Державна служба з надзвичайних ситуацій України

Львівський державний університет безпеки життєдіяльності

Олександр ПРИДАТКО, Олександр ХЛЕВНОЙ, Назарій БУРАК

ОСНОВИ ПРОГРАМУВАННЯ (МОВОЮ JAVA)

Курс лекцій

Львів 2019

УДК 81'25
ББК 81.18
П 87

Придатко, Олександр Володимирович.
Основи програмування (мовою Java): курс лекцій / О. В. Придатко, О. В. Хлевной, Н. Є. Бурак,. – Львів : ЛДУ БЖД, 2019. – 180 с.

Рецензенти:

Луб П.М., канд. техн. наук, доцент, доцент кафедри інформаційних систем та технологій Львівського національного аграрного університету;

Кухарська Н.П., канд. фіз.-мат. наук, доцент, доцент кафедри управління інформаційною безпекою Львівського державного університету безпеки життєдіяльності.

У курсі лекцій подано загальний огляд мови програмування Java, висвітлено історію її створення, основні принципи і структуру, порівняння з іншими мовами програмування. Розглянуто основи мови Java: типи даних, змінні, операції та оператори.

Наведено особливості, які є тільки у мові Java, і зв’язані з написанням операційно і платформно незалежних програм. Викладено основи об’єктно-орієнтованого програмування: класи, інтерфейси, пакети.

Для курсантів, студентів, науковців, програмістів персональних комп΄ютерів, які бажають перейти на нові технології програмування та створювати сучасні програмні продукти.

Рекомендовано до друку рішенням вченої ради
Львівського державного університету безпеки життєдіяльності
(протокол № 3 від 30 жовтня 2019 року)

© Придатко О.В., 2019
© Хлевной О.В., 2019
© Бурак Н.Є., 2019
© ЛДУБЖД, 2019

ЗМІСТ

Лекція 1.
Вступ до курсу. Апаратні та програмні засоби ЕОМ........................6
1.1. Загальні відомості про ЕОМ...6
1.2. Архітектура та принцип роботи ЕОМ...10
1.3. Програмне забезпечення...12
1.4. Файлова система..14
Лекція 2. Системи числення...17
2.1. Загальні засади кодування алфавітно-цифрової інформації в ЕОМ. 17
2.2. Основні позиції систем числення...19
2.3. Переведення чисел між системами числення. 21
Лекція 3. Огляд мов програмування...27
3.1. Загальні відомості про мови програмування. Мови низького рівня. 27
3.2. Огляд основних мов програмування високого рівня. 30
Лекція 4. Інтегроване середовище розробки мовою Java.............38
4.1. Огляд об'єктно-орієнтованої мови Java.....................................38
4.2. Інтегроване середовище розробки..43
Лекція 5. Базові елементи мови Java..51
5.1. Типи даних та змінні...51
5.2. Оператори..55
Лекція 6. Оператори вибору..61
6.1. Умовний оператор if..61
6.2. Оператор розгалуження (множинного вибору) switch...............65
Лекція 7. Оператори циклів...71
7.1. Цикл з передумовою while...71
7.2. Цикл з післяумовою do-while..74
7.3. Цикл з лічильником for..75
7.4. Оператори переходу..81
Лекція 8. Масиви..91
8.1. Одновимірні масиви...91
8.2. Багатовимірні масиви...96
Лекція 9. Потоки введення/виведення та рядки в Java..............103
9.1. Потоки введення та виведення. Клас Scanner..........................103
9.2. Рядки. Клас String...105
Лекція 10. Числові методи в Java..115
10.1. Математичний клас Math та його методи..............................115
10.2. Псевдовипадкові числа...120

10.3. Методи класу чисел 122
Лекція 11. Введення в класи 127
11.1. Введення в класи 127
11.2. Конструктори 132
Лекція 12. Введення в методи 140
12.1. Типізовані та void-методи 140
12.2. Методи, що приймають параметри 144
12.3. Використання об'єктів в якості параметрів методу 147
Лекція 13. Поліморфізм 154
13.1. Вступ. Введення в поліморфізм 154
13.2. Перевизначення методів 155
13.3. Перевизначення конструкторів 160
13.4. Рекурсія 162
Лекція 14. Ієрархія класів. Наслідування 168
14.1. Вступ. Введення в наслідування 168
14.2. Вкладені та внутрішні класи 168
14.3. Наслідування 172

ВСТУП

Освітня програма підготовки бакалавра зі спеціальності «Комп'ютерні науки» передбачає оволодіння студентами низкою фахових компетенцій в області програмування, досягнення яких організовано шляхом вивчення курсів «Об'єктно-орієнтоване програмування», «Системне програмування», «Клієнт-серверне програмування», «Програмування для мобільних платформ», «Якість програмного забезпечення та тестування», «Front end – розробка» тощо. Проте базовим та початковим курсом, який є фундаментальним для вивчення зазначених та інших дисциплін, є курс «Основи програмування».

Основною метою цього курсу є формування базових понять з основ програмування. З цією метою до курсу включено два розділи: основи процедурного програмування та основи об'єктно-орієнтованого програмування. Після опрацювання поданого матеріалу курсанти та студенти не стануть професійними програмістами, які використовують отримані навички для пошуку найбільш вигідних пропозицій на ринку праці. Курс не призначено також для навчання методологій програмування на рівні, що перевищує початковий. Його мета – висвітлення базових принципів сучасного програмування із наведенням зрозумілих прикладів.

Для курсантів та студентів цей курс може стати першим щаблем для побудови успішної кар'єри у сфері комп'ютерних наук та інформаційних технологій.

ЛЕКЦІЯ 1.
ВСТУП ДО КУРСУ.
АПАРАТНІ ТА ПРОГРАМНІ ЗАСОБИ ЕОМ

1.1. Загальні відомості про ЕОМ
1.2. Архітектура та принцип роботи ЕОМ
1.3. Програмне забезпечення
1.4. Файлова система

1.1. Загальні відомості про ЕОМ

Електронна обчислювальна машина (ЕОМ) – це комплекс апаратних пристроїв для автоматичної обробки цифрової інформації за наперед заданою програмою з метою її подальшого перетворення, зберігання, введення або виведення.

Інформація (від лат. пояснення, виклад, тлумачення) – це набір відомостей про об'єкти, явища і процеси.

Для розв'язування різних задач використовуються різні програми, але структура ЕОМ (програмний принцип роботи) залишається незмінною.

Покоління ЕОМ:

▪ Перше покоління (1946 р. – початок 50-х рр.). *Базувалося на електронних вакуумних лампах.* ЕОМ вирізнялися великими габаритами, значним споживанням енергії, малою швидкодією, низькою надійністю, програмуванням в кодах. Для вводу-виводу даних використовувалися перфокарти та перфострічки, швидкодія сягала 10-20 тис. операцій в секунду.

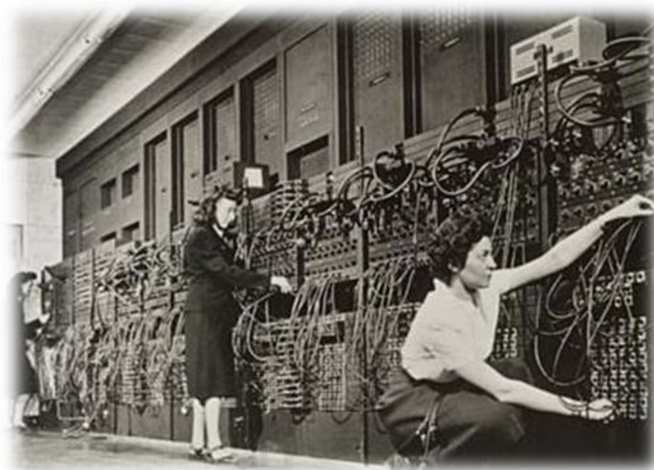

Рисунок 1.1 – Загальний вигляд ЕОМ І покоління

Рисунок 1.2 – Загальний вигляд електронних вакуумних ламп

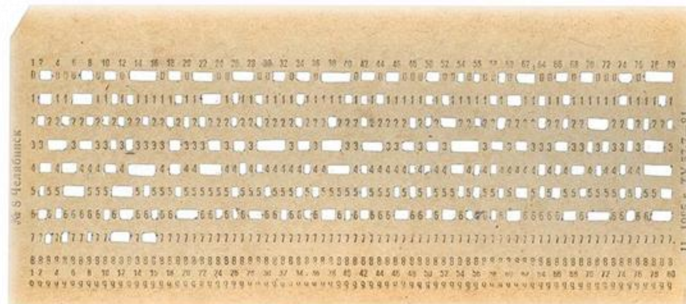

Рисунок 1.3 – Загальний вигляд перфокарти

▪ Друге покоління (1955 р. – початок 60-х рр.). *Базувалося на транзисторах*. Покращилися в порівнянні з ЕОМ попереднього покоління всі технічні характеристики. Для програмування використовувалися алгоритмічні мови, а для вводу-виводу даних – магнітна стрічка. Швидкодія: 100 тис. – 1 млн. операцій в секунду.

Рисунок 1.4 – Загальний вигляд ЕОМ II покоління

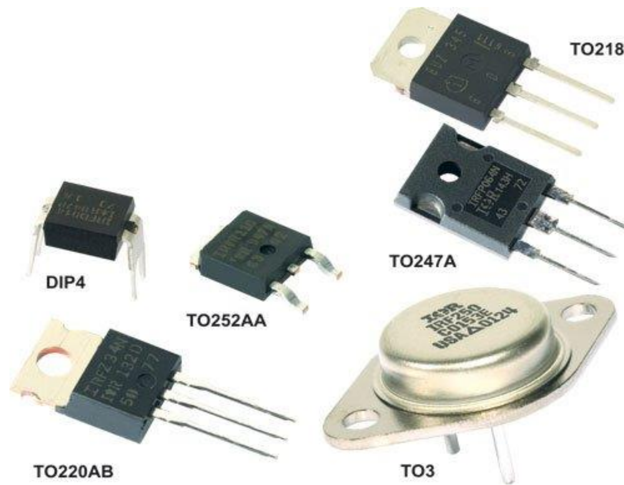

Рисунок 1.5 – Загальний вигляд транзисторів

Рисунок 1.6 – Загальний вигляд магнітних носіїв інформації у вигляді стрічки

▪ Третє покоління (1965 р. – кінець 70-х рр.). *Базувалося на інтегральних мікросхемах*, які дозволили монтувати електронні схеми з дуже високою щільністю розташування елементів. Внаслідок цього відбулося різке зниження габаритів ЕОМ, підвищення їх надійності, збільшення продуктивності. З'явилася можливість доступу з віддалених терміналів. Швидкодія: 1 – 10 млн. операцій в секунду.

Рисунок 1.7 – Загальний вигляд ЕОМ III покоління

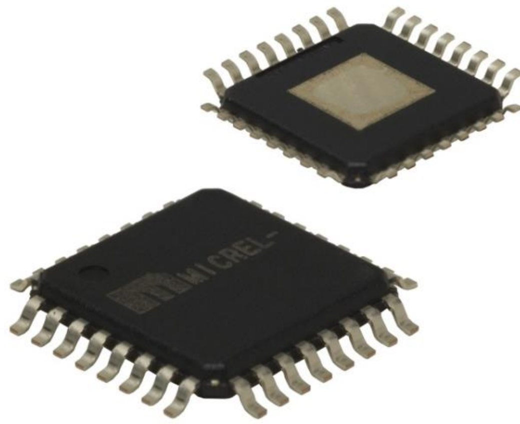

Рисунок 1.8 – Загальний вигляд інтегральних мікросхем

▪ Четверте покоління (70-ті рр. – кінець 80-х). *Базувалося на використанні мікропроцесорів.* Вчергове покращились технічні характеристики. Розпочалось масове виробництво персональних комп'ютерів. З’явились магнітні диски та дискети для вводу-виводу інформації.

Рисунок 1.9 – Загальний вигляд ЕОМ IV покоління

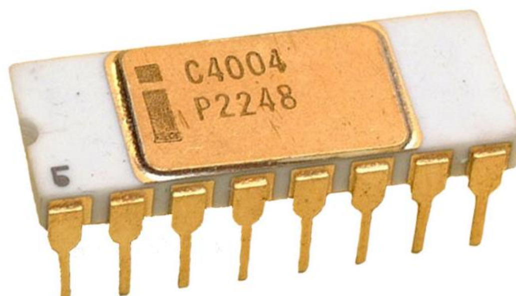

Рисунок 1.10 – Загальний вигляд мікропроцесора IV покоління ЕОМ

Рисунок 1.11 – Магнітний диск та дискета для вводу-виводу інформації

▪ П’яте покоління (з середини 80-х рр.). Почалася розробка інтелектуальних комп'ютерів. Відбулося впровадження в усі сфери комп'ютерних мереж та їх об'єднання, розпочалось використання розподіленої обробки даних. Багато спеціалістів об’єднують четверте та п’яте покоління та вважають ці два покоління логічним продовженням один одного, адже обидва покоління ЕОМ засновані на використанні мікропроцесорів.

Рисунок 1.12 – Загальний вигляд мікропроцесора останнього покоління

1.2. Архітектура та принцип роботи ЕОМ

Архітектура комп’ютерів (ЕОМ) – це сукупність апаратних ресурсів (hardware) комп’ютера (ЕОМ).

В основу архітектури переважної більшості ЕОМ (в тому числі сучасних) покладено принципи, які було сформульовані ще 1945 року американським математиком Джоном фон Нейманом. Усі комп’ютери, побудовані згідно з цими принципами, відомі тепер як комп’ютери з фоннейманівською архітектурою.

Основні принципи архітектури комп'ютерів фон Неймана:

- використання двійкової системи числення для кодування інформації у комп'ютері;
- програмне керування роботою комп'ютера;
- зберігання програм у пам'яті комп'ютера;
- адресація пам'яті.

Рисунок 1.13 – Типова архітектура ЕОМ (персонального комп'ютера)

Складові типової архітектури:

- *Арифметично-логічний пристрій (АЛП)* – призначений для виконання арифметичних та логічних операцій.
- *Керуючий пристрій* (пристрій управління) *(КП)* – призначений для управління роботою АЛП та автоматичного і послідовного виконання програм, введених в ЕОМ. *АЛП і КП є основними складовими процесора*.
- *Оперативний запам'ятовуючий пристрій (ОЗП)* – основна пам'ять ЕОМ. Служить для запису та зберігання програм, даних, проміжних і кінцевих результатів під час сеансу роботи.
- *Пристрої введення та виведення інформації* (ПВ та ПВВ) – призначені для вводу програм та даних і виводу результатів їх роботи. Вони служать для зв'язку ЕОМ з навколишнім світом.

Взаємодію між усіма компонентами забезпечує так звана *системна шина*. Вона об'єднує усі складові елементи ЕОМ та забезпечує передачу сигналу між окремими пристроями. Фізично її

розташовано на материнській платі, навкруги якої і збирається сучасний комп'ютер.

Принцип відкритої архітектури – це підхід до проектування комп’ютера із забезпеченням можливості його збирання з незалежно виготовлених частин (аналогічно до дитячого конструктора). Способи з’єднання різних елементів комп’ютера мають забезпечувати можливість для удосконалення його окремих частин (долучення нових, або заміна існуючих на більш досконалі).

Принцип роботи ЕОМ :

✓Програма та початкові дані через пристрій вводу потрапляють в ЕОМ та розміщуються в пам’яті, де зберігаються протягом всього часу її виконання. Керування роботою ЕОМ здійснюється КП за алгоритмом введеної програми.

✓Після того, як оператор подає вказівку розпочати виконання програми, КП посилає запит першої команди до ОЗП (пам’яті).

✓ОЗП надсилає першу команду до КП, де вона розкодовується і сигнали керування розходяться на відповідні блоки ЕОМ.

✓У результаті дії сигналів керування дані пересилаються з ОЗП до АЛП, де над ними виконуються відповідні операції. Результати операцій з АЛП повертаються до ОЗП, де зберігаються до моменту подальшого використання.

✓АЛП повідомляє КП про завершення операції, після чого КП надсилає до пам’яті запит наступної команди і пункти 3-5 повторюються до завершення виконання програми.

Представлений опис є дуже загальний, в реальності все значно складніше. Проте основна мета курсу не полягає у вивченні детальної конструкції та принципу роботи ЕОМ, ця тема необхідна лише для формування часткової уяви про процеси, які виконує ЕОМ під час реалізації програм.

1.3. Програмне забезпечення

Програмне забезпечення (ПЗ) (software) – сукупність програм та службових даних, призначених для керування роботою ЕОМ (комп’ютера). ПЗ комп’ютерів можна поділити на такі основні класи:

✓операційна система та сервісні програми (системне ПЗ);

✓інструментальні мови та системи програмування;

✓прикладні системи (прикладне ПЗ).

Операційна система (ОС) (operating system) – це сукупність програмних засобів, яка здійснює розподіл ресурсів ПК та керування роботою усієї обчислювальної системи.

Відомими операційними системами сьогодні є ОС Windows, Linux, iOS, Android та ін.

Операційна система забезпечує функціонування комп’ютера шляхом: керування пам’яттю, введенням-виведенням даних, файловою системою, взаємодією процесів; диспетчеризацію процесів; захистом інформації; обліком використовування ресурсів; опрацювання командної мови.

До складу сучасних операційних систем входять кілька підсистем, основні з яких:

✓ підсистема управління процесами;

✓ файлова підсистема;

✓ драйвери – спеціальні програми, які забезпечують роботу з апаратурою;

✓ функції для організації взаємодії програм із користувачем;

✓ служба безпеки – розмежовування прав доступу.

Інструментальні мови та системи програмування використовують для реалізації програмних алгоритмів, тобто для розробки програм системного та прикладного призначення (програмування).

Інструментальні мови програмування поділяють на дві основні категорії:

✓ мови низького (машинного) рівня – асемблери, близькі за структурою до інструкцій процесора (машинні мови програмування);

✓ мови високого рівня – призначені для полегшення процесу програмування, а їх інструкції багато в чому нагадують людські мови (кожна з команд високорівненої мови поєднує в собі одразу кілька машинних команд). Розрізняють чотири різновиди мов високого рівня: імперативні (процедурні), наприклад Pascal, C; функціональні – Lisp; логічні – Prolog; об’єктно-орієнтовані – C++, C#, Java, Object Pascal тощо. Програми, створені мовою високого рівня, перекодовуються на машинну мову інтерпретаторами та компіляторами. Інтерпретатор автоматично, у міру виконання програми, перетворює її команди (чи код) на команди машинної мови. А компілятор транслює усю введену програму за командою програміста. Результатом компілювання є згенерований об’єктний код (байт-код).

