
    }
```

```
    public double getArea() {
        return area;
    }

    public String toString() {
        return "Регіон " + name + ".\tТериторія   " + area + " кв.км.";
    }

}

class PopulatedRegion extends Region
{
    private int population;

    public PopulatedRegion(String name, double area, int population) {
        super(name, area);
        this.population = population;
    }

    public int getPopulation() {
        return population;
    }

    public int density() {
        return (int) (population / getArea());
    }

    public String toString() {
        return "Населений регіон " + getName() + ".\tТериторія    " +
getArea() +
                " кв.км.   \tНаселення " + population +
                "   чол.\tЩільність   населення   "   +   density()   +   "
чол/кв.км.";
    }

}

class Country extends PopulatedRegion
{
    private String capital;

    public Country(String name, double area, int population, String
capital) {
        super(name, area, population);
        this.capital = capital;
    }

    public String getCapital() {
        return capital;
    }
```

```
    public String toString() {
        return "Країна " + getName() + ".\tТериторія   " + getArea() +
               " кв.км.   \tНаселення " + getPopulation() +
               " чол.\tЩільність населення " + density() +
               " чол/кв.км.\tСтолиця " + capital;
    }

}

class City extends PopulatedRegion
{
    private int boroughs; // Кількість районів

    public City(String name, double area, int population, int boroughs)
{
        super(name, area, population);
        this.boroughs = boroughs;
    }

    public int getBoroughs() {
        return boroughs;
    }

    public String toString() {
        return "Місто " + getName() + ".\tТериторія   " + getArea() +
               " кв.км.   \tНаселення " + getPopulation() +
               " чол.\tЩільність населення " + density() +
               " чол/кв.км.\tРайонів – " + boroughs;
    }

}

class Island extends PopulatedRegion
{
    private String sea;

    public Island(String name, double area, int population, String sea)
{
        super(name, area, population);
        this.sea = sea;
    }

    public String getSea() {
        return sea;
    }

    public String toString() {
        return "Острів " + getName() + ".\tТериторія   " + getArea() +
               " кв.км.   \tНаселення " + getPopulation() +
               " чол.\tЩільність населення " + density() +
```

```
                " чол/кв.км.\tМоре – " + sea;
    }
}

public class Regions
{

    public static void main(String[] args) {
        Region[] a = { new City("Київ", 839, 2679000, 10),
                       new    Country("Україна",    603700,    46294000,
"Київ"),
                       new City("Харків", 310, 1461000, 9),
                       new Island("Зміїний", 0.2, 30, "Чорне") };
        for (Region region : a) {
            System.out.println(region);
        }
  }
```

Приклад 3.2. Клас для представлення масиву точок [5]

3.2.1 Постановка завдання і створення абстрактного класу

Припустимо, необхідно розробити клас для представлення масиву точок. Кожна точка представлена двома числами типу `double` – x і y. Необхідно забезпечити завдання точки, отримання інформації про координати конкретної точки та загальну кількість точок, а також додавання точки в кінець масиву і видалення останньої точки. Крім того, необхідно організувати сортування масиву за зростанням заданої координати і виведення координат точок у рядок.

Найбільш простим, але не єдиним рішенням є створення класу Point з двома полями та створення масиву посилань на Point. Таке рішення – правильне з точки зору організації структури даних, але не достатньо ефективне, оскільки воно припускає розміщення в динамічної пам'яті як самого масиву, так і окремих об'єктів-точок. Альтернативні варіанти – використання двох масивів, двовимірного масиву тощо.

Остаточне рішення про структуру даних може бути прийнято тільки в контексті конкретного завдання. Поліморфізм дозволяє реалізувати необхідні алгоритми без прив'язування до конкретної структури даних. Для цього створюємо абстрактний клас, в якому функції доступу оголошені як абстрактні, а алгоритми сортування й виведення в рядок реалізовані з використанням абстрактних функцій доступу. Крім того можна визначити функцію для тестування. Для того щоб був згенерований абстрактний клас, у вікні майстра нового класу обираємо опцію `abstract`. Додаємо до шаблону необхідний код. Відповідний абстрактний клас буде таким:

```
public abstract class AbstractArrayOfPoints
{
    // Запис нових координат точки:
    public abstract void setPoint(int i, double x, double y);
```

```
// Отримання X точки i:
public abstract double getX(int i);

// Отримання Y точки i:
public abstract double getY(int i);

// Отримання кількості точок:
public abstract int count();

// Додавання точки в кінець масиву:
public abstract void addPoint(double x, double y);

// Видалення останньої точки:
public abstract void removeLast();

// Сортування за значеннями X:
public void sortByX() {
    boolean mustSort; // Повторюємо доти,
                      // доки mustSort дорівнює true
    do {
        mustSort = false;
        for (int i = 0; i < count() - 1; i++) {
            if (getX(i) > getX(i + 1)) {
                // обмінюємо елементи місцями
                double x = getX(i);
                double y = getY(i);
                setPoint(i, getX(i + 1), getY(i + 1));
                setPoint(i + 1, x, y);
                mustSort = true;
            }
        }
    }
    while (mustSort);
}