Системи програмування надають зручний інтерфейс для створення та налагодження програм тією чи іншою мовою (їх ще називать інтегрованими середовищами для програмування). Прикладами систем програмування є Eclipse, C++ Builder, Visual C++, IntelliJ IDEA, NetBeans, MonoDevelop та багато інших.

Прикладні програми призначені для розв'язування прикладних задач певних класів. Приміром, для виконання математичних інженерних обчислень використовують спеціальні математичні пакети MathCAD, Mathematika, Matlab, для введення й редагування текстової інформації застосовують текстові редактори Microsoft Word, Блокнот та інші, опрацьовувати табличні дані зручно в Microsoft Excel, а для опрацювання графічних даних існують пакети Adobe Photoshop, Paint тощо.

Отже, кожна ЕОМ має дві основні складові – апаратне *(hardware)* і програмне *(software)* забезпечення. Збої у роботі однієї з програм можуть спричинити збої у функціонуванні комп'ютера й одержання помилкових результатів його роботи. Помилки апаратної частини приводять до неможливості реалізації команд програмного забезпечення.

1.4. Файлова система

Файлова система – це сукупність каталогів та файлів, які зберігаються на носіях зовнішньої пам'яті довготермінового зберігання інформації.

Файлова система є основним інформаційним об'єктом ОС. *Файл* – це іменована область зовнішньої пам'яті для зберігання програм та даних.

Будь-який файл має такі характеристики: ім'я, розмір, дату останнього зберігання, певне місце розташування на диску та атрибути доступу. Імена файлів складаються з власного імені, крапки і розширення:

<ім'я>.<розширення>,

де ім'я – набір із символів алфавіту, цифр і спеціальних символів; розширення визначає тип файлу і містить, зазвичай, три або чотири символи (наприклад, текстові файли мають розширення txt, документи Microsoft Word – doc чи rtf, таблиці Microsoft Excel – xls,

бази даних Microsoft Access – mdb, графічні файли – bmp, psd, jpg, gif тощо).

Інформація про всі атрибути файлу міститься у каталогах. *Каталог* (тека, директорія) – це логічна одиниця організації диска, яка має власне ім'я і може містити в собі файли та інші каталоги (підкаталоги).

Головний каталог диска називають кореневим. Ім'я кореневого каталогу складається з імені диска та символу (двокрапки). Інформація про всі атрибути файлів та підкаталогів використовується ОС для визначання повного місцеперебування файлу, яке записується у вигляді послідовності імен каталогів та підкаталогів, розпочинаючи з кореневого (наприклад: C:\Program Files\Microsoft Office\Clipart \A16.gif).

Підкаталоги, які входять до кореневого каталога, називаються підкаталогами 1-го рівня. Підкаталоги, які входять до складу підкаталога 1-го рівня, називаються підкаталогами 2-го рівня і т. д. Ієрархічну побудову диска можна подати у вигляді дерева підкаталогів.

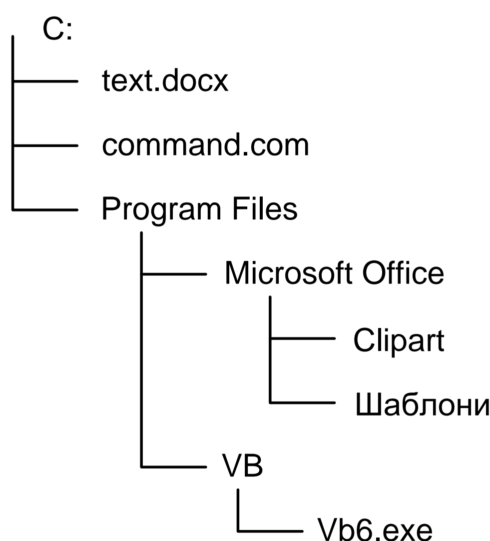

Рисунок 1.14 – Фрагмент дерева каталогів

Для роботи з файлами існує кілька стандартних операцій, які підтримують усі операційні системи: створеання, копіювання, переміщення, перейменування, видалення. Спеціальне призначення мають файли-ярлики. У такому файлі міститься посилання на інший файл (каталог, програму, документ тощо). Запуск ярлика відкриє той об'єкт, на який він посилається.

Висновок: основною задачею цієї теми є донесення загальних понять про апаратні та програмні засоби ЕОМ, які необхідні для

формування у майбутніх фахівців базових понять про процеси, які виконує ЕОМ під час реалізації системних та прикладних програм. Детальне ознайомлення з історією розвитку, архітектурою та принципом роботи ЕОМ можливе під час вивчення курсу «Комп'ютерна схемотехніка та архітектура комп'ютерів»,.

Література

1. С++. Основи програмування. Теорія та практика : підручник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін.]; за ред. О. Г.Трофимнко. – Одеса: Фенікс, 2010. – 544 с. (стор. 11-14)
2. Prometheus. Курс «Основи програмування» (Лекція 1: Техноманія навколо). [Електронний ресурс]. – Доступний з https://edx.prometheus.org.ua/ courses /KPI/ Programming101/2015_T1

Запитання до лекції

1. Що таке ЕОМ?
2. Скільки поколінь ЕОМ виділяють? Чим відрізняються ЕОМ різних поколінь?
3. Що таке архітектура ЕОМ та які її складові та принципи?
4. Що таке програмне забезпечення, на які класи воно поділяється?
5. Перелічіть різновиди мов програмування високого рівня.
6. Що таке файлова система?

ЛЕКЦІЯ 2.
СИСТЕМИ ЧИСЛЕННЯ

2.1. Загальні засади кодування алфавітно-цифрової інформації в ЕОМ.
2.2. Основні позиції систем числення.
2.3. Переведення чисел між системами числення.

2.1. Загальні засади кодування алфавітно-цифрової інформації в ЕОМ

С***истема числення*** – це набір правил записування і позначення чисел за допомогою певного набору знаків (цифр).

Залежно від способу використання цих знаків системи числення поділяються на:

- непозиційні;
- позиційні;
- змішані.

Непозиційні системи – це системи числення, в яких значення цифр не залежить від позиції (розряду) в записі числа.

Прикладом непозиційної системи є римська система числення, яка має такі числові значення: I – 1, V – 5, X – 10, L – 50, C – 100, D – 500, M – 1000. Отож, число 30 має вигляд: XXX. Тут цифра X в будь-якому місці означає число десять. Запис інших чисел: IV – 4, VI – 6, IX – 9, XI – 11, XL – 40, LX – 60, XC – 90, CX – 110, CM – 900, MC – 1100, MCMLXXXIX – 1989. У записі цих чисел значення кожної літери не залежить від місця, на якому вона стоїть. Для запису великих чисел доводиться долучати все нові й нові знаки. Непозиційні системи незручні для запису великих чисел і для виконання арифметичних дій.

Позиційні системи – системи числення, в яких значення цифр залежить від позиції (розряду) в записі числа.

У позиційній системі числення значення кожної цифри залежить від її розряду (позиції) у послідовності цифр, що відображають число. Десяткова система числення, якою ми користуємося у повсякденній практиці, є позиційною системою. Наприклад, у записі числа 111 цифра 1 повторюється три рази, але при цьому перша означає кількість сотень, друга – кількість десятків, третя – кількість одиниць. ЕОМ працюють із використанням позиційної системи числення.

Змішані системи – це системи числення, в яких кількість допустимих цифр для різних розрядів (позицій) є різною.

Найвідомішим прикладом змішаної системи числення є представлення часу у вигляді кількості діб, годин, хвилин і секунд. При цьому величина *d* днів *h* годин *m* хвилин *s* секунд відповідає значенню $d \cdot 24 \cdot 60 \cdot 60 + h \cdot 60 \cdot 60 + m \cdot 60 + s$ секунд.

Будь-яка (числова, текстова, графічна, аудіо, відео тощо) інформація в ЕОМ кодується у цифровому вигляді за допомогою двійкової системи числення. Використання двійкової системи числення суттєво спрощує апаратну реалізацію пристроїв ПК, оскільки в цій системі є лише дві цифри: **0** та **1**.

Зручність використання двійкової системи числення в обчислювальній техніці зумовлена тим, що електронні перемикачі можуть перебувати лише в одному із двох станів: ***увімкненому чи вимкненому***. Ці стани можна кодувати двома цифрами: ***1*** чи ***0***. Так же у двох станах може перебувати канал передавання даних "рівень напруги є високий" чи "рівень напруги є низький".

Зважаючи на основні принципи кодування інформації, перейдемо до одиниць її виміру.

Одна двійкова цифра називається ***бітом*** (від англ. ***bi***nary digi***t*** – двійкова цифра). За допомогою одного біта можна закодувати два різних інформаційних повідомлення, які умовно позначаються символами «0» або «1».

За допомогою *n* бітів можна закодувати 2^n інформаційних повідомлень. Отже, *біт є мінімальною одиницею обсягу пам'яті*, проте на практиці ніхто не опрацьовує дані розміром в один біт.

Байтом називають послідовність з восьми бітів. За допомогою одного байта можна закодувати 2^8 (256) різних комбінацій бітів, тобто змінна розміром в один байт може зберігати числа в межах від 0 до 255. Наприклад, число 1101 0011 – це інформація обсягом в один байт.

Ще однією одиницею вимірювання інформації, меншою за один байт, окрім біта, є ніббл. *Ніббл* – одиниця вимірювання інформації, яка становить чотири біти і, відповідно, може мати 2^4 різних значення. Цього об'єму інформації цілком вистачає для кодування кирилиці. Проте в ЕОМ використовуються також літери латинського алфавіту, цифри, різні математичні символи, де цього обсягу пам'яті буде недостатньо. Тому для кодування усіх згаданих символів використовується восьмирозрядна послідовність бітів (один байт). Наприклад: цифра «9» кодується послідовністю бітів 0011 1001, літера латини «W» – 0101 0111.

Для зручності позначання тисяч та мільйонів байтів використовуються такі одиниці виміру інформації:

- кілобайт (1 Кб (kB) = 2^{10}байт = 1024 байт);
- мегабайт (1 Мб (MB) = 2^{20} байт = 1024 Кб = 1 048 576 байт)
- гігабайт (1 Гб (GB) = 2^{30} байт = 1024 Мб = 1 073 741 824 байт);
- терабайт (1 Тб (TB) = 2^{40} байт = 1024 Гб = 1 099 511 627 776 байт);
- петабайт (1 Пб (PB) = 2^{50} байт = 1024 Тб);
- ексабайт (1 Еб (EB) = 2^{60} байт = 1024 Пб);
- зетабайт (1 Зб (ZB) = 2^{70} байт = 1024 Еб);
- йотабайт (1 Йб (YB) = 2^{80} байт = 1024 Зб).

Наприклад, якщо на сторінці тексту розміщується в середньому 2500 знаків, то 1 Мб – це приблизно 400 сторінок, а 1 Гб – 400 тисяч сторінок. *Зауважимо, що одиниці вимірювання інформації ґрунтуються на степенях числа 2*. Десяткові префікси (кіло, мега тощо) дописуються лише умовно, оскільки 2^{10} = 1024 – число, близьке до 1000.

Іноді десяткові префікси застосовують і у прямому значенні. Наприклад, при зазначенні ємності жорстких дисків чи модулів пам’яті, а також при зазначенні пропускної здатності каналів передавання даних (мереж). Виробники використовують число 10 як основу для піднесення до степеня. Один гігабайт у цьому випадку дорівнює 10^9 б. При цьому реальна ємність, наприклад, 250-гігабайтного вінчестера (HDD) становить приблизно 232 Гб.

2.2. Основні позиції систем числення

Основа системи числення – це кількість символів в алфавіті системи числення (десяткова – 10, двійкова – 2, вісімкова – 8 тощо).

Кожне число N у позиційній системі числення з основою q можна подати в єдиний спосіб у вигляді полінома:

$$N_q = a_n \cdot q^n + a_{n-1} \cdot q^{n-1} + ... + a_1 \cdot q^1 + a_0 \cdot q^0 + a_{-1} \cdot q^{-1} + ... + a_k \cdot q^k = \sum_{i=-k}^{n-1} a_i \cdot q^i, \quad (1)$$

де q – основа позиційної системи числення, яка визначає її назву; a_i – цифри числа відповідного i-го розряду (позиції в числі); n – кількість цифр цілої частини числа; k – кількість цифр дробової частини числа.

Наприклад, число ***5692*** можна розписати

$$5692 = 5 \cdot 1000 + 6 \cdot 100 + 9 \cdot 10 + 2 \cdot 1.$$

Пронумеруємо розряди (позиції) числа:

Цифра	5	6	9	2
Позиція в числі	3	2	1	0

Використовуючи вираз (1) та одержані розряди, число 5692 можна записати:

$$5692_{10} = 5 \cdot 10^3 + 6 \cdot 10^2 + 9 \cdot 10^1 + 2 \cdot 10^0 = 5000 + 600 + 90 + 2.$$

За $q = 10$ матимемо найпоширенішу десяткову систему числення. У ній кожне число записується поєднанням десяткових цифр, а внесок конкретної цифри залежить від її позиції – розряду. *Відлік розрядів ведеться справа наліво*. Перший розряд називається розрядом одиниць, другий – десятків, третій – сотень і т.д.

Для двійкової системи за $q = 2$ вираз (1) набуде вигляду:

$$N_q = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^{-1} + \ldots + a_k \cdot 2^k = \sum_{i=-k}^{n-1} a_i \cdot 2^i,$$

Наприклад:

$$10101.101_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + + 1 \cdot 2^{-3} = 21.625_{10}.$$

За $q = 8$ в аналогічний спосіб здобувають формулу для вісімкової системи, а за $q = 16$ – для шістнадцяткової системи числення. Існує чимало різноманітних позиційних систем числення, відмінних від десяткової, але у зв’язку із розвитком обчислювальної техніки, найбільшого поширення набули саме двійкова, вісімкова та шістнадцяткова.

Таблиця 2.1

Основні системи числення

Система числення	*q*	*Алфавіт системи числення*
Двійкова	2	0, 1
Вісімкова	8	0, 1, 2, 3, 4, 5, 6, 7
Десяткова	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Шістнадцяткова	16	0, 1, …, 9, A(10), B(11), C(12), D(13), E(14), F(15)

Розглянемо кілька прикладів записування чисел у різних системах числення та їхнє десяткове представлення:

- ✓ $194.38_{10} = 1 \cdot 10^2 + 9 \cdot 10^1 + 4 \cdot 10^0 + 3 \cdot 10^{-1} + 8 \cdot 10^{-2} = 194.38_{10}$;
- ✓ $10011.1_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 19.5_{10}$;
- ✓ $237.2_8 = 2 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 + 2 \cdot 8^{-1} = 159.25_{10}$;
- ✓ $A1F_{16} = 10 \cdot 16^2 + 1 \cdot 16^1 + 15 \cdot 16^0 = 2591_{10}$.

Для зберігання і опрацювання даних в ЕОМ здебільшого використовується двійкова система числення. Але на практиці для скорочення запису та зручності користувачів частіше використовується шістнадцяткова система.

Отже, для закріплення технології переведення чисел двійкової системи числення в зручну для користувача десяткову, наведемо ще декілька прикладів:

- ✓ $1_2=1\cdot2^0=1_{10}$;
- ✓ $10_2=1\cdot2^1 + 0\cdot2^0=2_{10}$;
- ✓ $11_2=1\cdot2^1+1\cdot2^0=3_{10}$;
- ✓ $100_2=1\cdot2^2+0\cdot2^1+0\cdot2^0=4_{10}$;
- ✓ $101_2=1\cdot2^2+0\cdot2^1+1\cdot2^0=5_{10}$;
- ✓ $110_2=1\cdot2^2+1\cdot2^1+0\cdot2^0=6_{10}$;
- ✓ $111_2=1\cdot2^2+1\cdot2^1+1\cdot2^0=7_{10}$;
- ✓ $1000_2=1\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0=8_{10}$;
- ✓ $1001_2=1\cdot2^3+0\cdot2^2+0\cdot2^1+1\cdot2^0=9_{10}$.

2.3. Переведення чисел між системами числення

При виконанні задач із використанням ЕОМ введення початкових даних і виведення результатів обчислень зазвичай виконується користувачами у звичній для них десятковій системі числення. Але, з огляду на те, що переважна більшість ЕОМ використовує двійкову систему числення, постає потреба у переведенні числа з однієї системи числення до іншої. Переведення чисел з *q*-ої системи числення до десяткової виходить з поліноміального виразу (1) та описано в попередньому навчальному питанні.

Переведення цілого десяткового числа в *q*-ту систему числення відбувається у два етапи: спочатку переводиться ціла частина, потім дробова, після чого ліворуч від крапки записується ціла частина, а праворуч – дробова. Суть переведення полягає у послідовному діленні десяткового числа та його часток на значення основи системи *q*. Ділення виконується, допоки чергова частка не буде меншою за основу *q*. Остача, обчислена на останньому кроці, є старшою (першою) цифрою переведеного числа. Результатом переведення такого числа до *q*-тої системи числення є запис останньої частки і всіх остач у зворотному порядку.

Наприклад, переведення числа 133 з десяткової системи числення до вісімкової виконується у такий спосіб:

133 : 8 = 16 (**5**) ↑
16 : 8 = **2** (**0**)
→

Результат: $\boldsymbol{133_{10} = 205_8}$.

Остачі від ділення записують в дужках після часток. Остання частка є старшою цифрою вісімкового числа, до якого дописуються решта остач у порядку, зворотному до порядку її обчислення.

Переведемо те саме число 133 до двійкової системи числення:

133 : 2 = 66 (**1**)
66 : 2 = 33 (**0**)
33 : 2 = 16 (**1**)
16 : 2 = 8 (**0**)
8 : 2 = 4 (**0**)
4 : 2 = 2 (**0**)
2 : 2 = **1** (**0**)

Результат: $\boldsymbol{133_{10} = 10000101_2}$.

Переведення числа 133 до шістнадцяткової системи числення має вигляд:

133 : 16 = **8** (**5**)

Результат: $\boldsymbol{133_{10} = 85_{16}}$.

Переведення десяткового дробу виконується послідовним множенням дробу та подальших результатів на основу системи *q*. Множення виконується до знаходження нульової дробової частини або до обчислення числа із заданою точністю. Запис у прямому порядку всіх цілих частин добутку дає зображення даного дробу в *q*-тій системі числення.

Для прикладу переведемо десяткове число 0.125 почергово до двійкової, вісімкової та шістнадцяткової систем числення:

✓до двійкової:

0.125 · 2 = **0**.25
0.25 · 2 = **0**.5
0.5 · 2 = **1**

Результат: $\boldsymbol{0.125_{10} = 0.001_2}$;

✓до вісімкової:

0.125 · 8 = **1**

Результат: $\boldsymbol{0.125_{10} = 0.1_8}$;

✓до шістнадцяткової:

0.125 · 16 = **2**

Результат: $\boldsymbol{0.125_{10} = 0.2_{16}}$.

А тепер наведемо приклад переведення десяткового числа з цілою та дробовою частинами 122.6 до двійкової системи числення з точністю у шість значущих цифр дробової частини:

Ціла частина	*Дробова частина*
122 : 2 = 61 (**0**) ↑	0.6 · 2 = **1**.2 ↓
61 : 2 = 30 (**1**)	0.2 · 2 = **0**.4
30 : 2 = 15 (**0**)	0.4 · 2 = **0**.8
15 : 2 = 7 (**1**)	0.8 · 2 = **1**.6
7 : 2 = 3 (**1**)	0.6 · 2 = **1**.2
3 : 2 = **1** (**1**) →	0.2 · 2 = **0**.4

Результат: $\boldsymbol{122.6_{10} = 1111010.100110_2}$.

Отже, при переведенні цілої частини числа остачі, отримані в результаті послідовного ділення часток, є цифрами цілої частини числа у новій системі числення. Остача, обчислена на останньому кроці, є старшою (першою) цифрою переведеного числа. А при переведенні дробової частини числа цілі частини чисел, які отримують при множенні, не беруть участі у наступних множеннях. Вони є цифрами дробової частини результату. Значення першої цілої частини є першою цифрою після десяткової крапки переведеного числа. Якщо при переведенні дробової частини отримуємо періодичний дріб, необхідно здійснити округлення, керуючись заданою точністю обчислень.

Переведення десяткових чисел до вісімкової системи числення здійснюється аналогічно. В якості прикладу переведемо вищенаведене число 122.6 до вісімкової системи числення:

Ціла частина	*Дробова частина*
122 : 8 = 15 (**2**) ↑	0.6 · 8 = **4**.8 ↓
15 : 8 = **1** (**7**) →	0.8 · 8 = **6**.4
	0.4 · 8 = **3**.2
	0.2 · 8 = **1**.6
	0.6 · 8 = **4**.8

Результат: $\boldsymbol{122.6_{10} = 172.463146_8}$.

Відповідне переведення десяткового числа 122.6 до шістнадцяткової системи числення здійснюється у такий спосіб:

Ціла частина	*Дробова частина*
122 : 16 = **7** (**10=A**)	0.6 · 16 = **9**.6 ↓
	0.6 · 16 = **9**.6
	0.6 · 16 = **9**.6
	0.6 · 16 = **9**.6

Результат: $\boldsymbol{122.6_{10} = 7A.9999_{16}}$.

Переведення вісімкового числа до двійкової системи числення і навпаки здійснюються за допомогою табл. 2.2. Для цього потрібно виписати відповідні двійкові ***тріади*** усіх вісімкових цифр числа, розпочинаючи зі старшого розряду, наприклад:

✓ $237_8 = 10\ 011\ 111_2$

✓ $504_8 = 101\ \ 000\ 100_2$

✓ $145.26_8 = 001\ 100\ 101.010\ 110 = 1100101.01011_2$

Початкові й кінцеві незначущі нулі можна вилучити.

Для переведення двійкового числа до вісімкової системи числення потрібно виокремити тріади бітів спочатку цілої частини справа наліво, а потім – дробової частини зліва направо, доповнюючи їх, за потреби, незначущими нулями. Потім кожну тріаду слід замінити на відповідну вісімкову цифру, наприклад:

✓ $11000100_2 = 011\ 000\ 100 = 304_8$

✓ $1011.11101_2 = 001\ 011.111\ \ 010 = 13.72_8$

Таблиця 2.2

Відповідність чисел різних систем числення

Десяткові числа	Двійкові числа	Вісімкові числа	Шістнадцяткові числа
0,0625	0,0001	0,04	0,1
0,125	0,001	0,1	0,2
0,25	0,01	0,2	0,4
0,5	0,1	0,4	0,8
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Переведення шістнадцяткових чисел до двійкової системи числення та навпаки здійснюється за допомогою ***тетрад*** (від грецької τετραζ – чотири), тобто кожен шістнадцятковий розряд подається чотирма двійковими. Правила переведення є аналогічні до правил для вісімкової системи числення, наприклад:

✓ $1AF8_{16} = 0001\ 1010\ 1111\ 1000_2$

✓ $14.28_{16} = 0001\ 1010.0010\ 1011_2$

✓ $1100100001011_2 = 1\ 1001\ 0000\ 1011 = 190B_{16}$

✓ $1011010.01011_2 = 101\ 1010.0101\ 1000 = 5A.58_{16}$

Отже, у різноманітних перетворюваннях головну роль відіграє двійкова система числення, а вісімкова та шістнадцяткова є допоміжними, тобто їх можна розглядати як скорочений запис двійкових чисел.

Основами цих систем є цілі степені числа $2^3 = 8$, $2^4 = 16$. Шістнадцяткова система числення широко використовується програмістами, оскільки подання чисел у цій системі є компактнішим, аніж у двійковій чи вісімковій, а переведення з цієї системи до двійкової та навпаки виконується доволі просто.

Висновок: знання систем числення та кодування (перекодування) алфавітно-цифрової інформації потрібні майбутнім ІТ-фахівцям для кращої уяви про процеси цифрової обробки інформації в ЕОМ, а також для формування фундаментальних засад подальшого ознайомлення з мовами машинного рівня (асемблер). Необхідність знань систем числення для інженера-програміста можна прирівняти до необхідності знань алфавіту для письменника.

Література

1. C++. Основи програмування. Теорія та практика : підручник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін.]; за ред. О. Г.Трофимнко. – Одеса: Фенікс, 2010. – 544 с.
2. Prometheus. Курс «Основи програмування» (Лекція 1: Техноманія навколо). [Електронний ресурс]. – Доступний з https://edx.prometheus.org.ua/ courses /KPI/ Programming101/2015_T1
3. C++. Основи програмування. Теорія та практика : підручник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко, Л. М. Буката та ін.]; за ред. О. Г.Трофимнко. – Одеса: Фенікс, 2010. – 544 с.

Запитання до лекції

1. Що таке система числення? Яка різниця між позиційними і непозиційними системами числення?
2. Розкрийте принцип кодування інформації. Які є одиниці вимірювання інформації?
3. Опишіть основні системи числення.
4. Як здійснюється переведення чисел з однієї системи числення в іншу? Наведіть приклади.

ЛЕКЦІЯ 3.
ОГЛЯД МОВ ПРОГРАМУВАННЯ

3.1. Загальні відомості про мови програмування. Мови низького рівня.
3.2. Огляд основних мов програмування високого рівня.

3.1. Загальні відомості про мови програмування. Мови низького рівня

Мова програмування – це система позначень для опису алгоритмів та структур даних, або певна штучна формальна система, засобами якої можна виражати алгоритми. Алгоритмічну мову програмування визначає набір *лексичних (алфавітних), синтаксичних і семантичних правил*, що задають зовнішній вигляд коду та логіку програми.

Алфавіт – це фіксований для конкретної мови набір основних символів (літер абетки) та основних операторів.

Синтаксис – це правила побудови виразів, які дозволяють визначити правильність тієї чи іншої комбінації символів.

Під *семантичними* правилами слід розуміти значення лексичного представлення мови для ЕОМ.

Як вже відомо, мови програмування бувають ***низького*** та ***високого рівня***.

На початку епохи зародження обчислювальної техніки основними засобами програмування були машинні коди, а основним носієм інформації – перфокарти і перфострічки, з допомогою яких відбувалось введення програми в ЕОМ. Перехід до символьного кодування машинних команд ознаменувався початком програмування мовою асемблер (перший вагомий крок в програмуванні). Власне мова ***асемблер*** (assembly language, або assembler) є мовою програмування низького рівня. Словосполучення «низького рівня» в назві мови насправді жодним чином не оцінює мову. Низькорівневість означає, що ця мова є більш ближчою до машинного коду, та, відповідно, є зрозумілішою для комп’ютера.

```
Assembler
 00428F0F 00                    add   %dl,-0x77(%ebp)
 unit1.pas:39                   begin
 00428F10 55                    push  %ebp
 00428F11 89e5                  mov   %esp,%ebp
 00428F13 83ec08                sub   $0x8,%esp
 00428F16 8945f8                mov   %eax,-0x8(%ebp)
 00428F19 8955fc                mov   %edx,-0x4(%ebp)
 unit1.pas:40                   ShowMessage('Foo');
 00428F1C b8f8705700            mov   $0x5770f8,%eax
 00428F21 e8ea6c1000            call  0x52fc10 <SHOWMESSAGE>
 unit1.pas:41                   end;
 00428F26 c9                    leave
 00428F27 c3                    ret
 00428F28 0000                  add   %al,(%eax)
 00428F2A 0000                  add   %al,(%eax)
```

Рисунок 3.1 – Порівняння машинного коду та коду мовою асемблер

Програми на асемблері характеризуються високою швидкодією та малими обсягами використаної пам’яті. У зв’язку з складністю програмування мовами низького рівня, програмісти, які ними володіють, користуються високим попитом.

Наряду з основними перевагами, мови програмування низького рівня володіють низкою недоліків:

✓програміст, що працює з асемблером, має мати хорошу кваліфікацію, добре розуміти будову мікропроцесорної системи, для якої створює програму (знати особливості роботи процесора ПК, під який створюється програма).

✓асемблер є машино-залежною мовою (програму не можна перенести на комп'ютер або пристрій з іншим типом процесора).

✓розробка великих і складних програм потребує значних затрат часу.

Мови низького рівня, як правило, використовують для написання невеликих системних програм, драйверів пристроїв, модулів приєднання нестандартного обладнання, програмування спеціалізованих мікропроцесорів, коли найважливішими вимогами є компактність, швидкодія і можливість прямого доступу до апаратних ресурсів.

На відміну від мов низького рівня, ***високо-рівневі мови*** певною мірою є машинно-незалежними. Вони полегшують роботу програміста і підвищують надійність (безпомилковість) створених програм.

Як вже було зазначено, мови високого рівня значно зрозуміліші для людини. Проте для того, щоб код такої мови став зрозумілим для

ЕОМ (був трансформований в машинний код), необхідно передбачити процес ***трансляції***.

За принципом дії розрізняють два типи трансляторів: *компілятори* та *інтерпретатори*.

Інтерпретатори працюють як синхронні перекладачі (поетапно). Вони беруть по черзі усі оператори програми, транслюють їх в машинний код (або в якийсь проміжний код, близький до машинного коду) і виконують його. Основним недоліком інтерпретації є те, що у випадку повторення програмного коду в різних місцях програми, інтерпретатор цього не бачитиме. Кожного разу, коли інтерпретатор транслюватиме аналогічний код, він буде бачити його вперше. В результаті вихідний машинний код програми стає громіздким.

Компілятори обробляють програму в кілька кроків. Спочатку компілятор кілька разів переглядає вихідний текст (зазвичай він називається вихідним кодом), знаходять ідентичні місця, виконує перевірку на відсутність помилок синтаксису і внутрішніх суперечностей, і лише потім перекладає текст в машинний код. В результаті програма стає компактною та ефективною.

Якщо програма написана мовою на основі інтерпретації, вона виконуватиметься тільки на тому комп'ютері, де попередньо встановлений інтерпретатор (він бере участь у виконанні програми). Програми, написані мовами на основі компіляції, не потребують попереднього встановлення компілятора. В результаті компіляції завжди буде отримано машинний код, який працюватиме на будь-якій платформі. Такі мови програмування називають кросплатформними.

Кросплатформність – здатність програмного забезпечення працювати більш ніж на одній платформі або операційній системі.

Розглянемо, як поділяють мови високого рівня за технологією реалізації (технологією програмування):

- ✓ процедурно-орієнтовані;
- ✓ об'єктно-орієнтовані;
- ✓ проблемно-орієнтовані (логічні).

Процедурне програмування – головним елементом алгоритму є підпрограма, яка створена з метою виконання певної процедури. Програма розглядається як набір підпрограм, що доповнюють одна одну. До основних мов процедурного програмування відносять C, Pascal, Basic.

Об'єктно-орієнтоване програмування – це технологія створення складного програмного забезпечення, заснованого на представленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного класу, а класи утворюють ієрархію із спадкоємством властивостей. Проте, для написання програмної логіки всередині класів, в більшості випадків використовується процедурне програмування. Основна перевага ООП – це значне спрощення процесів створення та модифікації програмних систем. Значно простіше маніпулювати сотнею об'єктів, кожен з яких сам відповідає за свою поведінку і обробку даних, пов'язаних з ним, ніж тисячами функцій, розкиданих у різних модулях.

Проблемно-орієнтоване програмування (можна зустріти назву декларативне програмування) – на відміну від алгоритмічних мов, де потрібно описати порядок вирішення задачі (алгоритм), у декларативних мовах потрібно лише описати постановку задачі за допомогою фактів і правил. Пошук рішень бере на себе мова. Сформувавши базу фактів і правил, можна легко створити систему, яка приймає рішення.

3.2. Огляд основних мов програмування високого рівня

Перехід від програмування виключно низько-рівневими мовами до застосування мов високого рівня відбувся в 1956 році, коли в компанії IBM групою розробників на чолі з Джоном Бекуса була розроблена перша високо-рівнева мова ***Fortran***. Особливістю цієї мови є те, що транслятор розумів формули, записані в формі, наближеній до математичної. Внаслідок цього мова отримала назву Fortran – від слів formula та translation. До речі, розробники мови ставили перед собою основну задачу: одержання високоефективного машинного коду в результаті компіляції програми, що наклало відбиток на зниженні зручності використання мови. Для цієї мови було створено велику кількість бібліотек (набір готових і доступних для загального застусування програм), починаючи від статистичних комплексів і закінчуючи пакетами управління супутниками.

На етапі зародження високо-рівневі мови піддавались нищівній критиці. Перш за все, це було пов’язано з необхідністю використання проміжної ланки – компіляції. Наявність компілятора значно знижувала швидкодію програми. Для вирішення нескладних завдань програмісти використовували асемблер, що дозволяло отримати код,

який працюватиме швидше, ніж код отриманий в результаті компіляції.

Проте через деякий час з'явилося розуміння того, що реалізація великих та складних проектів неможлива без застосування мов високого рівня. Потужність обчислювальних машин зростала, а разом з тим зростала ефективність і швидкодія програм, отриманих в результаті компіляції. Переваги мов високого рівня ставали очевидними, що спонукало до розроблення нових, більш досконалих мов.

Cobol – це компільована мова для застосування в економічній галузі і вирішення бізнес-завдань, розроблена на початку шістдесятих років XX століття. Вона вирізняється "багатослівністю", деякі її оператори виглядають як звичайна мова. На цій мові створено дуже багато додатків, які використовуються і сьогодні.

Algol – компільована мова, створена в 1960 році, щоб замінити Fortran. У зв'язку із складністю структури вона не отримала широкого розповсюдження. В 1968 році створена версія Алгол-68, яка за своїми можливостями випереджає навіть деякі сучасні мови програмування. На жаль, у зв'язку з відсутністю ефективних комп'ютерів на момент створення мови, для неї не вдалося вчасно створити хороших компіляторів. Найбільш популярним представником «алгол-подібних» мов, створених пізніше, є Pascal.

PL/1 – мова, створена в 1964 році компанією IBM, що отримала назву Programming Language One. При створенні мови було перейнято все найкраще з мов Fortran, Cobol та Algol. У ній було реалізовано багато унікальних рішень, користь від яких вдалося оцінити тільки через три десятиліття, в епоху потужних програмних систем. Ця мова і сьогодні підтримується компанією IBM за рахунок присутності унікальних можливостей, якими не володіють навіть деякі сучасні мови.

Pascal – мова програмування, створена в кінці сімдесятих років швейцарським ученим Ніколасом Віртом. Має багато спільного з Algol. Pascal став прототипом сучасних мов програмування, встановивши багато правил і обмежень та, водночас, розширивши можливості. Мову було створено для навчальних цілей, проте з часом вона перетворилась в потужний інструмент професіоналів і стала першою мовою, що набула широкого розповсюдження. В мові вперше впроваджена сувора перевірка типів (суворо типізована мова), що дозволило в подальшому уникати багатьох помилок під час компіляції коду.

Однією з найбільш вдалих версій цієї мови стала версія Turbo Pascal, яка об'єднує в собі редактор тексту і високоефективний компілятор (розробка американської фірми Borland). Версія Turbo Pascal 7 дозволяла швидко і ефективно створювати програми для операційної системи MS–DOS. В подальшому, для створення програм під операційну систему Windows, було розроблено мову, яка отримала назву ***Delphi*** і є більше відомою як середовище візуального проектування.

Мова стала фундаментом для створення *Modula-2*, *Modula-3*, *Oberon* та *Oberon-2* з елементами об'єктно-орієнтованості.

Basic (Beginners ALL Purpose Symbolic Instruction Code – універсальна мова символьних команд для початківців). Для цієї мови розроблено і компілятори, і інтерпретатори. Мова проста для навчання програмуванню, тому набула широкого розповсюдження. Її створили в шістдесятих роках працівники Дартмудського коледжу в США Джон Кемені і Томас Курц.

Ще однією вдалою версією цієї мови є *QBasic* (Quick Basic), яку, свого часу, відносили до найкращих засобів створення програм під MS–DOS.

Подальшим втіленням основних парадигм мови стала версія *Visual Basic*, яка дозволила впровадити елементи візуального програмування для створення графічних об'єктів (вікна, кнопки) під операційну систему Windows. Ці можливості стали ключовими для використання мови в багатьох додатках компанії Microsoft для запису макрокоманд (макросів).

В загальному мову програмування Basic було створено для перших персональних комп'ютерів. Компілятор цієї мови включав в себе і простий редактор тексту, і транслятор, який займав лише 8кБ пам'яті.

С-подібні мови. У 1972 році було створено мову програмування С. Спочатку мова створювалась для розробки операційної системи UNIX. Вона дозволяє працювати з даними практично так же ефективно, як на асемблері, надаючи при цьому структуровані конструкції і абстракції високого рівня (структури і масиви). Саме з цим пов'язана величезна популярність мови. Варто зазначити, що компілятор мови С дуже слабко контролює типи, тому дуже легко написати технічно абсолютно правильну, але логічно помилкову програму. Мова С є суто процедурною мовою програмування.

У 1986 році Б'ярн Страуструп створив першу версію мови ***С++***, додавши в мову С об'єктно-орієнтовані риси, взяті з Simula, і виправивши деякі помилки та невдалі рішення мови. Мова стала основою для розробки сучасних великих і складних проектів. На думку автора мови, відмінність між ідеологією С та С++ полягає приблизно в наступному: програма на С відображає «спосіб мислення процесора» (процедурність), а С++ – спосіб мислення програміста. Клас є ключовим поняттям С++. Опис класу містить опис даних, потрібних для подання об'єктів цього типу, і набір операцій для роботи з подібними об'єктами.