// Аналогічно можна реалізувати функцію sortByY()

// Виведення точок у рядок:
@Override
public String toString() {
    String s = "";
    for (int i = 0; i < count(); i++) {
        s += "x = " + getX(i) + " \ty = " + getY(i) + "\n";
    }
    return s + "\n";
}

// Тестуємо сортування на чотирьох точках:
public void test() {
    addPoint(22, 45);
```

```
        addPoint(4, 11);
        addPoint(30, 5.5);
        addPoint(-2, 48);
        sortByX();
        System.out.println(this);
    }
}
```

Тепер можна реалізувати різні варіанти представлення структури даних.

3.2.2 Реалізація через масив об'єктів типу Point [5]

Першою з можливих реалізацій буде створення класу `Point` та використання масиву посилань на `Point`. У тому ж проекті створюємо клас ArrayOfPointObjects. У вікні New Java Class вибираємо опції `public` і `abstract`, у рядку `Superclass` уводимо `AbstractArrayOfPoints`. Крім того, доцільно вибрати опції `public static void main(String[] args)` і `Inherited abstract methods`. Отримаємо такий код:

```
public class ArrayOfPointObjects extends AbstractArrayOfPoints
{

    @Override
    public void setPoint(int i, double x, double y) {
        // TODO Auto-generated method stub
    }

    @Override
    public double getX(int i) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double getY(int i) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public int count() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void addPoint(double x, double y) {
        // TODO Auto-generated method stub
    }

    @Override
    public void removeLast() {
```

```
        // TODO Auto-generated method stub
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Клас для представлення точки можна додати в той же пакет. Клас Point міститиме два поля і конструктор:

```
public class Point
{
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

У класі `ArrayOfPointObjects` створюємо поле – посилання на масив Point та ініціалізуємо його порожнім масивом. Реалізація більшості функцій видається очевидною. Найбільшу складність являють функції додавання та видалення точок. В обох випадках необхідно створити новий масив потрібної довжини й переписати в нього вміст старого. У функції `main()` здійснюємо тестування. Весь код файлу `AbstractArrayOfPoints.java` матиме такий вигляд:

```
public class ArrayOfPointObjects extends AbstractArrayOfPoints
{
    private Point[] p = { };

    @Override
    public void setPoint(int i, double x, double y) {
        if (i < count()) {
            p[i].setPoint(x, y);
        }
    }

    @Override
```

```
    public double getX(int i) {
        return p[i].getX();
    }

    @Override
    public double getY(int i) {
        return p[i].getY();
    }

    @Override
    public int count() {
        return p.length;
    }

    @Override
    public void addPoint(double x, double y) {
        // Створюємо масив, більший на один елемент:
        Point[] p1 = new Point[p.length + 1];
        // Копіюємо всі елементи:
        System.arraycopy(p, 0, p1, 0, p.length);
        // Записуємо нову точку в останній елемент:
        p1[p.length] = new Point(x, y);
        p = p1; // Тепер p вказує на новий масив
    }

    @Override
    public void removeLast() {
        if (p.length == 0) {
            return; // Масив уже порожній
        }
        // Створюємо масив, менший на один елемент:
        Point[] p1 = new Point[p.length - 1];
        // Копіюємо всі елементи, крім останнього:
        System.arraycopy(p, 0, p1, 0, p1.length);
        p = p1; // Тепер p вказує на новий масив
    }

    public static void main(String[] args) {
        // Можна створити безіменний об'єкт:
        new ArrayOfPointObjects().test();
    }
}
```

У результаті отримаємо в консольному вікні точки, розсортовані за координатою X.

3.2.3 Реалізація через два масиви [5]

Альтернативна реалізація передбачає створення двох масивів для окремого зберігання значень `X` і `Y`. Створюємо клас `ArrayWithTwoArrays` із використанням аналогічних опцій. У класі `ArrayWithTwoArrays` створюємо два поля – посилання на масиви дійсних

чисел і ініціалізуємо їх порожніми масивами. Реалізація функцій аналогічна попередньому варіанту. У функції `main()` здійснюємо тестування:

```
public class ArrayWithTwoArrays extends AbstractArrayOfPoints
{
    private double[] ax = { };
    private double[] ay = { };

    @Override
    public void setPoint(int i, double x, double y) {
        if (i < count()) {
            ax[i] = x;
            ay[i] = y;
        }
    }

    @Override
    public double getX(int i) {
        return ax[i];
    }

    @Override
    public double getY(int i) {
        return ay[i];
    }

    @Override
    public int count() {
        return ax.length; // Можна ay.length, вони однакові
    }