С#. У 1999-2000 роках корпорацією Microsoft була створена мова С#. Вона не має жодного відношення до мов С++ та С та певною мірою схожа на Java (створювалась як альтернатива). Мова С#, як, власне, і мова Java є кросплатформною (дозволяє реалізовувати програмний код на різних платформах) та об'єктно-орієнтованою.

Мови Ada і Ada 95. У 1983 році під егідою Міністерства Оборони США була створена мова Ada. Мова передбачала виявлення великої кількості помилок на етапі компіляції. У 1995 році був прийнятий стандарт мови Ada 95, який розвинув попередню версію, додавши в неї об'єктно-орієнтованість. Обидва варіанти мови не отримали широкого використання у масштабних проектах, окрім військових. Основною причиною є складність мови і досить громіздкий синтаксис.

Об'єктно-орієнтовані мови. Як вже відомо, найбільш розповсюдженими об'єктно-орієнтованими мовами програмування є С++, С#, Python, Java тощо. Проте, крім згаданих мов, існує ціла низка менш популярних. Розглянемо їх коротку характеристику.

Simula – перша об'єктно-орієнтована мова (1967). Ця мова була призначена для моделювання різних об'єктів і процесів. Об'єктно-орієнтовані риси з'явилися в ній саме для опису властивостей модельних об'єктів.

Smalltalk – принесла справжню популярність об'єктно-орієнтованому програмуванню (1972). Мова призначалась для проектування складних графічних інтерфейсів і була першою по-справжньому об'єктно-орієнтованою мовою. Великим недоліком Smalltalk є великі вимоги до пам'яті і низька продуктивність отриманих програм (не дуже вдала реалізація об'єктно-орієнтованих особливостей). Популярність мов С++ і Ada 95 пов'язана саме з тим, що об'єктно-орієнтованість реалізована без істотного зниження продуктивності.

Swift – багатопарадигмова компільована мова програмування, розроблена компанією Apple, щоб співіснувати з Objective C і бути стійкішою до помилкового коду (2014). Swift успадкувала найкращі елементи мов C та Objective-C. За заявою представників Apple, код Swift виконується в 1,3 рази швидше, аніж код на Objective-C.

Eiffel – мова з дуже гарною реалізацією об’єктно-орієнтованості (1986).

Скриптові мови. Останнім часом, у зв’язку з розвитком інтернет-технологій, широким поширенням високопродуктивних комп’ютерів і рядом інших чинників, набули поширення так звані скриптові мови. Характерними особливостями цих мов є їх інтерпретованість (компіляція або неможлива, або небажана), простота синтаксису, легка розширюваність. Далі перерахуємо основні мови скриптового типу.

JavaScript – мова створена в компанії Netscape Communications в якості мови для опису складної поведінки веб-сторінок. Спочатку називалась LiveScript. Причиною зміни назви став маркетинговий хід (до розробників мови Java не має відношення). Інтерпретується браузером під час відображення веб-сторінки. За синтаксисом схожа з Java та віддалено з C / C++.

VBScript – мова, створена корпорацією Microsoft як альтернатива JavaScript. Має схожу область застосування. Синтаксично схожа з мовою Visual Basic. Як і JavaScript, користовується браузером при відображенні веб-сторінок.

Perl – створювалась в якості допомоги системним адміністраторам операційної системи Unix для обробки різного роду текстів і виділення потрібної інформації. Застосовується при обробці текстів, а також для динамічної генерації веб-сторінок на веб-серверах.

Python – об’єктно-орієнтована мова програмування. За структурою і областю застосування близька до Perl. Набула широкого розповсюдження за рахунок кросплатформності. Вважається однією з найпростіших сучасних мов, тому знайшла широке розповсюдження серед початківців та професіоналів.

PHP – скриптова мова програмування, створена для генерації HTML-сторінок на стороні веб-сервера (back end). PHP є однією з найпоширеніших мов, що використовуються у сфері веб-розробок. PHP інтерпретується веб-сервером у HTML-код, який передається на сторону клієнта (front end). На відміну від скриптової мови JavaScript,

користувач не бачить PHP-коду, тому що браузер отримує готовий html-код. Це перевага з точки зору безпеки, але погіршення інтерактивності сторінок.

HTML – мова розмітки гіпертекстових документів. Загальновідома мова для оформлення (верстки) веб-документів, дуже проста і містить елементарні команди форматування тексту, додавання малюнків, виставлення шрифтів і кольорів, організації посилань і таблиць. Документ HTML опрацьовується браузером та відтворюється на екрані у звичному для користувача вигляді. Мова HTML, разом із каскадними таблицям стилів ***CSS*** (мова для опису зовнішнього вигляду сторінки), – основи технології побудови веб-сторінок. Певною мірою мови PHP і JavaScript – це додатки, елементи яких додаються до веб-сторінок з метою надання динамічності та сучасності.

Мови обробки даних. Всі розглянуті мови є мовами загального призначення (універсальні) в тому сенсі, що вони не орієнтовані і не оптимізовані під використання будь-яких специфічних структур даних або на застосування в будь-яких специфічних областях. До мов для специфічного застосування належать: ***APL*** (1957, математична обробка даних), ***Snobol і Icon*** (1962 і 1974, обробка рядків), ***SETL*** (1969, операції над множинами), ***Lisp***. На детальному описі зазначених мов детально зупинятись не будемо, окрім останньої. У 1958 році з'явився мова Lisp – мова для обробки списків. Мова отримала досить широке поширення в системах штучного інтелекту. Має кілька нащадків: *Planner* (1967), *Scheme* (1975), *Common Lisp* (1984). Багато основних рис мови успадковані сучасними мовами функціонального програмування.

Go – розробка Google, яка використовується всередині компанії. Відрізняється надійністю і стабільністю.

Мови для математичних розрахунків.

R – мова програмування і програмне середовище для статистичних обчислень, аналізу та представлення даних в графічному вигляді. R має значні можливості для здійснення статистичного аналізу, включаючи лінійну і нелінійну регресію, класичні статистичні тести, аналіз часових рядів (серій), кластерний аналіз і багато іншого.

Matlab – пакет прикладних програм для чисельного аналізу, а також мова програмування, що використовується в даному пакеті.

Мови паралельного програмування. Більшість комп'ютерних архітектур і мов програмування орієнтовані на послідовне виконання операторів програми. Сьогодні також існують програмно-апаратні комплекси, які дозволяють організувати паралельне виконання різних частин одного і того ж обчислювального процесу. Для програмування таких систем необхідна спеціальна допомога мов програмування, таких як Occam, Linda.

Всі мови програмування, про які йшла мова раніше, мають одну загальну властивість: вони ***імперативні***. Це означає, що програми, написані цими мовами, в кінцевому підсумку являють собою покрокове вирішення того чи іншого завдання (алгоритмічне програмування). Проте в світі програмування із розвитком технологій штучного інтелекту розвиваються мови, які дозволять описувати лише постановку проблеми, а її вирішення реалізується компілятором на основі закладених математичних функцій або формул математичної логіки (***декларативні***). Існує два основні підходи до розвитку цієї ідеї: функціональне та логічне програмування.

Мови програмування баз даних. Ця група мов відрізняється від алгоритмічних мов перш за все завданнями. База даних – це файл (або група файлів), який представляє упорядкований набір записів, що мають єдину структуру і організовані за єдиним шаблоном (як правило, в табличному вигляді). При роботі з базами даних найчастіше потрібно виконувати такі операції: створення, модифікація властивостей, видалення таблиць в базі даних; пошук, відбір, сортування інформації за запитами користувача; додавання нових записів; модифікація наявних записів; видалення наявних записів.

Перші бази даних з'явились через потребу в обробленні великих масивів інформації і створенні груп записів за визначеними ознаками. Для цього була створена структурована мова запитів SQL (Structured Query Language). Ця мова дозволяє виконувати ефективну обробку баз даних, оперуючи не окремими записами, а групою записів.

Для управління великими базами даних та їх ефективною обробкою були розроблені СУБД (системи управління базами даних), в основі яких лежить мова SQL. Сьогодні нараховується п'ять основних виробників СУБД: Microsoft (SQL Server), IBM (db2), Oracle, Sybase, Software AG (Adabas). Їх продукти направлені на підтримку одночасної роботи тисяч користувачів в мережі, а бази даних можуть зберігатися на кількох серверах.

Висновок: Знання загальних відомостей про мови та технології програмування є одним із ключових базисних понять, які потрібні фахівцеві для подальшого здобуття фахових компетенцій. А знання основних властивостей різних мов даватиме можливість здійснювати оптимальний вибір конкретної мови для реалізації поставлених завдань.

Література:

1. Список мов програмування – довідник [Електронний ресурс]. – Доступний з https://uk.wikipedia.org/wiki/Список_мов_програмування
2. Еволюція мов програмування [Електронний ресурс]. – Доступний з http://easy-code.com.ua/2012/08/evolyuciya-mov-programuvannya-rizne-programuvannya-statti/
3. Огляд мов програмування високого рівня [Електронний ресурс]. – Доступний з http://ruszura.in.ua
4. Сучасні мови програмування та їх використання [Електронний ресурс]. – Доступний з http://damp.biz/suchasni-movi-programuvannya-ta-yih-vikoristannya-iteach/
5. Мови програмування. Огляд, можливості, переваги, недоліки [Електронний ресурс]. – Доступний з http://damp.biz/movi-programuvannya-oglyad-mozhlivosti-perevagi-nedoliki

Запитання до лекції

1. Що таке мова програмування? Класифікація мов програмування.
2. Опишіть основні відмінності між мовами програмування низького та високого рівня.
3. Які бувають технології реалізації мов програмування високого рівня?
4. Назвіть найпоширеніші мови програмування високого рівня.

ЛЕКЦІЯ 4.
ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ МОВОЮ JAVA

4.1. Огляд об’єктно-орієнтованої мови Java
4.2. Інтегроване середовище розробки

4.1. Огляд об’єктно-орієнтованої мови Java

Попри те, що існують простіші у застосуванні мови із дещо приємнішим синтаксисом, зупинимось на виборі мови програмування Java. Чому саме ця мова? Для відповіді на це запитання розглянемо рейтинг найбільш популярних та затребуваних мов програмування у світі. Зупинимось на перших 15 позиціях. В якості джерела рейтингу взято дані всесвітньовідомого журналу IEEE Spectrum, що видається Інститутом інженерів електротехніки та електроніки (IEEE) (остання редакція рейтингу за 2018 рік).

Рейтинг побудовано за популярністю використання мов програмування для створення Web-додатків, прикладних програм, мобільних застосунків та програмування вбудованих пристроїв.

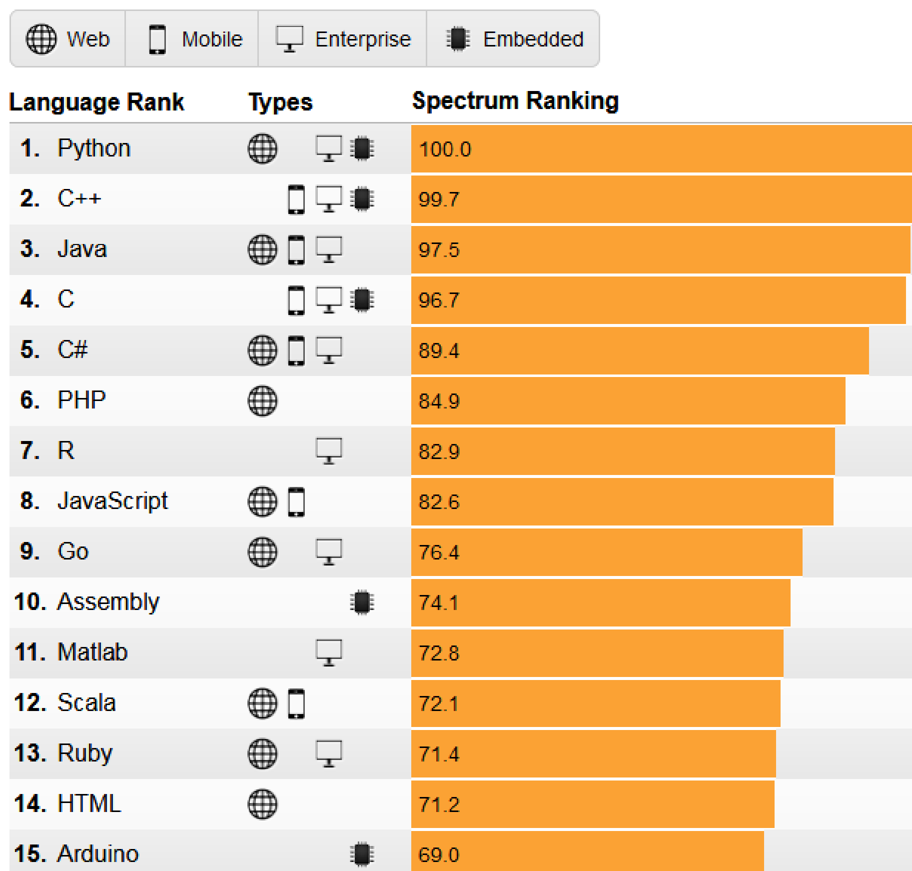

Рисунок 4.1 – Рейтинг мов програмування з даними журналу IEEE

В світі зустрічається низка різноманітних рейтингів, які дещо відрізняються. До прикладу, рейтинг TIOBE, який сформовано на основі суми запитів в пошуковій системі Google, виглядає наступним чином:

Feb 2019	Feb 2018	Change	Programming Language
1	1		Java
2	2		C
3	4	↑	Python
4	3	↓	C++
5	6	↑	Visual Basic .NET
6	8	↑	JavaScript
7	5	↓	C#
8	7	↓	PHP
9	11	↑	SQL
10	20	⇈	Objective-C

Рисунок 4.2 – Рейтинг мов програмування з даними TIOBE (TIOBE Index for February 2019)

Будь-який інший рейтинг також відрізнятиметься від двох попередніх. Це пов’язано з особливостями критеріїв, за якими їх побудовано. Проте всі ці рейтинги об’єднує єдине – незмінна група лідерів, які лише чергуються між собою. Зважаючи на високу затребуваність мови програмування Java, відповідність її характеристик основним парадигмам програмування і, що найважливіше, кросплатформність, саме її обрано для вивчення.

В якості завершального аргументу на рисунку 4.3 представлено приріст середніх зарплат за 5 років у спеціалістів, що програмують різними мовами. Відповідно до наведених статистичних даних, найбільше отримують так звані «Джавісти».

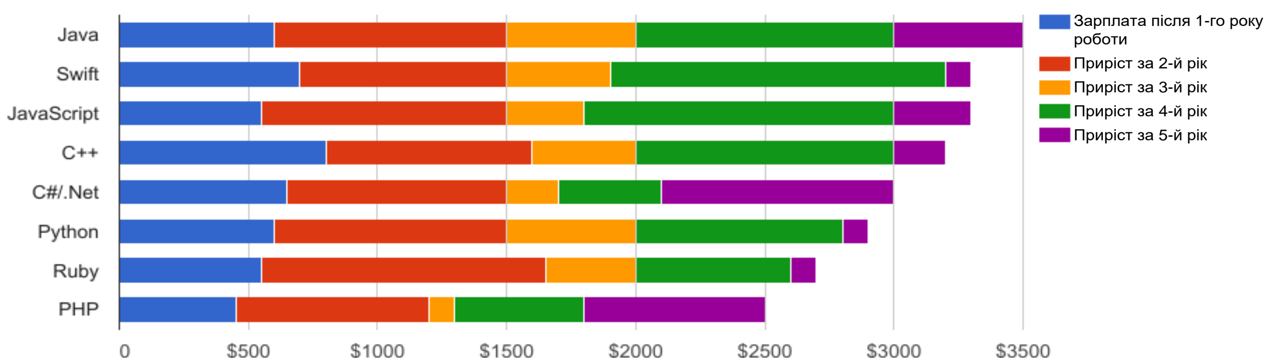

Рисунок 4.3 – Усереднений приріст заробітних плат за 5 років

Об'єктно-орієнтована мова програмування Java розроблена компанією Sun Microsystems у 1995 році. Синтаксис мови багато в чому походить від C та C++ (Java містить всі позитивні аспекти C++).

Мова Java зародилася як частина проекту створення передового програмного забезпечення для різних побутових приладів (в тому числі для кавоварок – звідти і походить логотип мови). Загалом же реалізацію цього проекту було розпочато на мові C++. Вже незабаром виник ряд проблем, що і привели до заміни мови програмування. В процесі реалізації проекту розробники зіштовхнулись з необхідністю використання платформо-незалежної мови програмування, яка б дозволяла створювати програми без необхідності їх перекомпіляції окремо під кожну архітектуру та давала можливість використовувати програму на різних процесорах із різними операційними системами.

Розробники мови Java створювали її під гаслом: «Написав один раз – використав скрізь». Код, написаний цією мовою, може запускатись на будь-якій платформі чи операційній системі. Яким чином досягнуто кросплатформність? Про це далі.

Написані мовою Java програми виконуються у середовищі віртуальної машини Java. Ряд дій, які в C/C++ мають здійснювати програмісти, в Java доручено віртуальній машині. Програма, написана на Java, компілюється у байт-код, який при виконанні інтерпретується віртуальною машиною для конкретної платформи (ось і вся магія кросплатфомності).

Мова має менше низькорівневих можливостей для роботи з апаратним забезпеченням. Для реалізації таких дій передбачена можливість виклику підпрограм, написаних іншими мовами програмування.

Інколи можна зустріти мови програмування, назви яких співзвучні з Java. Декілька слів про це. Після виходу мови на ринок нею зацікавились мегакорпорації, в тому числі і Microsoft, якою, після невдалої спроби поглинути компанію Sun Microsystems, розпочато створення ідентичної мови під назвою J++. Проте роботу над створенням J++ було припинено через судовий позов Sun Microsystems. Після цього на новій платформі від Microsoft – .NET, було випущено J#, яка мала полегшити міграцію програмістів J++ або Java на нову платформу. З часом нова мова програмування з маркетингових міркувань отримала назву C#, попри відсутність жодного відношення до творців мов сімейства C. Отже, зрозуміло, чому мови Java та C# є настільки подібними.

Після багаторічного протистояння з Microsoft в 2009 році Sun Microsystems все ж таки була викуплена компанією Oracle, яка продовжує розвивати мову.

Скриптова мова сценаріїв JavaScript хоча і має схожу назву із Java та багато в чому повторює синтаксис, все ж зовсім не пов'язана з Java сферами застосування, розробниками, реалізацією основних парадигм тощо. Перша назва цієї мови LiveScript, проте з тих самих маркетингових міркувань пізніше пізніше з'явилася сучасна назва.

В процесі еволюції назви випусків мови Java подавалися розробниками дещо по-різному. Перші версії отримали назву JDK 1.0, JDK 1.1 (Java Development Kit). Перші серйозні зміни було впроваджено в версії 1.2, після чого маркетологи вирішили змінити назву – Java2SE (Java-2). Подальші версії 1.3 та 1.4 насправді були модифікаціями Java2. В літературі можуть зустрічатись їх назви в такому вигляді: J2SE 1.3 та J2SE 1.4 (SE – це Standart Editon). Друга версія Java окрім версії SE постачалась на ринок ще в варіанті J2EE – Enterprise Edition (додатки для підприємств) та J2ME – Micro Edition (мобільні додатки)).

Чергова революційна версія 1.5 (зазнала низку змін та доповнень) отримала назву J2SE 5.0, що викликало багато суперечностей (Java2 чи Java5). Зважаючи на це, з назви було вилучено двійку. Відтоді усі чергові версії отримували назви Java SE 5, Java SE 6, Java SE 7, Java SE 8 … Java SE 11 і т. д.

Подібні кардинальні зміни в найменуванні передусім пояснювалися ґрунтовними змінами у можливостях мови. За історію Java найбільш серйозні зміни внесені у версію 1.2 та 1.5. Для кращої уяви про назви версій, згрупуємо їх в таблицю.

Таблиця 4.1

Версії мови Java

Java		Java 2			Java 5	Java 6	Java …
1.0	1.1	1.2	1.3	1.4	1.5	1.6	…
250 класів	500 класів. Початок популярності. Комфортний синтаксис	2300 класів. Потужні версії. Пік популярності. Розробка більшості промислових систем, веб-орієнтованих та мобільних додатків тощо			Понад 3500 класів. Потужність мови зростає, складність написання зменшується. Запозичуються популярні можливості з інших мов		

Як працює Java та в чому полягає особливість компіляції?

Основною задачею розробника на Java є створення вихідного документу (первинний код програми). Файл з кодом програми матиме

розширення .java. Після написання первинного коду відбувається процес його компіляції. Для компіляції файлу необхідно запустити додаток-компілятор javac. За умови використання інтегрованого середовища розробки (про це пізніше), компіляція відбуватиметься автоматично.

Компілятор перевіряє отриманий код на наявність помилок та видає кінцевий результат після перевірки його працездатності та коректності. Результатом роботи компілятора є новий документ з розширенням .class, в якому міститься скомпільований байт-код. Будь-який пристрій, який здатний виконувати Java, зможе транслювати (інтерпретувати) цей файл в такий формат, який потім запускатиметься на ньому. Скомпільований байт-код не залежить від платформи, де реалізується, в цьому і полягає суть кросплатформної особливості Java. Уся магія інтерпретації байт-коду відбувається у віртуальні машині Java (JVM), яка встановлюється та запускається на пристроях виконання програми. Суть інтерпретації байт-коду JVM полягає у його зчитуванні та адаптації під особливості архітектури пристрою (перетворенні у відповідний формат).

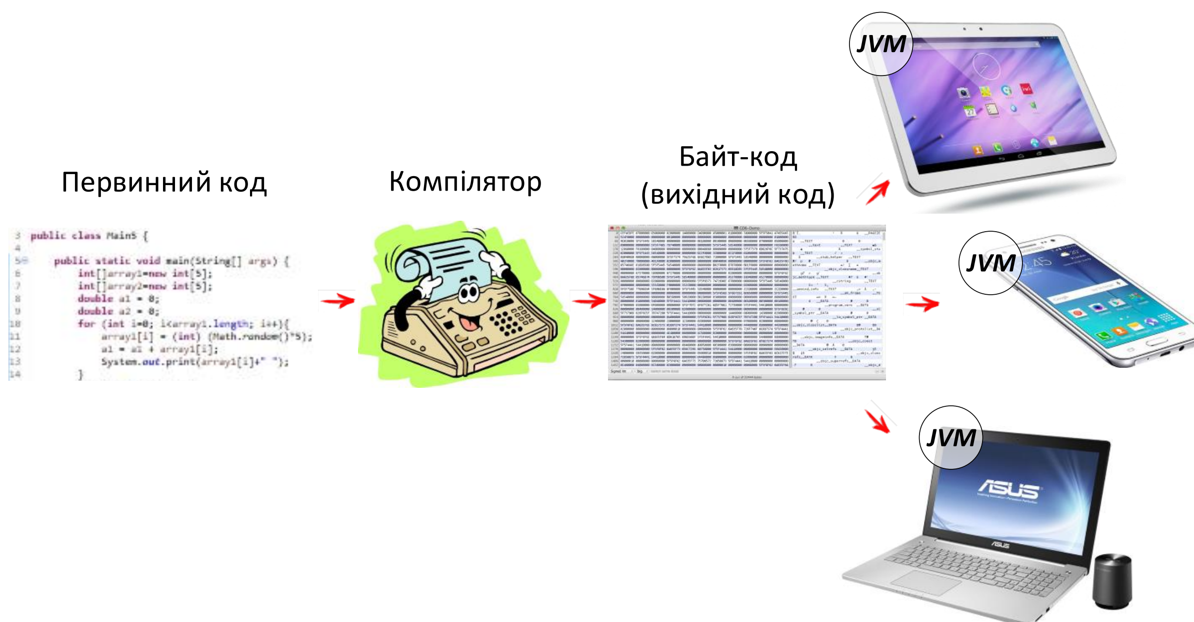

Рисунок 4.4 – Процес компіляції та виконання Java-програми

Отже, зважаючи на короткий опис особливостей платформи Java та її мови, проведемо узагальнення ***основних переваг***:

✓ об’єктно-орієнтованість – в Java все представлено у вигляді об’єктів. За необхідності доповнення, виправлення або зміни логіки програми, це можна зробити «безболісно», оскільки структура коду побудована за об’єктною моделлю (модульний принцип);

✓ кросплатформність (незалежність від платформи) – на відміну від більшості розповсюджених мов програмування, Java-код не компілюється на платформі, де виконується. Компіляція реалізується в байт-код на платформі, де створена програма, а інтерпретація та адаптація коду під особливості платформи відбувається JVM там, де програма виконується;

✓ архітектурна нейтральність – компілятор генерує архітектурно-нейтральні байт-коди, що дозволяє їх використання на різних типах процесорів;

✓ грамотність – Java платформа реалізує низку заходів для виявлення помилок на стадії компіляції та виконання програми;

✓ багатопоточність – дозволяє створювати програми, які виконують множину задач одночасно;

✓ розповсюдженість – висока популярність мови дозволяє використовувати в процесі програмування безліч доступних фреймворків, а також здійснювати вільний пошук варіантів вирішення різноманітних проблем;

✓ управління пам'яттю – Java є унікальною мовою, яка слідкує за чистотою пам'яті, знищуючи об'єкти, на які відсутні робочі посилання (не використовуються програмою). Очищення реалізується за допомогою «збирача сміття», роботу якого розглянемо згодом.

4.2. Інтегроване середовище розробки

Мінімум, необхідний для програмування на Java – це JDK (Java Developer Kit – комплект розробника Java) та звичайний текстовий редактор. Проте, за умови використання текстового редактора для написання кодів, підсвічування синтаксису програми, компіляція та запуск програми здійснюватись не будуть.

Розглянемо спочатку, що таке комплект розробника Java. JDK це набір бібліотек класів, утиліт (генерування коду, генерування документації, автоматична компіляція і т. д.) та документації (допоміжна інформація, що стосується елементів мови – класів, методів тощо) для програмування на Java. Він складається з кількох основних компонентів:

✓ *компілятор javac* – транслює текст програми Java в байт-коди віртуальної машини;

✓ *інтерпретатор java* – запускає програми, відкомпільовані в байт-код. Містить в собі JVM (Віртуальну машину Java);

✓*утиліта appletviewer* – запуск аплетів (графічних програм), які виконуються в браузері. Фактично являє собою браузер, який може запускати тільки аплети;

✓*утиліта javadoc* — призначена для створення документації.

Якщо перед вами поставлена задача лише запуску програми, створеної на Java, то достатньо просто завантажити JRE (Java Runtime Environment). Це мінімальний набір, який дозволяє виконувати програми, написані на Java, проте не містить інструментів для розробки програмного забезпечення.

Як вже було відмічено, для створення програм на Java, достатньо лише JDK та текстового редактора. Проте, якщо користуватися лише ними, то розробка програм буде доволі складною та довготривалою роботою. Для полегшення розробки слугують інтегровані середовища розробки (Integrated Development Environment (IDE) – це комплексні програмні засоби, які дозволяють полегшити розробку та модифікацію тексту програми. Вони містять багато корисних функцій та можливостей, полегшують значну кількість рутинної роботи, яка, зазвичай, є джерелом додаткових помилок.

Інтегровані середовища розробки створені для того, аби максимізувати продуктивність розробника, надавши йому всі інструменти у вигляді однієї програми. Інтегроване середовище розробки надає можливість об'єднати усі процеси створення програмного забезпечення:

✓написання початкового коду;

✓перевірка (редакція) коду на його безпомилковість та працездатність;

✓компіляція коду;

✓автодоповнення коду;

✓розгортання та налагодження програмного забезпечення.

Основною перевагою інтегрованих середовищ розробки є синтаксичний аналіз коду на стадії його редагування, тобто виявлення помилок ще до трансляції коду.

Деякі інтегровані середовища розробки призначені для використання певної мови програмування (або декількох споріднених мов) і надають набір можливостей, які підходять до програмування відповідними мовами. Такими IDE є, наприклад PhpStorm, Xcode, Xojo та Delphi.

Варто зазначити, що існує чимало більш універсальних IDE, які є багатомовними, наприклад Eclipse, IntelliJ IDEA, NetBeans, Microsoft Visual Studio тощо.

Деякі середовища розробки містять або компілятор, або інтерпретатор, або і перше, і друге (наприклад NetBeans та Eclipse), інші ж не містять жодного з них (SharpDevelop та Lazarus).

Таблиця 4.2

Перелік основних IDE та їх відповідність мовам програмування

Універсальні	IntelliJ IDEA,Visual Studio, NetBeans, Eclipse
C/C++	Borland C++, C++ Builder, Code::Blocks, CodeLite, Dev-C++, Oracle Solaris Studio, Ultimate++, Microsoft QuickC
BASIC	Gambas, PowerBASIC, Turbo Basic, Visual Basic, QBasic
Java	MyEclipse, Oracle WebLogic Workshop, IBM WebSphere Studio, BlueJ, DrJava, JDeveloper, JBuilder
Pascal	Delphi, Lazarus, PascalABC.NET, Turbo Pascal, QuickPascal
PHP	Aptana Studio with PHP plugin, Delphi for PHP (RadPHP), Eclipse PDT, Zend Studio, PHP Expert Editor, phpStorm, Dreamweaver
Python	Eric, PyCharm, PyDev, PyScripter, Wing IDE
Ruby	RubyMine

Найпопулярнішим середовищем розробки для Java є ***Eclipse***. Також для програмування на Java часто застосовують ***NetBeans***, роботи над яким фінансує Oracle, та ***IntelliJ IDEA*** компанії JetBrains. Ще одним представником IDE для Java є JDeveloper, розробником якого є все та ж компанія Oracle. Основні налаштування перелічених середовищ найкраще адаптовані під Java, хоча можуть бути використані для програмування іншими мовами. Існує й зворотній зв'язок – низку середовищ розробки для інших мов програмування також можна налаштувати на роботу з Java.

Eclipse (від англійського «затемнення») – вільне модульне інтегроване середовище розробки програмного забезпечення. Розробляється і підтримується Eclipse Foundation і включає проекти, такі як платформа Eclipse, набір інструментів для розробників на мові Java, засоби для управління вихідним кодом, GUI (графічний інтерфейс користувача) тощо.

Інтегроване середовище розробки написане на Java. Найчастіше середовище використовується для розробки мовою Java, проте за допомогою різних плаґінів воно може бути адаптоване для програмування іншими мовами: Ada, C, C++, COBOL, Fortran, Perl, PHP, Python, R, Ruby, Scala тощо. Прикладами різних модифікацій є

Eclipse CDT для C/C++; Eclipse JDT для Java; Eclipse PDT для PHP тощо.

Випущене на умовах *Eclipse Public License*, інтегроване середовище є вільним програмним забезпеченням. Eclipse – це фреймворк для розробки модульних кросплатформних застосунків із низкою особливостей:

✓можливість розробки ПЗ багатьма мовами програмування (рідною є Java);

✓кросплатформність (платформно-незалежний продукт);

✓модульність (призначена для подальшого розширення незалежним розробниками).

З огляду на доступність платформи, у багатьох організаціях Eclipse став корпоративним стандартом для розробки ПЗ на Java. Як вже було зазначено, Eclipse (завдяки тому, що написана на Java) є платформно-незалежним продуктом (виняток становить бібліотека графічного інтерфейсу SWT, яка використовує графічні засоби операційної системи). Це стало додатковою причиною широкого розповсюдження середовища.

Середовище розробки представлено безліччю різноманітних версій, останні з яких представлені в таблиці.

Таблиця 4.3

Останні версії середовища Eclipse

Назва версії	Дата випуску	Номер
Luna	25 червня 2014	4.4
Mars	24 червня 2015	4.5
Neon	22 червня 2016	4.6
Oxygen	червень 2017	4.7
Photon	червень 2018	4.8

Інтерфейс Eclipse умовно поділено на три основні зони: Package Explorer (ліворуч); область написання коду (центральна частина); Console для періодичного виводу та перевірки результатів роботи програми (нижня частина).

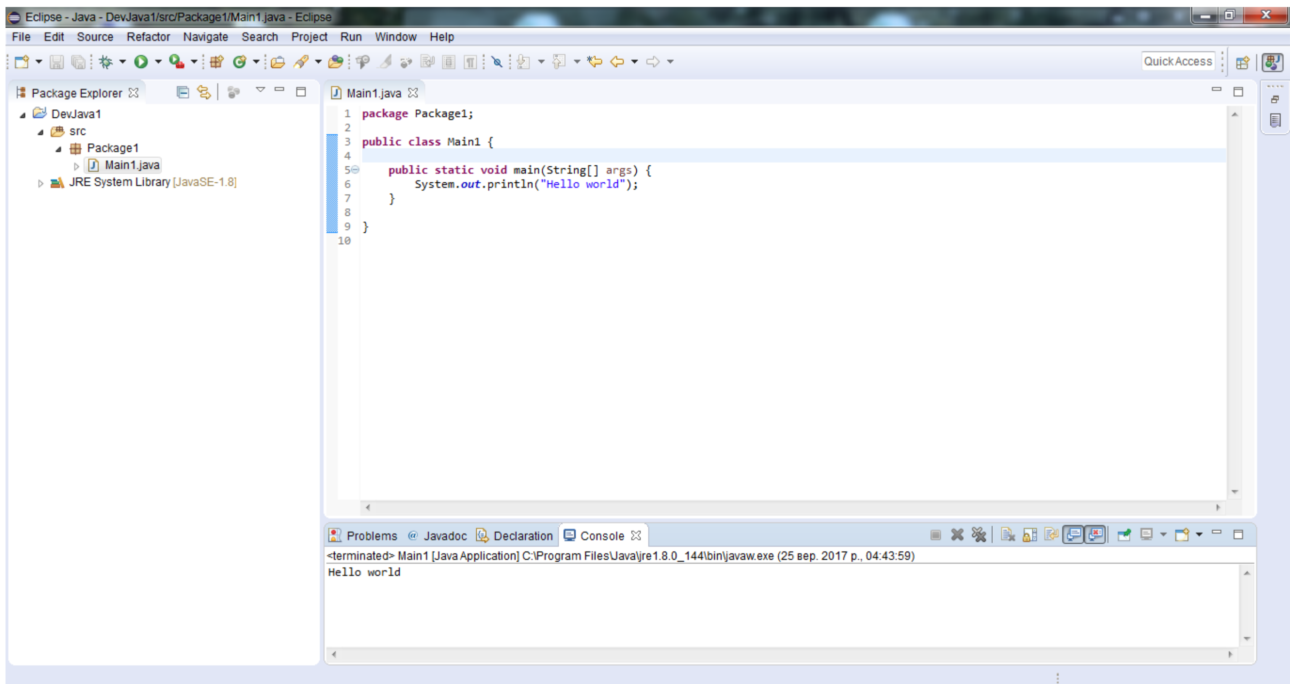

Рисунок 4.5 – Інтерфейс середовища Eclipse

Кореневим каталогом в Package Explorer є папка, яка містить Java проект. В наведеному прикладі це папка з назвою DevJava1, що автоматично зберігається в каталозі комп'ютера, адреса якого зазначається під час завантаження середовища розробки.

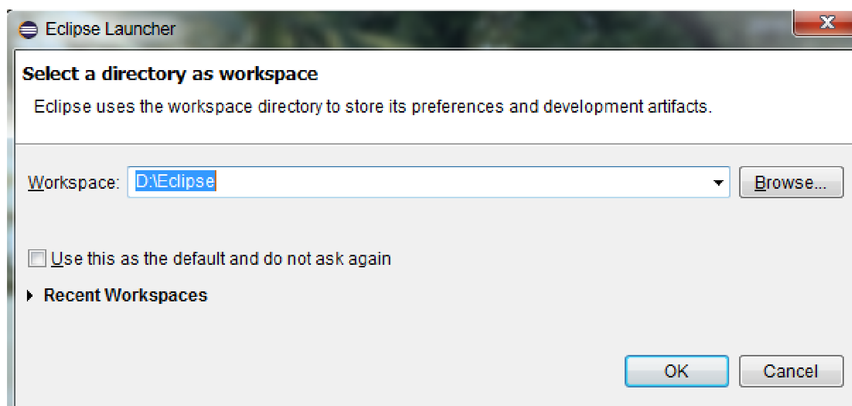

Рисунок 4.6 – Запуск середовища Eclipse (зазначення адреси каталогу, в якому зберігатимуться Java-проекти)

Створення чергового проекту супроводжується його автоматичним наповненням підкаталогом "src" та бібліотекою JRE System Library. Усі подальші роботи за проектом проводяться в межах підкаталогу "src".

Кожен проект містить пакети, які наповнюють класами. Подібна структура відображена на рисунку та повністю відповідає файловій системі комп'ютера. До прикладу, ви створили новий підкаталог (папку) «Eclipse» в кореневому каталозі одного з дисків комп'ютера.