    @Override
    public void addPoint(double x, double y) {
        double[] ax1 = new double[ax.length + 1];
        System.arraycopy(ax, 0, ax1, 0, ax.length);
        ax1[ax.length] = x;
        ax = ax1;
        double[] ay1 = new double[ay.length + 1];
        System.arraycopy(ay, 0, ay1, 0, ay.length);
        ay1[ay.length] = y;
        ay = ay1;
    }

    @Override
    public void removeLast() {
        if (count() == 0) {
            return;
        }
        double[] ax1 = new double[ax.length - 1];
        System.arraycopy(ax, 0, ax1, 0, ax1.length);
```

```
        ax = ax1;
        double[] ay1 = new double[ay.length - 1];
        System.arraycopy(ay, 0, ay1, 0, ay1.length);
        ay = ay1;
    }

    public static void main(String[] args) {
        new ArrayWithTwoArrays().test();
    }
}
```

Результати мають бути ідентичними.

3.2. Задачі [5]

Розробити ієрархію класів для представлення сутностей індивідуального завдання попередньої лабораторонї роботи. Базовий абстрактний клас, який представляє другу сутність індивідуального завдання, не повинен містити даних, лише абстрактні методи доступу, перевизначення функцій `toString()` та `equals()`, а також реалізацію функцій, визначених попереднім завданням.

Базовий абстрактний клас, який представляє першу з сутностей індивідуального завдання, повинен містити:

- абстрактні функції для доступу до даних;
- абстрактні функції для доступу до послідовності елементів типу другого абстрактного класу;
- абстрактні функції сортування елементів послідовності за визначеними ознаками відповідно до індивідуального завдання;
- перевизначення функції `toString()` для виведення даних про об'єкти;
- перевизначення методу `equals()` для перевірки еквівалентності об'єктів;
- реалізацію методів пошуку за визначеними ознаками;
- реалізацію функції додавання об'єкта з перевіркою, чи такий елемент вже присутній;
- реалізацію методу тестування функціональності класів.

Похідні класи від створених абстрактних класів повинні містити поля конкретних типів, зокрема, послідовність елементів другої сутності може бути представлена у різних похідних класах або у вигляді масиву, або списку.

ПРАКТИЧНА РОБОТА 4
Робота з винятками і файлами

Мета практичної роботи – опанувати написання програм з використанням блоків з винятками та файлами.

4.1. Теоретичні основи [4]

Використання механізму обробки винятків є дуже важливою складовою частиною практики програмування на Java. Майже кожна програма на Java містить певні частини цього механізму. Об'єкти-винятки дозволяють програмісту відокремити точки виникнення помилок часу виконання від коду, який ці помилки повинен обробити. Це дозволяє створювати більш надійно працюючі універсальні класи і бібліотеки.

Виняток – це подія, що виникає під час виконання програми і порушує нормальне виконання інструкцій коду. *Механізм генерації та обробки винятків* дозволяє передати інформацію про помилку з місця виникнення у місце, де ця помилка може бути оброблена. Винятки в Java поділяють на синхронні (помилка часу виконання, ситуація, згенерована за допомогою `throw`) і асинхронні (системні, збої віртуальної машини Java). Місце виникнення другої групи винятків виявити досить складно.

Механізм винятків присутній в усіх сучасних мовах об'єктно-орієнтованого програмування. У порівнянні з C++, Java реалізує більш строгий механізм роботи з винятками.

Для генерації винятку використовується оператор `throw`. Після ключового слова `throw` міститься об'єкт классу `java.lang.Throwable`, або класів, похідних від нього. Для програмних винятків найчастіше використовується класс `java.lang.Exception` (похідний від `Throwable`). Використання `Exception` замість `Throwable` дозволяє відокремити власний виняток від системних помилок. Найкраща практика керування винятками – створювати класи, похідні від `Exception`. Такі похідні класи зазвичай відбивають специфіку конкретної програми.

```
class SpecificException extends Exception { }
```

Є також базовий клас для генерації системних помилок – клас `Error`. Класи `Exception` і `Error` мають загальний базовий клас – `Throwable`. Виняток генерується шляхом використання ключового слова **`throw`**, за яким розташовують об'єкт-виняток. У більшості випадків об'єкт-виняток

створюється в точці генерації винятку за допомогою оператора new. Наприклад, типове твердження throw може виглядати так:

```
void f() . . .
    . . .
    if (/* помилка */) {
        throw new SpecificException();
    }
```

У заголовку функції необхідно перелічити усі типи винятків, які генерує ця функція. Це слід зробити за допомогою ключового слова throws:

```
void f() throws SpecificException, AnotherException {
    . . .
    if (/* помилка */) {
        throw new SpecificException();
    }
    if (/* інша помилка */) {
        throw new AnotherException();
    }
    . . .
}
```

У наведеному нижче прикладі функція reciprocal() генерує виняток у випадку ділення на нуль.