У випадку зазначення адреси цієї папки, під час завантаження інтегрованого середовища розробки усі проекти будуть автоматично збережені в зазначеній дерикторії (розміщення каталогу «DevJava1» матиме адресу D:\Eclipse\DevJava1\...).

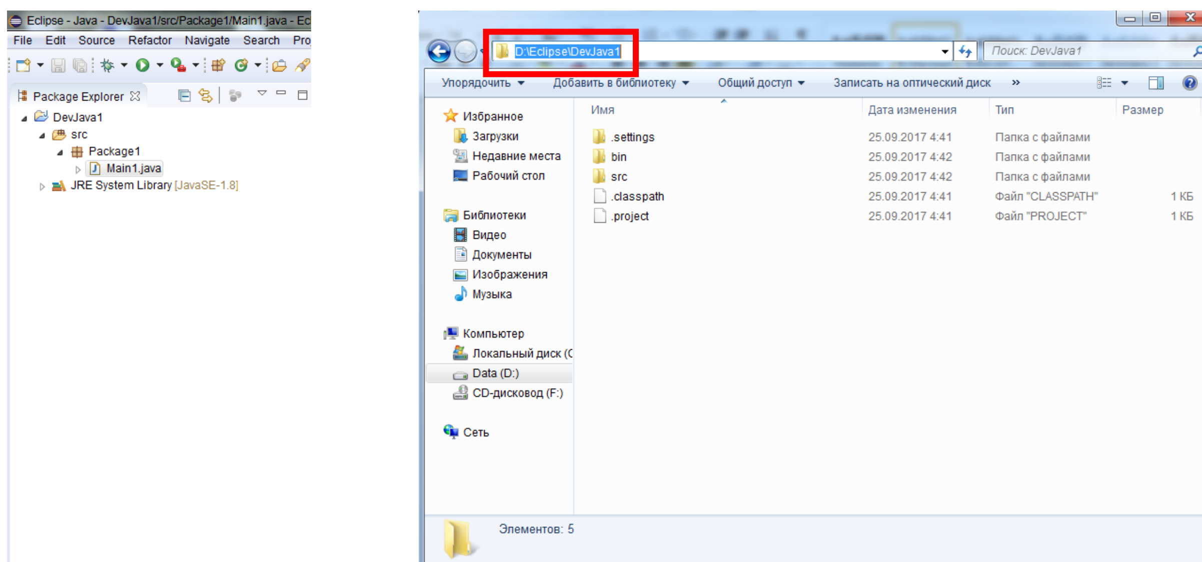

Рисунок 4.7 – Відповідність структури Java-проекту в Eclipse файловій системі ОС

В середовищі Eclipse усі пакети, а відповідно, і класи, розміщено в папці “scr”. Аналогічна ієрархія дублюється в файловій системі ОС. За адресою D:\Eclipse\DevJava1\src\Packege1 зберігається файл Main1.java. Розширення файлу вказує на те, що в ньому зберігається вихідний код програми. У випадку створення в межах Package1 інших класів, вони автоматично зберігатимуться за зазначеною в прикладі адресою під іншим іменем, але з ідентичним розширенням.

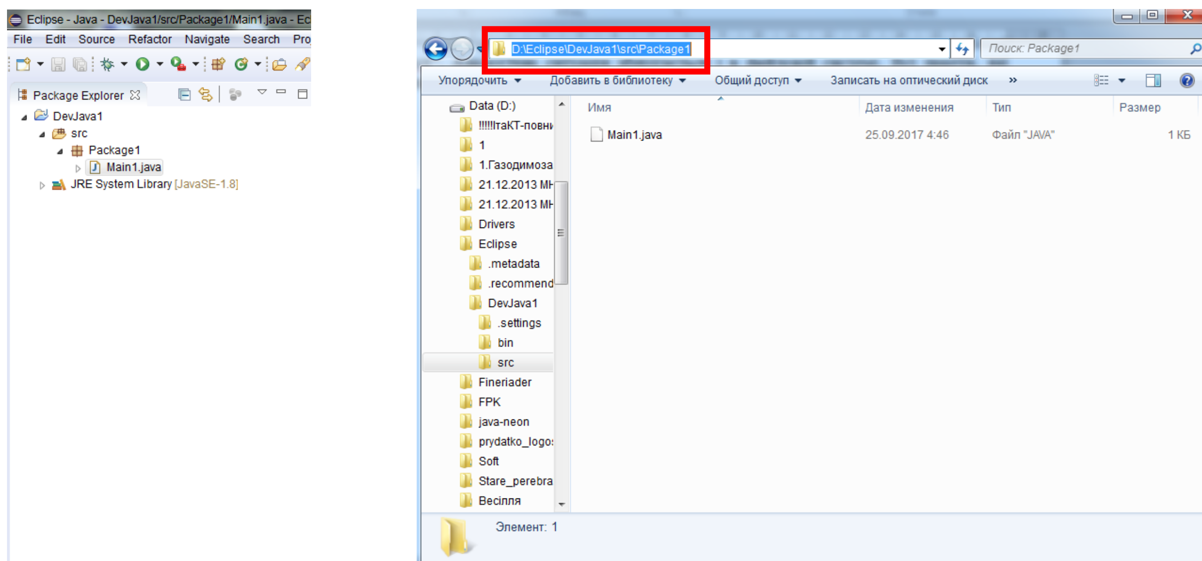

Рисунок 4.8 – Адреса розміщення файлу з вихідним (первинним) кодом програми у файловій системі ОС

Де зберігається відкомпільований байт-код? Для цього ми знову повернемось до підкаталогу «DevJava1». Крім згаданої раніше папки “src”, в підкаталозі міститься папка “bin”. ЇЇ внутрішня структура, назви підкаталогів і файлів повністю відповідають вмісту папки “src”, лише з однією відмінністю – в зазначеному прикладі файл Main1 має розширення .class. Власне, файл Main.class – це і є байт-код.

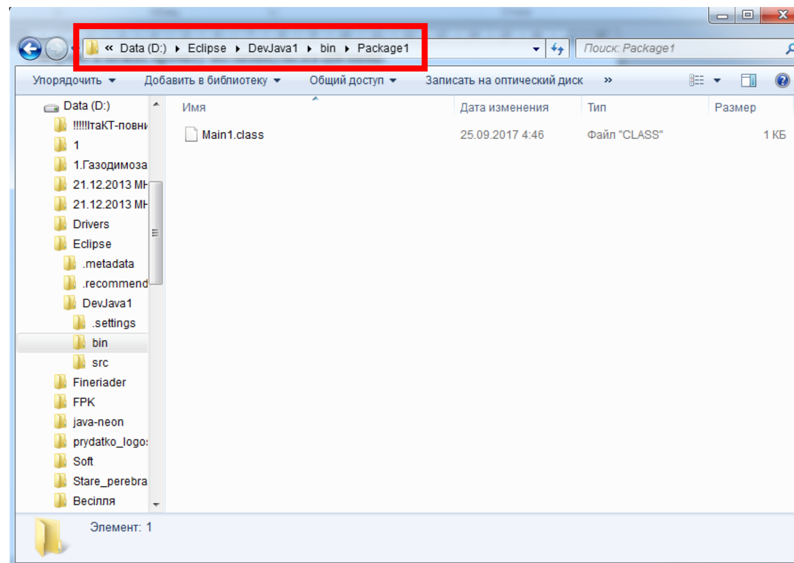

Рисунок 4.9 – Адреса розміщення файлу з байт-кодом програми у файловій системі ОС

Висновок: детальний опис особливостей мови програмування Java та процесів її компіляції необхідний для подальшого розуміння усіх етапів розробки програмного забезпечення. А стисле ознайомлення із можливостями та призначенням інтегрованих середовищ розробки є фундаментальним для їх подальшого використання в навчальному процесі та професійній діяльності.

Література:

1. Рейтинг мов програмування 2018 року від IEEE Spectrum [Електронний ресурс]. – Доступний з https: // spectrum.ieee.org
2. TIOBE Index for February 2019 [Електронний ресурс]. – Доступний з https://www.tiobe.com/tiobe-index/
3. Ріст зарплат з досвідом роботи: аналітика [Електронний ресурс]. – Доступний з https://dou.ua/lenta/articles/salary-n-expierence-growth/
4. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java.
5. Опановуємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java.

6. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

Запитання до лекції

1. Розкрийте особливості мови програмування Java та її основні переваги.
2. Перелічіть основні компоненти комплекту розробника Java (JDK).
3. Що таке інтегроване середовище розробки програмного забезпечення?
4. Які можливості дає розробнику Java інтегроване середовище розробки Eclipse?

ЛЕКЦІЯ 5.
БАЗОВІ ЕЛЕМЕНТИ МОВИ JAVA

5.1. Типи даних та змінні
5.2. Оператори

5.1. Типи даних та змінні

Ще однією важливою характеристикою мови Java є її сувора типізація. Мова про те, що при оголошенні будь-якої змінної необхідно вказувати тип даних. Існують примітивні та об’єктні типи даних. В Java є вісім основних ***примітивних*** типів даних. П'ять із них – цілочисельні (включаючи символьний тип char), два – з плаваючою крапкою і один логічний (булевий) тип даних. Розглянемо окремо кожен із згаданих типів.

Таблиця 5.1

Типи цілочисельних даних та діапазон їх значень

Тип цілочисельних даних	Довжина (в байтах)	Діапазон значень
byte	1	−128..127
short	2	−32768..32767 ($-2^{15}..2^{15}-1$)
int	4	−2147483648..2147483647 ($-2^{31}..2^{31}-1$)
long	8	приблизно $-9.2\cdot10^{18}..9.2\cdot10^{18}$ ($-2^{63}..2^{63}-1$)
char	2	0..65535 ($0..2^{16}-1$)

Тип даних char використовують для запису символів. Усі решта типи даних – для запису цілочисельних значень. Тип int (integer – ціле число) є найбільш вживаним, оскільки займає відносно невеликий обсяг пам’яті попри значний діапазон вмістимих значень.

Таблиця 5.2

Типи дробових даних та діапазон їх значень

Тип дробових даних	Довжина (в байтах)	Діапазон значень
float	4	$-3.4\cdot10^{38}..3.4\cdot10^{38}$
double	8	$-1.8\cdot10^{308}..1.8\cdot10^{308}$

Щодо типів даних із плаваючою крапкою, то найбільш вживаним є double.

Логічний тип даних boolean має два значення: істина або хиба. Його використовують для запису результатів логічних операцій.

Таблиця 5.3

Логічний тип даних

Логічний тип даних	Довжина (в байтах)	Діапазон значень
boolean	не визначено	true, false

Інколи виникає необхідність запису або виводу стрічки значень. Для цього використовують об’єктний тип даних у вигляді екземпляру класу String (про це трішки пізніше).

Оголошення змінної розпочинають з обов'язкового зазначення типу даних, після чого вказують її назву. Оголошення закінчується символом « ; ». Приклад:

```
int volume;
double degree;
char oneChar;
boolean kukuruku;
```

Важливо! Назва змінної повинна починатись з літери, після чого в назві можна застосовувати необмежену кількість цифр та інших літер. Java надає можливість присвоювати назви змінним кирилицею. В мові Java враховується регістр символів, тому «K» та «k» – це різні змінні.

Категорично заборонено використовувати в якості назв зарезервовані слова (boolean, true, class тощо). В одному рядку можна оголошувати декілька змінних, проте в такому разі ускладнюється читання тексту програми:

```
int k, i;
```

Ініціалізація змінної – це присвоєння їй певного значення. Існує декілька способів ініціалізації. Перший це ініціалізація змінної після її оголошення. В такому випадку оголошується змінна (тип даних, назва), а присвоєння значення реалізується в іншому місці програми (зазвичай після оголошення). Як правило, ініціалізація за такою моделлю проводиться якомога ближче до місця використання змінної в програмі. Приклад:

```
int volume;
double degree;
char oneChar;
boolean kukuruku;
```

```
///
volume = 15;
degree = 36.6;
oneChar = 'a';
kukuruku = true;
```

Інший спосіб (найбільш вживаний) – це ініціалізація під час оголошення. Приклад:

```
int volume = 15;
double degree = 36.6;
char oneChar = 'a';
boolean kukuruku = true;
```

Змінну можна ініціалізувати не тільки фіксованим значенням, а, до прикладу, арифметичним виразом, відношенням тощо. Для цього потрібно розглянути оператори, проте, забігаючи вперед, розглянемо такий приклад:

```
int k, i;
i = 5;
k = 5;
int j = k+i;
System.out.println(j);
```

Досить легко здогадатись, що результатом роботи програми буде j = 10.

Вибір назви змінної залежить від розробника, проте є кілька рекомендацій щодо імен. Зокрема, бажано давати змістовні імена. Так, наприклад, якщо бажаєте позначити кількість студентів, не слід застосовувати арифметичні позначення *n* або *s*, краще використати *numStudents*. Змістовні імена полегшують роботу з великим текстом програми як розробнику, так і решті зацікавлених людей.

Далі розглянемо ***приведення типів*** – в програмуванні це зміна типу сутності одного типу даних на інший, що може відбуватися *явно* або *неявно*. Навіщо це? До прикладу, в процесі розробки програми виникає необхідність визначити дробову частину з певної арифметичної операції. Проте ділення відбувається цілого числа на ціле. В такому випадку, якщо не використати приведення типів, результатом роботи програми буде виключно цілочисельне значення.

Для кращого розуміння розглянемо декілька реальних прикладів. Розпочнемо з найбільш популярного. В процесі розробки виникає необхідність збільшити можливий діапазон запису значень і привести тип даних «short» в «int»:

```
int a = 3;
short b = 5;
a = b; // неявне приведення
```

```
int a = 3;
short b = 5;
a = (int) b; // явне приведення
```

Або навпаки: тип даних «int» в «short»:

```
int a = 3;
short b = 5;
b = a; // помилка компіляції (неявне приведення)
```

```
int a = 3;
short b = 5;
b = (short) a; // явне приведення
```

Явне приведення типів можна процитувати: «Я знаю, що це не той тип даних, але я впевнений у працездатності коду». Неявне приведення компілятор відслідковує та вказує на будь-який нелогічний крок (приведення «int» в «short» тощо):

```
int a = 1611;
double b = 71.92;

a = b; // помилка компіляції (неявне приведення)

System.out.println((double) a); // результат
компіляції 1611.0 (явне приведення)

System.out.println((int)    b);    //    результат
компіляції 71 (явне приведення)
```

Крім примітивних типів даних, існують ще так звані об’єктні – класи, інтерфейси, масиви тощо.

Стрічки в програмуванні вважаються об’єктними типами даних, адже є об’єктами класу String. Важливо, що стрічкові літерали записують в подвійних лапках:

```
String oneString = "Перша стрічка";
```

Поняття типів даних, а найголовніше – змінних, в програмуванні потрібні для того, щоб реалізовувати в подальшому операції над ними. Отже, перейдемо до розгляду основних операторів в програмуванні.

5.2. Оператори

Оператор – це спеціальний символ, який повідомляє транслятору про необхідність виконання операції з визначеними змінними (операндами). В певному сенсі, оператори є спеціальними функціями. Окрім арифметичних дій, оператори в мовах програмування можуть виконувати логічні операції, операції з рядками, операції порівняння двох значень тощо. Оператори є базовими діями мови програмування та позначаються спеціальними символами.

Деякі оператори вимагають одного операнда, їх називають *унарними*. Якщо оператор застосований для двох операнд (додавання, ділення тощо) – *інфіксний* оператор. Якщо знак оператора ставиться перед операндами, він називається *префіксним*, якщо після – *постфіксним*.

Більшість операторів можна поділити на 4 групи: *арифметичні, розрядні (бітові), оператори відношення, логічні (булеві)*. В окремих джерелах можуть виділяти ще оператори присвоєння, хоча представники цього типу зустрічаються у всіх згаданих групах.

Арифметичні оператори використовують в математичних виразах (як і в арифметиці, алгебрі тощо).

Таблиця 5.4

Арифметичні оператори

Оператор	Операція	Оператор	Операція
+	Додавання	+=	Додаваня з присвоєнням
-	Віднімання (або унарний мінус)	-=	Віднімання з присвоєнням
*	Множення	*=	Множення з присвоєнням
/	Ділення	/=	Ділення з присвоєнням
%	Залишок ділення по модулю	%=	Залишок ділення по модулю з присвоєнням
++	Інкремент (збільшення на 1)	--	Декремент (зменшення на 1)

Розглянемо деякі приклади. Для початку візьмемо ліву частину таблиці:

```
int a = 10;
int b = 5; // оголошуємо та ініціалізуємо дві змінні

int c = a + b; // додавання
int d = a - c; // віднімання
int e = a * b; // множення
int f = e / a; // ділення
int g = a++; // постфіксний інкремент
int h = ++a; // префіксний інкремент
int k = g++;
int j = g % b; // залишок від ділення
int l = h % b;

System.out.println(c); // результат: 15
System.out.println(d); // результат: -5
System.out.println(e); // результат: 50
System.out.println(f); // результат: 5
System.out.println(g); // результат: 11
System.out.println(h); // результат: 12
System.out.println(k); // результат: 10
System.out.println(j); // результат: 1
System.out.println(l); // результат: 2
```

В результаті розгляду наведених прикладів можуть виникати питання щодо *декрементів* та *інкрементів*. Як зазначено в прикладі, оператори можуть бути як префіксними так і постфікс ними, тому *потрібно бути уважним* при їх застосуванні. Так, наприклад:

```
int x = 50;
y = x++;
```

В результаті *y = 50, а не 51*. Чому? Тому що спочатку відбувається операція присвоєння *y* значення *x*, а згодом *x* буде збільшено на 1 (в наступній за чергою операції, хоча ми цього наочно не побачимо). Щоб значення змінної було еквівалентне інструкції:

```
x = x + 1;
y = x;
```

необхідно записати вираз так:

```
y = ++x;
```

Аналогічно і з декрементом. Далі перейдемо до правої частини таблиці, тут цікавіше:

```
int a = 10;
int b = 5; // оголошуємо та ініціалізуємо ті самі змінні

int c = a += b; // додавання з присвоєнням
int d = a -= b; // віднімання з присвоєнням
int f = a *= b; // множення з присвоєнням
int g = a /= b; // ділення з присвоєнням
int h = a %= 3; // залишок ділення з присвоєнням
int k = --a; // префіксний декремент

System.out.println(c); // результат: 15
System.out.println(d); // результат: 10
System.out.println(f); // результат: 50
System.out.println(g); // результат: 10
System.out.println(h); // результат: 1
System.out.println(k); // результат: 0
```

Операнди арифметичних операторів повинні бути чисельних типів. Неможливо застосовувати арифметичні оператори до операндів булевого типу. Варто зазначити, що їх можна застосовувати з типом char, оскільки тип char є підмножиною типу int. Приклад:

```
char one = 'a';
char two = 'b';
char three = 'c'; // оголошуємо та ініціалізуємо змінні типу char

int rezult = one + two + three; // зверніть увагу на тип даних

System.out.println(rezault); // результат: 294
```

Оператори відношення (також називають операторами порівняння) визначають відношення одного операнду до іншого. Зокрема, вони визначають рівність та впорядкування операндів. Дані оператори наведені в таблиці.