```
class DivisionByZero extends Exception { }
class Test
{
    double reciprocal(double x) throws DivisionByZero {
        if (x == 0) {
            throw new DivisionByZero();
        }
        return 1 / x;
    }
}
```

На відміну від C++, Java не допускає створення винятків примітивних типів. Дозволені тільки об'єкти, похідні від Throwable або Exception.

Під час успадкування для перевизначених функцій список винятків повинен зберігатися.

Виняток, який був згенерований у певній частині коду, повинен бути перехоплений в іншій частині. Наприклад, якщо ми хочемо звернутися до функції, яка потенційно може згенерувати виняток, виклик цієї функції поміщають у блок try { }.

Після блоку try повинен міститись один чи декілька оброблювачів (блоків catch). Кожен такий оброблювач відповідає визначеному типу винятку:

```
catch (DivisionByZero d) {
    // обробка винятку
}
catch (Exception ex)  {
    // обробка винятку
}
```

Класи винятків утворюють ієрархію. Під час порівняння типів винятків оброблювач базового типу сприймає також винятки всіх створених від нього типів. Звідси випливає, що оброблювачі похідних типів варто розміщати до оброблювачів базових типів. Припустимо, є така ієрархія класів винятків:

```
class BaseException extends Exception { }
class FileException extends BaseException { }
class FileNotFoundException extends FileException { }
class WrongFormatException extends FileException { }
class MathException extends BaseException { }
class DivisionByZero extends MathException { }
class WrongArgument extends MathException { }
```

Залежно від логіки програми різні типи винятків можна обробляти більш детально:

```
try {
    Exceptions.badFunc();
}
catch (FileNotFoundException ex) {
    // файл не знайдено
}
catch (WrongFormatException ex) {
    // хибний формат
}
catch (FileException ex) {
    // інші помилки, пов’язані з файлами
}
catch (MathException ex) {
    // усі математичні помилки обробляємо разом
}
catch (BaseException ex) {
    // підбираємо всі інші винятки функції badFunc()
}
catch (Exception ex) {
    // про всяк випадок
}
```

Після останнього блоку `catch` можна розмістити блок `finally`. Цей код завжди виконується незалежно від того, виник чи не виник виняток, навіть якщо в якомусь з блоків був здійснений вихід з функції.

На відміну від C++, не можна використовувати `catch (...)` для перехоплення будь-якого винятку. Замість цього можна

використовувати перехоплення винятків базових класів:

```
catch (Exception ex)
{
    // обробка винятку
}
```

або

```
catch (Throwable ex)
{
    // обробка винятку
}
```

Якщо в межах блоку `catch () { }` не можна повністю обробити виняток, його можна передати далі:

```
catch (SomeException ex)
{
    // локальна обробка винятку
    throw ex;
}
```

4.2. Приклади

Приклад 4.1. Порядкове копіювання текстових файлів [4]

Припустимо, необхідно створити програму, яка здійснює копіювання текстових файлів рядок за рядком. Імена файлів задаються аргументами командного рядка. Текст програми буде таким:

```
import java.io.*;

public class TextFileCopy
{
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Необхідні аргументи!");
            return;
        }
        try    (BufferedReader    in    =    new    BufferedReader(new
FileReader(args[0]));
             PrintWriter out = new PrintWriter(new FileWriter(args[1])))
{
            String line;
            while ((line = in.readLine()) != null) {
                out.println(line);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Приклад 4.2. Сортування дійсних чисел [5]

Припустимо, необхідно реалізувати програму читання з текстового

файлу дійсних значень у діапазоні від -1000 до 1000, сортування за збільшенням і за зменшенням модулів та зберігання обох результатів у двох нових текстових файлах. Числа у вихідному файлі розділені пробілами, їх слід читати до кінця файлу.

У класі `DoubleNumbers`, який ми проектуємо, створюємо вкладений статичний клас для опису винятку, пов'язаного з хибним дійсним значенням (менше, ніж -1000 або більше, ніж 1000). Крім того, під час роботи функції `sortDoubles()`, яка виконує основне завдання, можуть виникати винятки типу `IOException` (файл не знайдено, файл не можна створити тощо) та `InputMismatchException` (об'єкт типу `Scanner` намагається отримати `Double` з лексеми, яка не може бути переведена у число). Для сортування за зменшенням модулів створюємо окрему статичну функцію `comareByAbsValues()`, у якій створюється локальний клас та повертається його об'єкт. Вихідний код матиме такий вигляд:

```
import java.io.*;
import java.util.*;
import static java.lang.Math.*;

public class DoubleNumbers
{
    /**
     * Внутрішній клас-виняток, який дозволяє зберігати хибне дійсне
     * значення, прочитане з файлу (менше, ніж -1000 або більше, ніж
1000)
     *
     */
    public static class DoubleValueException extends Exception {
        private double wrongValue;

        public DoubleValueException(double wrongValue) {
            this.wrongValue = wrongValue;
        }

        public double getWrongValue() {
            return wrongValue;
        }

    }

    /**
     * Статична функція, яка визначає метод порівняння дійсних
     * чисел під час сортування за зменшенням абсолютної величини
     *
     * @return об'єкт, який реалізує інтерфейс Comparator
     *
     */
```

```
    public static Comparator<Double> comareByAbsValues() {
        // Локальний клас:
        class LocalComparator implements Comparator<Double> {
            @Override
            public int compare(Double d1, Double d2) {
                return -Double.compare(abs(d1), abs(d2));
            }
        }
        return new LocalComparator();
    }

    /**
     * Функція здійснює читання дійсних чисел у діапазоні від -1000 до
1000, сортування
     * за двома ознаками та занесення у два результуючі файли
     *
     * @param inFileName - ім'я вихідного файлу
     * @param firstOutFileName - ім'я файлу, який міститиме числа,
відсортовані
     * за зростанням
     * @param secondOutFileName - ім'я файлу, який міститиме числа,
відсортовані
     * за зменшенням абсолютних величин
     * @throws DoubleValueException
     * @throws IOException
     * @throws InputMismatchException
     */
    public static void sortDoubles(String inFileName, String
firstOutFileName,
            String secondOutFileName) throws DoubleValueException,
IOException,
            InputMismatchException {
        Double[] arr = {};
        try (BufferedReader reader = new BufferedReader(new
FileReader(inFileName));
             Scanner scanner = new Scanner(reader)) {
            while (scanner.hasNext()) {
                double d = scanner.nextDouble();
                if (abs(d) > 1000) {
                    throw new DoubleValueException(d);
                }
                Double[] arr1 = new Double[arr.length + 1];
                System.arraycopy(arr, 0, arr1, 0, arr.length);
                arr1[arr.length] = d;
                arr = arr1;
            }
        }
        PrintWriter firstWriter = new PrintWriter(new
FileWriter(firstOutFileName));
        PrintWriter secondWriter = new PrintWriter(new
FileWriter(secondOutFileName));
```

```
            try {
                Arrays.sort(arr);
                for (Double x : arr)
                    firstWriter.print(x + " ");
                Arrays.sort(arr, comareByAbsValues());
                for (Double x : arr)
                    secondWriter.print(x + " ");
            }
            // Результуючі файли доцільно закрити у блоці finally:
            finally {
                firstWriter.close();
                secondWriter.close();
            }
        }

    public static void main(String[] args) {
            try {
                sortDoubles("in.txt", "out1.txt", "out2.txt");
            }
            // Неправильне дійсне значення:
            catch (DoubleValueException e) {
                e.printStackTrace();
                System.err.println("Wrong value: " + e.getWrongValue());
            }
            // Помилка, пов'язана з файлами:
            catch (IOException e) {
                e.printStackTrace();
            }
            // Файл містить щось, що не є дійсним числом:
            catch (InputMismatchException e) {
                e.printStackTrace();
            }
        }
}
```

Функція **hasNext()** повертає **true**, якщо за допомогою об'єкта типу **Scanner** можна прочитати наступне значення.

4.3. Задачі [5]

Спроектувати та реалізувати класи для представлення сутностей попередньої лабораторної роботи. Рішення повинне базуватися на раніше створеній ієрархії класів. Слід створити два похідних класи від класу, який представляє основну сутність. Один клас повинен бути доповненим можливостями читання даних з відповідно підготовленого текстового файлу та запису цих даних в інший файл після сортування.

Окрім роботи з файлами повинно бути реалізоване виведення результатів у консольне вікно.

ПРАКТИЧНА РОБОТА 5
Створення програм графічного інтерфейсу користувача

Мета практичної роботи – опанувати написання програм з графічним інтерфейсом користувача.

5.1. Теоретичні основи [4]

Інтерфейс користувача – це набір технічних та програмних засобів, за допомогою яких людина взаємодіє з комп'ютером. Далі йтиметься про програмні засоби інтерфейсу комп'ютеру.

Інтерфейс командного рядку – це метод взаємодії з програмою за допомогою інтерпретатору команд, які користувач уводить, як правило, у текстовому режимі, або у спеціальному консольному вікні. *Графічний інтерфейс користувача* (Graphical user interface, GUI) дає можливість користувачеві взаємодіяти з комп'ютером за допомогою графічних елементів управляння (вікон, піктограм, меню, кнопок, списків тощо) та технічних пристроїв позиціонування, таких як маніпулятор "миша" Програми, які реалізують цей тип інтерфейсу, мають назву *застосунків графічного інтерфейсу користувача*.

Реалізація застосунків графічного інтерфейсу користувача базується на механізмі отримання та обробки подій. Уся програма складається з ініціалізації (реєстрації візуальних елементів управління) та основного циклу отримання та обробки подій. Події – це переміщення або натискання кнопок миші, клавіатурне введення, тощо. Кожний зареєстрований візуальний елемент управління може отримувати події, які до нього стосуються, та виконувати функції обробки цих подій.