Таблиця 5.5

Оператори відношення

Оператор	Опис
==	Рівно
!=	Не рівно
>	Більше
<	Менше
>=	Більше рівне
<=	Менше рівне

Результатом операції порівняння є змінна типу boolean із значеннями true або false. Найчастіше оператори відношення використовують у виразах, що застосовуються в умовних інструкціях (if - else) та у циклах.

Оператори рівно (==) та не рівно (!=) можна застосовувати практично зі всіма типами. Зверніть увагу, що оператор порівняння складається з двох знаків «дорівнює»: лише в такому застосунку здійснюється порівняння операндів. Використавши один знак «дорівнює», отримаємо оператор присвоєння. Оператори впорядкування (більше, менше) можна застосовувати лише до числових типів.

Булеві логічні оператори застосовуються для операндів типу boolean, результатом є булева величина.

Таблиця 5.6

Логічні оператори

<table>
<tr><th>Оператор</th><th>Опис</th></tr>
<tr><td>&</td><td>Логічне І (AND)</td></tr>
<tr><td>|</td><td>Логічне АБО (OR)</td></tr>
<tr><td>^</td><td>Логічне виняткове АБО (XOR)</td></tr>
<tr><td>||</td><td>Коротке АБО</td></tr>
<tr><td>&&</td><td>Коротке І</td></tr>
<tr><td>!</td><td>Логічне одиничне НІ (NOT)</td></tr>
<tr><td>&=</td><td>І з присвоєнням</td></tr>
<tr><td>|=</td><td>АБО з присвоєнням</td></tr>
<tr><td>^=</td><td>Виняткове АБО з присвоєнням</td></tr>
<tr><td>==</td><td>Рівність</td></tr>
<tr><td>!=</td><td>Не рівність</td></tr>
<tr><td>?:</td><td>Тернарний if-then-else</td></tr>
</table>

Оператори булевої логіки «&», «|», «^» оперують над булевими значеннями подібно до того, як аналогічні порозрядні оператори над бітами.

Таблиця 5.7

Таблиця істинності операторів булевої логіки

A	**B**	**A | B**	**A & B**	**A ^ B**	**!A**
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Розглянемо декілька прикладів застосування:

```
boolean a = true;
boolean b = false; / оголошуємо та ініціалізуємо змінні

boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;

System.out.println(a); // результат: true
System.out.println(b); // результат: false
System.out.println(c); // результат: true
System.out.println(d); // результат: false
System.out.println(e); // результат: true
System.out.println(f); // результат: true
System.out.println(g); // результат: false
```

Оператор присвоєння – доволі вживаний оператор при ініціалізації змінних та присвоєнні результату обчислення:

```
int x = 5;
int k = x + 8;
```

Крім того, в Java він може застосовуватися для ланцюгового присвоєння змінних:

```
x = k = z = 100;
```

Ця можливість оператора присвоєння доволі корисна за необхідності присвоєння кільком змінним одного і того ж значення.

?: – це тернарний оператор, який має три операнди. Він може заміняти в певних випадках умовну інструкцію виду if-then-else та має наступний вигляд:

```
Вираз1 ? вираз2 : вираз3;
```

Розглянемо такий приклад:

```
int x = 0;
int y = x==0 ? ++x : x;
```

В представленому прикладі перевіряється значення *x*. Якщо воно рівне 0, то *x* збільшується на одиницю і результат присвоюється *y*, якщо ні, то значення змінної *x* без змін присвоюється змінній *y*.

Висновок: знання типів даних, способів оголошення та ініціалізації змінних, а також знання основних операторів є невід'ємним базисом для подальшого оволодіння основами програмування. Розуміння базових елементів мови програмування необхідне для здобуття первинних навичок написання перших арифметичних застосунків.

Запитання до лекції

1. Що таке типізація? Перелічіть основні примітивні типи даних у Java.
2. Що таке ініціалізація змінної? Які способи ініціалізації змінної ви знаєте?
3. Наведіть приклади приведення типів.
4. Що таке оператор? На які групи поділяються оператори?

ЛЕКЦІЯ 6.
ОПЕРАТОРИ ВИБОРУ

6.1. Умовний оператор if
6.2. Оператор розгалуження (множинного вибору) switch

6.1. Умовний оператор `if`

В мовах програмування оператори керування застосовують для реалізації переходів і розгалужень в потоці виконання команд програми (залежно від її сценарію). Такі оператори в програмі мовою Java поділяються на:

- ✓ оператори вибору (розгалуження);
- ✓ оператори циклу
- ✓ оператори переходу.

Оператори вибору дозволяють реалізовувати ***розгалуження*** в виконанні команд відповідно до заданих умов. Оператори циклу дозволяють повторювати виконання програм, а оператори переходу реалізують можливість її нелінійного виконання.

В програмуванні мовою Java використовують два ***оператора вибору***: `if` і `switch`. Оператори вибору дозволяють керувати порядком виконання програми у відповідності до умов, які стають відомими лише в процесі виконання самої програми.

Спочатку розглянемо умовний ***оператор*** `if`. Це оператор умовного розгалуження програми. Його використовують з метою вибору варіантів виконання програми? залежно від заданих умов. Приклад:

```
if (умова) оператор1;
else оператор2;
```

де `оператор1` `(оператор2)` задає логіку виконання програми; `умова` – будь-який вираз, що повертає логічне значення типу `boolean`. Оператор `else` вказувати необов'язково.

Логіку оператора `if` можна описати наступним чином. Якщо умова істинна, то програма виконує оператор 1, якщо хибна – оператор 2. В жодному випадку *не буде виконано два оператори*.

Використовуючи алгоритмічне представлення, описаний вираз можна зобразити:

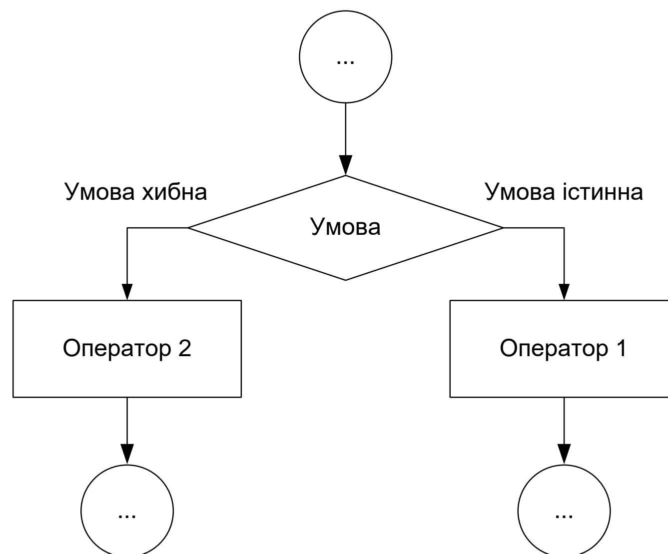

Рисунок 6.1. – Алгоритмічне представлення умовного оператора `if`

Для прикладу розглянемо подібний фрагмент програмного коду:

```
int a = 3;
int b = 2;

if (a>b) a = 0;
else b = 0;
System.out.println(a); // результат компіляції 0
```

Якщо в цьому прикладі значення змінної *a* більше, ніж значення змінної *b*, то нульове значення присвоюється змінній *a*, в іншому випадку – змінній *b*. Але в жодному разі нульове значення не може бути присвоєно одразу двом змінним.

Слід пам’ятати, що після ключових слів `if` або `else` можна використовувати тільки один оператор. Якщо в процесі виконання програми є необхідність застосування декількох операторів, їх потрібно об’єднувати в блоки операторів (фігурні дужки). Приклад:

```
int a = 3;
int b = 2;
int c = 1;
if (a>b){
    a = b+2;
    c = a+1;
}else{
    a = 0;
    c = a;
```

```
}
System.out.println(a); // результат компіляції 4
System.out.println(c); // результат компіляції 5
```

В іншому ж випадку, якщо умова не буде виконана:

```
int a = 3;
int b = 2;
int c = 1;

if (a<b){
    a = b+2;
    c = a+1;
}else{
    a = 0;
    c = a;
}
System.out.println(a); // результат компіляції 0
System.out.println(c); // результат компіляції 0
```

Вкладений умовний оператор `if` – це підоператор, який реалізує розгалуження в межах іншого умовного оператора `if` або `else`. Використовуючи вкладені оператори, слід пам'ятати, що кожен наступний оператор `else` завжди пов'язаний з найближчим оператором `if` того ж блоку (рівня). Приклад:

```
if (i == 10) {
    if (j < 20) a = b;
    if (k > 100) c = d; // вкладений умовний
оператор if
    else a = c; // оператор else, пов'язаний з if
(k > 100)
}
else a = d; // оператор else, пов'язаний з
зовнішнім оператором if (i == 10)
```

Як видно з представленого прикладу, зовнішній оператор `else` не пов'язаний з оператором `if (j < 20)`, оскільки не знаходиться з ним в межах одного блоку коду, навіть незважаючи на те, що є ближчим до нього, ніж до `if (i == 10)`. Відповідно, зовнішній оператор `else` пов'язаний з оператором `if (i == 10)`. А внутрішній оператор `else` – з оператором `if (k > 100)`.

Конструкція `if-else-if` дуже розповсюджена в програмуванні та дозволяє реалізувати декілька сценаріїв залежно від умов виконання програми.

```
if (умова)
    оператор1;
else if (умова)
    оператор2;
else if (умова)
    оператор3;
else операторN;
```

Умовний оператор `if` виконується послідовно зверху до низу. У випадку виконання певної умови (умова рівна true), яка керує оператором `if`, виконується код, що пов'язаний з даним умовним оператором `if`, а решта конструкції коду не виконується.

У випадку невиконання жодної із заданих умов, реалізується заключний оператор else за замовчуванням (якщо перевірка всіх умов дає результат false, виконується останній оператор else). Якщо оператор else не вказано, а результат перевірки заданих умов дорівнює false, то жодного оператора не буде виконано.

Далі розглянемо приклад програми, де конструкція `if-else-if` слугує для визначення пори року, до якої віднесено заданий місяць:

```
public class Month {
    public static void main(String[] args) {

        int month = 11; // Листопад
        String season;
        if (month == 12 || month == 1 || month == 2) {
            season = " зими ";
        } else if (month == 3 || month == 4 || month
== 5) {
            season = " весни ";
        } else if (month == 6 || month == 7 || month
== 8) {
            season = " літа ";
        } else if (month == 9 || month == 10 || month
== 11) {
            season = " осені ";
        } else {
```

```
            season = " не існуючий місяць ";
        }
System.out.println("Заданий місяць відноситься до " +
season + "."); // результат: Заданий місяць
відноситься до осені
    }
}
```

З представленого прикладу можна переконатись, що за умови правильної архітектури `if-else-if` виконання заданої умови можливе лише один раз.

6.2. Оператор розгалуження `switch`

В програмуванні мовою Java ***оператор*** `switch` є оператором розгалуження (з англійської – перемикач). В деяких джерелах можна зустріти іншу назву – «оператор множинного вибору». Цей оператор надає можливість обирати напрямок виконання програми залежно від значення керуючого виразу. У багатьох випадках оператор `switch` є ефективнішим від громіздкої архітектури коду `if-else-if`. Оператор `switch` працює наступним чином. Значення виразу, що записане у `switch`, почергово порівнюється із значеннями операторів `case`. За умови рівності значення `switch` та значення відповідного оператору `case` далі виконується послідовність коду, зв'язаного тільки з відповідним оператором `case`. Якщо рівності не виявлено – виконується оператор `default`.

Загальна форма оператора `switch` має наступний вигляд:

```
switch (вираз) {
    case значення1:
        // послідовність операторів
        break;
    case значення2:
        // послідовність операторів
        break;
    case значенняN:
        // послідовність операторів
        break;
    default:
```

```
            // послідовність операторів за
замовчуванням
    }
```

Вираз switch може бути будь-якого типу даних: byte, short, int, char, String (у версіях Java SE7 та вище) тощо. Значення операторів розгалуження case має бути константним виразом та сумісним за типом даних із зазначенням виразу switch. Дублювання значень в операторах розгалуження case недопустиме.

Оператор break слугує для переривання послідовності виконання операторів розгалуження case. Цей оператор відповідає за продовження виконання програмного коду за межами оператора розгалуження switch (з першого рядка коду, який слідує після всього оператора switch). Інакше кажучи, оператор break слугує для негайного виходу з оператора switch. Приклад:

```
public class MainSwitch {

    public static void main(String[] args) {

        switch (1) {
        case 0:
            System.out.println(" нуль ");
            break;
        case 1:
            System.out.println(" одиниця ");
            break;
        case 2:
            System.out.println(" двійка ");
            break;
        case 3:
            System.out.println(" трійка");
            break;
        default:
            System.out.println(" чітвірка ");
        }
    }
}
```

Результат виконання програми: одиниця.

Або ж приклад з типом даних String:

```
public class MainSwitchTwoo {

    public static void main(String[] args) {

        switch ("Віталій") {
        case "Віталій":
            System.out.println(" так ");
            break;
        case "Олег":
            System.out.println(" ні ");
            break;
        default:
            System.out.println(" не має співпадінь ");
        }
    }
}
```

Результат виконання програми: так.

За відсутності операторів break буде виконано програмний код усіх операторів розгалуження case та default. Приклад:

```
public class MainSwitchThree {

    public static void main(String[] args) {

        switch ("Віталій") {
        case "Віталій":
            System.out.println(" так ");
        case "Олег":
            System.out.println(" ні ");
        default:
            System.out.println(" немає співпадінь ");
        }
    }
}
```

Результат виконання програми: так
ні
немає співпадінь

Інколи під час створення програм виникає необхідність в певній частині коду не застосовувати оператор `break.` Для огляду подібного варіанту програмного коду, а також порівняння переваг і недоліків умовного оператора `if` та `switch`, розглянемо приклад програми із визначення пори року, до якої відносять заданий числом місяць (з оператором if розглянуто в попередньому питанні):

```
public class MainMonth {
    public static void main (String[] args) {
        int month = 4;
        String season;
        switch (month) {
        case 12:
        case 1:
        case 2:
            season = "зими";
            break;
        case 3:
        case 4:
        case 5:
            season = "весни";
            break;
        case 6:
        case 7:
        case 8:
            season = "літа";
            break;
        case 9:
        case 10:
        case 11:
            season = "осені";
            break;
        default:
            season = "не існуючий місяць";
        }
System.out.println("Заданий місяць відноситься до " +
season + "."); // результат: Заданий місяць
відноситься до весни

    }
}
```

Як бачимо, в певній частині коду не відбувалось застосування оператора `break` після оператора розгалуження `case`. Такий застосунок необхідний для скорочення коду та можливості застосування одного результату (пори року) відразу для трьох `case`ів.

Переваги застосування оператора `switch` над `if` очевидні – це простота написання коду. А найголовніше, відсутня необхідність створення складних блоків операторів із безліччю дужок, в яких просто припуститися помилки.

Вкладений оператор `switch` можна використовувати в послідовності зовнішніх операторів `switch` (аналогічно оператору if). Кожен оператор `switch` визначає свій блок програмного коду, відтак жодних конфліктів між зовнішніми та вкладеними `case`ами не відбувається. Приклад:

```
int count;
int target;
// ...
switch (count) {
    case 1:
        switch (target) { // вкладений оператор
switch
            case 0:
                System.out.println(" target равно 0");
                break;
            case 1: // жодних конфліктів з
зовнішнім оператором case 1
                System.out.println("target равно 1");
                break;
    case 2: // продовження програми

        }
}
//...
```

В наведеному прикладі оператор розгалуження `case 1` вкладеного оператора `switch` не конфліктує з оператором розгалуження `case 1` зовнішнього оператора `switch`. Значення змінної `count` порівнюється лише зі значеннями зовнішніх операторів розгалуження `case`. Якщо значення змінної `count` рівне 1, то значення змінної `target` порівнюється зі значеннями внутрішніх операторів розгалуження `case`.

Підводячи підсумок, виділимо деякі важливі особливості оператора `switch`:

✓оператор switch відрізняється від оператора if тим, що в ньому допустимо виконувати лише перевірку рівності. В операторі if передбачено можливість визначення результату логічного виразу будь-якого типу;

✓змінні в декількох операторах розгалуження `case` одного оператора `switch` не можуть мати однакового значення;

✓як правило, оператор `switch` є ефективнішим, ніж структура вкладених умовних операторів if.

Зважаючи на останню особливість, можна стверджувати, що в тих випадках, коли з’являється необхідність здійснювати вибір серед великої групи значень, оператор `switch` буде працювати значно швидше, аніж послідовність операторів `if-else`. Це пов’язано з тим, що змінні усіх розгалужень `case` мають один і той же тип, тому їх достатньо перевірити лише на рівність змінній оператора `switch`.

Висновок: в елементах процедурного програмування оператори розгалуження відіграють одну з найважливіших функцій. Більшість процедурних елементів програмного коду, залежно від певних умов, потребують реалізації розгалуження. Саме тому вивчення та розуміння принципу роботи основних операторів вибору (розгалуження) є важливим для подальшого оволодіння навиками програмування.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с.
2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.
3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java
4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java.
5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java**.**

Запитання до лекції

1. Які оператори застосовуються в мові програмування Java?
2. Розкрийте принцип роботи та наведіть приклади застосування умовного оператора *if*.
3. Розкрийте принцип роботи та наведіть приклади застосування оператора розгалуження switch.

ЛЕКЦІЯ 7.
ОПЕРАТОРИ ЦИКЛІВ

7.1. Цикл з передумовою while
7.2. Цикл з післяумовою do-while
7.3. Цикл з лічильником for
7.4. Оператори переходу

7.1. Цикл з передумовою while

Для керування конструкціями, що потребують циклічного виконання програмного коду, використовують оператори циклу `for`, `while` та `do-while`. ***Оператори циклу*** – це оператори багаторазової реалізації однієї інструкції до моменту виконання умови виконання циклу.

Оператор циклу `while` повторює свій оператор або блок операторів, поки значення його керуючого виразу (умови) є істинним. Цей оператор циклу має таку форму:

```
while(умова) {
    //тіло циклу
}
```

де `умова` – це будь-який логічний вираз; `тіло циклу`– код програми.