Наразі стандартними засобами розробки додатків графічного інтерфейсу користувача в Java є бібліотеки AWT і Swing, а також платформа JavaFX, засоби якої є альтернативою Swing. Крім того, різні розробники надають альтернативні нестандартні бібліотеки, такі як Qt Jambi, Standard Window Toolkit (SWT), XML Window Toolkit (XWT). Дві останні бібліотеки, поряд з AWT і Swing, підтримуються Eclipse.

Бібліотека `javax.swing` пропонує розробникові низку стандартних класів, які можна використовувати для проектування графічного інтерфейсу користувача. Ця бібліотека розширила попередню менш вдалу бібліотеку AWT (Abstract Window Toolkit) засоби якої використовує для обробки подій, роботи з графікою тощо. Для

ідентифікації приналежності до бібліотеки `javax.swing` до імен класів візуальних компонентів додана літера `J` (наприклад, `JButton`, `JPanel` і т.д.).

На відміну від компонентів AWT, компоненти Swing є "легковагими" (lightweight). Це означає, що компоненти Swing використовують засоби Java для відображення елементів графічного інтерфейсу користувача на поверхні вікна, без використання компонентів операційної системи.

Як і більшість бібліотек графічного інтерфейсу користувача, бібліотека `javax.swing` підтримує концепцію головного вікна застосунку. Це головне вікно створюється як об'єкт класу `JFrame`, або похідного від нього. Далі до головного вікна додають візуальні компоненти – мітки (`JLabel`), кнопки (`JButton`), рядки введення (`JTextField`) тощо.

Для того, щоб створити найпростішу програму графічного інтерфейсу користувача, необхідно створити новий клас з функцією `main()`, а потім у вихідному тексті додати твердження `import`:

```
import javax.swing.*;
```

У функції `main()` створюємо нове вікно та вказуємо його заголовок:

```
public class HelloWorldSwing {
    public static void main(String[] args) {
       JFrame frame = new JFrame("Привіт");
       . . .
    }
}
```

Далі додаємо нову мітку до компоненту, який відповідає за вміст вікна. Створюємо новий об'єкт типу `JLabel`:

```
frame.getContentPane().add(new JLabel("Привіт, світ!"));
```

Візуальні компоненти бібліотеки Swing успадковуються від классу `javax.swing.JComponent`, спадкоємця класу `java.awt.Container`. У свою чергу, цей клас є спадкоємцем `java.awt.Component`. Клас `java.awt.Component` – базовий клас, який визначає відображення на екрані і поведінку кожного елемента інтерфейсу під час взаємодії з користувачем. Методи класу, що відповідають за управління подіями, дозволяють задати розмір, колір, шрифт та інші атрибути елементів управління.

5.2. Приклади

Приклад 5.1. [5] Припустимо, необхідно створити програму графічного інтерфейсу користувача, у якій у двох рядках уведення

користувач задає два цілих числа і після натискання кнопки одержує в третьому рядку введення суму цих чисел.

Кореневим контейнером нашого застосунку буде `FlowPane`. Отже, досить створити три текстових поля й одну кнопку і послідовно додати їх до панелі. Одержимо таку програму:

```
import javafx.application.Application;
import javafx.event.Event;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class TextFieldsAndButton extends Application
{
    private Button button;
    private TextField field1, field2, field3;

    @Override
    public void start(Stage stage) throws Exception {
        stage.setTitle("Сума");
        FlowPane rootNode = new FlowPane(10, 10);// визначаємо розміри
горизонтального і
                                                 //       вертикального
зазорів між елементами
        rootNode.setAlignment(Pos.CENTER);
        Scene scene = new Scene(rootNode, 200, 200); // розміри вікна
        stage.setScene(scene);
        button = new Button("Знайти суму");    // визначаємо напис на
кнопці
        button.setOnAction(this::buttonClick);//  визначаємо  функцію,
яка обробляє подію
        field1 = new TextField();
        field2 = new TextField();
        field3 = new TextField();
        rootNode.getChildren().addAll(field1, field2, button, field3);
        stage.show();
    }

    private void buttonClick(Event event) {
        try {
            int i = Integer.parseInt(field1.getText());
            int j = Integer.parseInt(field2.getText());
            int k = i + j;
            field3.setText(k + "");
        }
        catch (NumberFormatException e1) {
```

```
            Alert alert = new Alert(AlertType.ERROR);
            alert.setTitle("Помилка");
            alert.setHeaderText("Хибні дані!");
            alert.showAndWait();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Приклад 5.2. Робота з кнопками RadioButton [5]

У наведеному нижче прикладі одночасно з вибором кнопки RadioButton у мітці (Label) відображається текст вибраної кнопки. Для того, щоб робота кнопок була узгодженою, їх об'єднують у групу ToggleGroup:

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ToggleGroupDemo extends Application
{
    private Label label = new Label("No button");

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Toggle Group Demo");

        RadioButton radioButtonFirst = new RadioButton("First");
        radioButtonFirst.setOnAction(this::showButtonText);
        RadioButton radioButtonSecond = new RadioButton("Second");
        radioButtonSecond.setOnAction(this::showButtonText);
        RadioButton radioButtonThird = new RadioButton("Third");
        radioButtonThird.setOnAction(this::showButtonText);

        ToggleGroup radioGroup = new ToggleGroup();
        radioButtonFirst.setToggleGroup(radioGroup);
        radioButtonSecond.setToggleGroup(radioGroup);
        radioButtonThird.setToggleGroup(radioGroup);
        VBox vbox = new VBox(radioButtonFirst, radioButtonSecond,
radioButtonThird, label);
        vbox.setSpacing(10);
```

```
        vbox.setPadding(new Insets(10, 10, 10, 10));
        Scene scene = new Scene(vbox, 150, 120);
        primaryStage.setScene(scene);
        primaryStage.show();

    }

    private void showButtonText(ActionEvent actionEvent) {

label.setText(((RadioButton)actionEvent.getSource()).getText());
    }

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

5.3. Задачі [5]

Необхідно реалізувати мовою Java за допомогою засобів JavaFX застосунок графічного інтерфейсу користувача, в якому здійснюється обробка даних індивідуальних завдань попередніх лабораторних робіт. Головне вікно повинно містити меню, в якому необхідно реалізувати такі функції:

- створення нового набору даних
- завантаження даних з документу для редагування
- зберігання змінених даних в документі
- пошук за ознаками, визначеними в попередній лабораторній роботі
- отримання вікна "Про програму" з даними про програму і автора.

ВИМОГИ ДО ЗВІТУ З ПРАКТИЧНОЇ РОБОТИ

Виконаний звіт з практичної роботи може бути подано у вигляді текстового файлу у форматі *.doc (.docx)* і прикріплено до відповідної активності в системі Moodle у форматі *.pdf*.

Практична робота виконується українською мовою з дотриманням норм наукового стилю, який передбачає:

- формально-логічний спосіб подання матеріалу, аргументовані міркування, що сприяють доведенню істинності положень і обґрунтуванню основних висновків дослідження;
- змістову завершеність, цілісність та логічну зв'язність викладу;
- об'єктивність, цілеспрямованість і відсутність емоційного забарвлення тексту.

Структура звіту:

- титульний аркуш (Додаток А);
- постанова задачі;
- опис програми (класів, методів);
- керівництво користувача (скріни працюючої програми);
- висновки.
- додаток (код програми).

Робота подається у вигляді текстового файлу текст роботи повинен бути виконаний у вигляді комп'ютерного набору на одному боці аркуша білого паперу формату А4 (210х297мм). Текст кожного індивідуального завдання здобувача розміщується на аркуші книжкової або альбомної орієнтації, яка обмежується полями: лівим – 30 мм, правим – 10 мм, верхнім – 20 мм, нижнім – 20 мм. Для великих таблиць і рисунків допускається альбомна орієнтація сторінок, на яких вони розміщені. Текст роботи оформлюється шрифтом Arial, кеглем 14 з одинарним міжрядковим інтервалом. Для таблиць допускається використання шрифту Arial, кеглем 12.

Заголовки пунктів у разі їх виділення слід починати з абзацного відступу і друкувати маленькими літерами, крім першої великої, не підкреслюючи, без крапки в кінці. Абзацний відступ повинен бути однаковим упродовж усього тексту і дорівнювати 1,25 см. Якщо заголовок складається з двох і більше речень, їх розділяють крапкою. Перенесення слів у заголовку не допускається.

Відстань між заголовком і попереднім текстом повинна бути два рядки, між заголовком і подальшим текстом – один рядок.

Не допускається розміщувати назву пункту в нижній частині сторінки, якщо після неї розміщено тільки один рядок тексту.

Сторінки роботи нумеруються арабськими цифрами з наскрізною нумерацією по всьому тексту. Номер сторінки розміщується у правому верхньому куті без крапки в кінці. Титульний аркуш та зміст входять до

загальної нумерації, але номер сторінки на них не проставляється. Ілюстрації та таблиці, які подані на окремих сторінках, також включаються до загальної нумерації. Пункти роботи нумеруються арабськими цифрами без крапки після номера.

Ілюстрації (рисунки, графіки, схеми, діаграми) повинні розміщуватися одразу після тексту, де вони згадуються вперше, або на наступній сторінці. У тексті роботи мають бути обов'язкові посилання на всі ілюстрації.

Цифрові дані зазвичай подаються у вигляді таблиць. Таблиці розміщуються безпосередньо після тексту, де вони вперше згадуються, або на наступній сторінці. У тексті повинні бути відповідні посилання на всі таблиці.

Назва таблиці складається зі слова «Таблиця», її порядкового номера та заголовка, який стисло відображає зміст поданих у ній даних. Повна назва таблиці зазначається один раз над таблицею зліва, з абзацним відступом.

Якщо таблиця переноситься на наступну сторінку, над її продовженням із абзацного відступу пишуть: «Продовження таблиці Х» або «Кінець таблиці Х», де Х – номер таблиці. Таблиці нумеруються арабськими цифрами послідовно в межах усієї роботи.

Заголовки та дані таблиці можуть бути оформлені через одинарний інтервал, шрифтом Arial, 12 кегль. Заголовки граф починають із великої літери, а підзаголовки – з малої, якщо вони становлять одне речення із заголовком. Якщо підзаголовки мають самостійне значення, їх пишуть з великої літери. У кінці заголовків і підзаголовків крапка не ставиться. Усі заголовки та підзаголовки граф подаються в однині.

ПОДАННЯ НА ПЕРЕВІРКУ ПРАКТИЧНОЇ РОБОТИ ТА КРИТЕРІЇ ОЦІНЮВАННЯ

Критерії оцінювання результатів виконання практичної роботи.

Захист практичної відбувається на практичному занятті згідно з графіком контрольних точок, передбаченим робочою програмою дисципліни, а оцінка за його виконання виставляється викладачем у відповідній активності в системі Moodl і враховується ним при визначенні поточної успішності здобувача.

Максимальна кількість балів, яку здобувач може отримати за кожне виконане практичну роботу – 5 балів. Оскарження оцінки може бути здійснене на останньому практичному занятті модуля.

Критерії оцінювання:

- здобувач вищої освіти підготував звіт з практичної роботи за своїм варіантом у вигляді файлу *pdf, в якому надав: умову задачі, скріни коду програми та її виконання у програмному забезпеченні, додаток з кодом програми (3 бали);

- здобувач вищої освіти захистив практичну роботу: продемонстрував роботу програми та відповів на запитання викладача (2 бали).

Додаткові зауваження:

- здобувач вищої освіти може оскаржити отримані оцінки в порядку, передбаченому Положенням про організацію освітнього процесу та Положенням про політику та процедури врегулювання конфліктних ситуацій;

- викладач не має права знижувати оцінку за практичну роботу, якщо воно не було складено вчасно, однак в разі, якщо така робота була оцінена пізніше.

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1 Порєв В. М. Об'єктно-орієнтоване програмування : конспект лекцій. Київ : КПІ ім. Ігоря Сікорського, 2022. 271 с.

2 Schildt H. Java: A Beginner's Guide. 8th Edition. McGraw-Hill Education, 2018. 684 p.

3 Horstmann C. S. Core Java Volume I – Fundamentals. 11th Edition. Prentice Hall, 2018. 889 p.

4 Об'єктно-орієнтоване програмування мовою Java : метод. вказівки до лабораторних робіт з курсу "Об'єктно-орієнтоване програмування" : для студентів спец. 122 – Комп'ютерні науки, 126 – Інформаційні системи та технології / уклад.: О. М. Нікуліна, Л. В. Іванов, Н. В. Коцюба. Харків : Друкарня Мадрид, 2022. 64 с.

Web-ресурси

5 Іванов Л. В. Основи програмування Java : Iwanoff : веб-сайт. URL: http://www.iwanoff.inf.ua/ (дата звернення: 17.04.2025).

6 Освоюємо Java : Вікіпідручник : веб-сайт. URL: http://uk.wikibooks.org/wiki/Освоюємо_Java (дата звернення: 17.04.2025).

7 Програмування на Java : Javaland : веб-сайт. URL: http://javaland.com.ua (дата звернення: 17.04.2025).

8 Java Підручник : W3SchoolsUA : веб-сайт. URL: https://w3schoolsua.github.io/java/index.html#gsc.tab=0 (дата звернення: 17.04.2025).

.

Додаток А

ПРИКЛАД ОФОРМЛЕННЯ ТИТУЛЬНОГО ЛИСТА ПРАКТИЧНОЇ РОБОТИ

ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ «МЕТІНВЕСТ ПОЛІТЕХНІКА»
Кафедра цифрових технологій та проєктно-аналітичних рішень

ПРАКТИЧНА РОБОТА № 1

з навчальної дисципліни «ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ»

ВИКОРИСТАННЯ БАЗОВИХ ЗАСОБІВ МОВИ

Виконав (ла): здобувач (ка) вищої освіти
за освітньо-професійною програмою
«Комп'ютерні науки»
гр. 0122-хх-х
_____
(Прізвище, ім'я, по батькові повністю)

Прийняла : д.т.н., професор кафедри ЦТПАР
Нікуліна О.М.

Запоріжжя – 202_

Навчально-методичне видання

Олена Миколаївна Нікуліна

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА JAVA

методичні вказівки
до виконання практичних робіт

самостійне електронне мережеве видання

Публікується в авторський редакції