Графічну модель роботи циклу `while` представлено на рисунку 7.1:

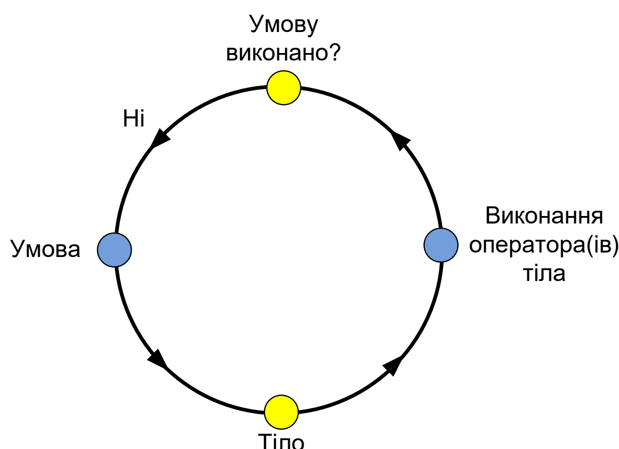

Рисунок 7.1 – Графічна модель циклу `while`

Тіло циклу викується, поки умова істинна. Коли умова стає хибою, виконання коду програми виходить за межі циклу. Фігурні

дужки можуть не використовуватись тільки у тому випадку, якщо в циклі повторюється лише один оператор.

В якості прикладу розглянемо використання циклу `while` для виконання зворотного рахунку від 10 до 1 (виконується 10 тактів):

```
publicclassWhile {

    publicstaticvoidmain(String[] args) {
        intn = 10;
        while (n> 0) {
            System.out.println("такт " + n);
            n--;
        }
    }
}
```

Результат роботи програми:

```
такт 10
такт 9
такт 8
такт 7
такт 6
такт 5
такт 4
такт 3
такт 2
такт 1
```

На початку циклу `while` відбувається перевірка умови. Якщо умова не виконується, тіло циклу не буде виконано жодного разу. Приклад:

```
int a = 10, b = 20;
    while (a > b)System.out.println ( " a більше
за b " );
// Цю стрічку не буде виведено
```

Тіло циклу `while` (як і будь-якого іншого циклу) може бути пустим. Розглянемо приклад, де умову об’єднано з тілом циклу:

```
publicclassMain {
    publicstaticvoidmain(String[] args) {
```

```
        inti, j;
        i = 100;
        j = 200;
        while ( ++i< --j ) ;
        System.out.println(i);
        System.out.println(j);
    }
}
```

В приведеному прикладі оголошено та ініціалізовано дві змінні $i = 100$ та $j = 200$. Умовою зазначено, що цикл `while` виконуватиметься до тих пір, поки змінна j більша за змінну i. Одночасно умовою передбачено інкремент змінної i (кожного разу збільшується на 1) та декремент змінної j (кожного разу зменшується на 1). Зважаючи на описану логіку, виконання циклу відбуватиметься до моменту присвоєння змінним однакового значення (в такому випадку умова `i<j` не виконуватиметься). Результатом роботи програми, незважаючи на те, що тіло циклу відсутнє, буде:

```
150
150
```

Для кращого уявлення про попередній приклад розглянемо програму виконання ідентичної умови, проте з представленням тіла циклу:

```
publicclassMain {
    publicstaticvoidmain(String[] args) {
        inti, j;
        i = 100;
        j = 200;
        while (i<j){
            ++i;
            --j;
        }
        System.out.println(i);
        System.out.println(j);
    }
}// Результат роботи ідентичний попередньому
прикладу
```

7.2. Цикл з післяумовою do-while

Мив вже знаємо, що якщо умова циклу while є хибною, то тіло не буде виконано жодного разу. Та інколи виникає необхідність виконати цикл хоча б один раз, навіть якщо умова хибна. Для цих цілей в програмуванні використовують цикл з післяумовою do-while, коли перевірка умови виконується після виконання тіла циклу. Тіло цього циклу завжди виконуватиметься щонайменше один раз. Цей оператор циклу має таку форму:

```
do {
    //тіло циклу
}while(умова);
```

Графічну модель роботи циклу do-while представлено на рисунку 7.2:

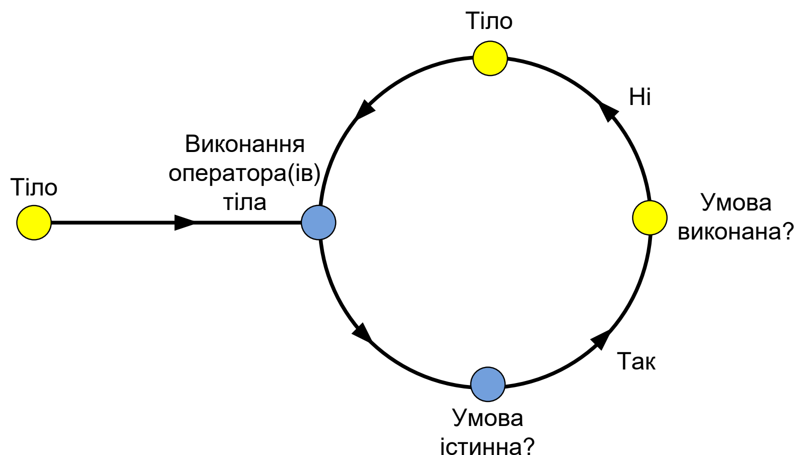

Рисунок 7.2 – Графічна модель циклу do-while

Отже, за умови виконання циклу do-while спочатку виконується тіло, а потім перевіряється істинність умови. Ось чому можливе виконання циклу хоча б один раз, навіть якщо умова хибна.

Якщо умова істинна, наступним кроком є почергова перевірка виконання умови та виконання тіла циклу. Цикл повторюватиметься до моменту виконання умови. Як і у всіх циклах, в циклі do-while ***умова має бути логічним виразом***.

Для прикладу розглянемо аналогічну програму виведення у консоль 10 тактів (перероблену під do-while):

```
publicclass DoWhile {

    publicstaticvoid main(String[] args) {
        intn = 10;
```

```
        do {
            System.out.println("такт " + n);
            n--;
        } while (n> 0);
    }
}// Результат роботи ідентичний першому прикладу в лекції
```

Беручи до уваги ідеї, висвітлені в попередньому питанні (про суміщення умови та логіки виконання), цілком правильним є приклад застосунку циклу `do-while`, представлений у вигляді:

```
intn = 10;
    do {
        System.out.println("такт " + n);
    } while (--n> 0);
```

В наведеному прикладі декремент змінної *n* та перевірка умови об'єднані в одному виразі `(--n> 0).` Спочатку виконується операція декременту, зменшуючи значення змінної *n* на одиницю, після чого це значення порівнюється з нулем. Якщо значення після виконання чергового циклу є більшим за нуль, то цикл повторюється.

Цикл `do-while` є корисним, коли виникає необхідність програмування меню, яке потрібно вивести перший раз без виконання будь-який дій.

7.3. Цикл з лічильником for

Форма традиційного оператора циклу `for` виглядає таким чином:

```
for(ініціалізація; умова; ітерація) {
    //тіло циклу
}
```

Класично, якщо в циклі повторюється виконання лише одного оператора, то застосування блоку (фігурних дужок) не потрібно.

Цикл `for` працює наступним чином. Коли цикл починає працювати, виконується його ініціалізація. Ініціалізація встановлює значення ***змінній керування циклом***, яка, в свою чергу, діє в якості лічильника. Важливо, що перша частина циклу (ініціалізований вираз) виконується лише раз. Наступним кроком є перевірка заданої

умови, яка обов'язково має бути логічним виразом. Як правило, в цій частині значення змінної керування циклом порівнюється з цільовим значенням. Якщо результат порівняння істинний, то виконується тіло циклу, якщо хибний – цикл завершується. Остання частина циклу `for` – ітерація. Зазвичай ця частина циклу містить вираз, в якому збільшується або зменшується значення змінної керування циклом. Після ітерації цикл повторюється та на кожному наступному кроці спочатку перевіряється виконання умови, після чого виконується тіло циклу та чергова ітерація. Цей процес повторюватиметься до тих пір, поки результат ітераційного виразу не стане хибним (не задовольнятиме умову).

Інакше кажучи, остання частина циклу `for` (ітерація) призначена для визначення кількості разів виконання тіла циклу до моменту виконання умови. Графічну модель роботи циклу `for` представлено на рисунку 7.3:

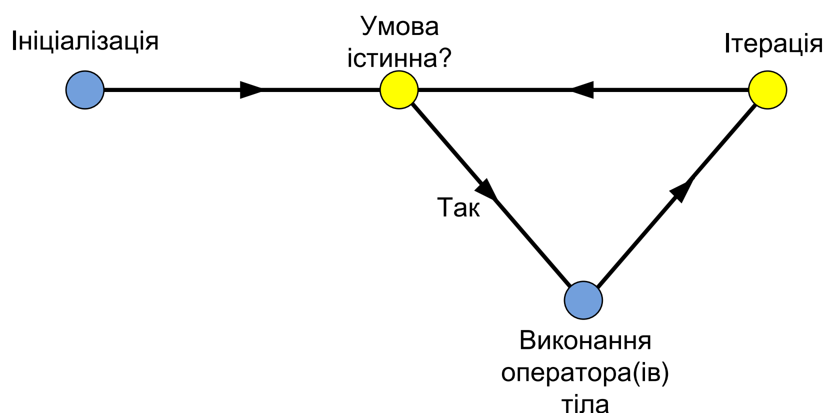

Рисунок 7.3 – Графічна модель циклу `for`

Розглянемо приклад програми виведення у консоль 10 тактів:

```
publicclassForTakt {
    publicstaticvoidmain(String[] args) {
        intn;
        for (n = 10; n> 0; n--)
            System.out.println("такт " + n);
    }
}
```

Інколи змінна керування циклом потрібна лише для самого циклу (більше ніде не застосовується). В такому разі змінну керування циклом можна оголосити та ініціалізувати безпосередньо в самому циклі (в частині ініціалізації оператора `for`). Приклад:

```
publicclassForTakt {
    publicstaticvoidmain(String[] args) {
        for ( intn=10; n>0 ; n-- )
            System.out . println ( "такт " + n );

    }
}
```

Область та термін дії змінної керування циклом, яка оголошена та ініціалізована безпосередньо в циклі співпадає з областю та терміном дії самого циклу. За межами циклу `for` така змінна припиняє своє існування. Якщо існує необхідність використання змінної в інших частинах програми, її не можна оголошувати в блоці циклу `for`.

Існує й інший випадок, коли оголошення та ініціалізація змінної проведені за межами циклу. В такому випадку повторної ініціалізації змінної в блоці оператора `for` проводити не потрібно. Приклад:

```
publicclassForTakt {

    publicstaticvoidmain(String[] args) {
    intn = 10;
        for ( ; n> 0; n--)
            System.out.println("такт " + n);
    }
}
```

Існують випадки, коли виникає необхідність вказати декілька змінних в частині ініціалізації та декілька операторів в ітераційній частині циклу. Для прикладу розглянемо наступний код:

```
publicclassMain {
    publicstaticvoidmain(String[] args) {
        inta, b;
        b = 4;
        for (a = 1; a<b ; a++){
            System.out.println ( "a = " + a);
            System.out.println ( "b = " + b);
            b--;
        }
```

```
    }
}
```

В представленому прикладі керування циклом реалізується одночасно двома змінними *a* та *b*. Оскільки змінних керування циклом є дві, то їх бажано включати безпосередньо в оператор циклу `for`. Для реалізації такої можливості в Java існує « , ».

Щоб передбачити можливість керування циклом декількома змінними, в Java допускається зазначати декілька змінних в частині ініціалізації та декілька операторів в частині ітерації, розділяючи їх *комою*. Застосовуючи таку можливість, попередній цикл можна переписати:

```
publicclassMain {

    publicstaticvoidmain(String[] args) {
        inta, b;
        for (a = 1, b = 4; a<b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
        }
    }
}
```

Результат роботи циклу в обох випадках однаковий:

```
a = 1
b = 4
a = 2
b = 3
```

Гнучкість циклу `for` полягає в тому, що три його частини (ініціалізація, перевірка умови, ітерація) не обов'язково завжди застосовувати за прямим призначенням. Розглянемо декілька прикладів.

В найбільш вживаному застосунку циклу `for` використовують умову виконання циклу у вигляді умовного виразу. Проте в окремих випадках не потрібно зрівнювати змінну керування циклом з певним числовим значенням. Умовою виконання циклу `for` може бути будь-який логічний вираз. Приклад:

```
public class Main {

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

```
            boolean done = false;
            for (inti = 1; !done; i++)
                if (interrupted())
                    done = true;
        }
    }
```

В зазначеному прикладі виконання циклу for відбувається, доки в змінній done не буде встановлено логічне значення true.

Розглянемо ще одну цікаву різновидність циклу. В одному з попередніх прикладів показано, що за умови оголошення та ініціалізації змінної перед циклом її повторно не потрібно зазначати в самому циклі. Існує аналогічний застосунок з ітераційною частиною циклу for. Ініціалізуючий та ітераційний вираз можуть не зазначатись в операторі циклу for. Приклад:

```
public class Main {

    public staticvoid main (String[] args) {
        int i;
        boolean done = false;
        i = 0;
        for ( ; !done; ) {
            System.out.println ("i рівне " + i);
            if (i == 10)
                done = true;
            i++;
        }
    }
}
```

В представленому прикладі ітераційна та ініціалізуюча часини винесені за межі циклу for (ініціалізуюча – перед циклом, ітераційна – після циклу). В результаті подібного представлення коду програми відповідні частини оператора циклу залишаються пустими (крапки з комою зазначати обов'язково!). Результатом роботи програми за наведеним прикладом буде:

```
i рівне 0
i рівне 1
i рівне 2
```

```
i рівне 3
i рівне 4
i рівне 5
i рівне 6
i рівне 7
i рівне 8
i рівне 9
i рівне 10
```

Представимо ще один різновид циклу. Якщо залишити усі частини оператора циклу пустими, ми навмисно створимо нескінченний цикл:

```
for ( ;; ) {
    // …
}
```

Починаючи з версії JDK 5, в Java з'явився новий різновид циклу `for` в стилі foreach. Цикл в цьому стилі призначений для суворо послідовного виконання повторюваних дій над колекцією об'єктів типу масиву.

Вкладені цикли. Як і в операторах розгалуження, в циклах допускається використання вкладених операторів. Це означає, що один цикл може виконуватись в межах іншого. Графічно це можна відобразити наступним чином:

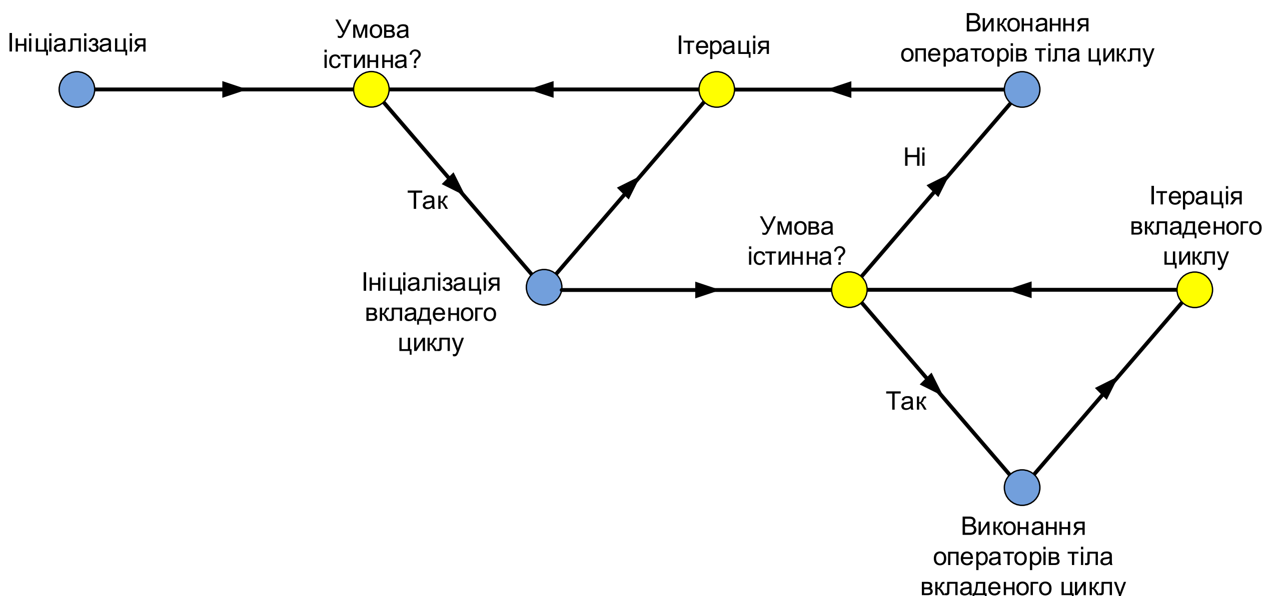

Рисунок 7.4 – Графічна модель вкладеного циклу `for`

Розглянемо приклад вкладеного циклу `for`.

```
public class Main {

    public static void main(String[] args) {
        int i, j;
        for (i = 0; i< 10; i++) {
            for (j = i; j< 10; j++)
                System.out.print ("* ");
            System.out.println();
        }
    }
}
```

Результатом роботи програми буде:

```
* * * * * * * * * *
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

7.4. Оператори переходу

В мові програмування Java визначені три оператори переходу: break, continue, return. Основна задача цих операторів – перехід між різними частинами коду програми.

В програмуванні на Java ***оператор break*** знайшов три застосування. Перше – це завершення послідовності в операторі switch (розглянуто в попередній темі). Друге – використання для виходу з циклу за певних умов. Третє – застосування в якості «цивілізованої» (за Г. Шилдтом) форми оператора безумовного переходу goto. Зважаючи, що перший випадок було детально розглянуто раніше, зараз зосередимо увагу на аналізі лише другого та третього різновидів застосувань.

Використання оператора break для виходу з циклу. Використовувати операторbreak можна виконати операцію миттєвого виходу за межі циклу пропускаючи умову його виконання та будь-який код в тілі циклу. Коли в тілі циклу зустрічається оператор break, виконання циклу припиняється, а керування передається оператору, який слідує відразу після циклу. Приклад застосування оператора break для виходу з циклу:

```
public class Main {

    public static void main (String[] args) {
        for (int i = 0; i< 100; i++) {
            if (i == 10)
                break; //вихід з циклу при
досягненні значення 10
            System.out.println ("i : " + i);
        }
        System.out.println ("Цикл завершений");
    }
}
```

Результатом роботи програми буде:

```
i : 1
i : 2
i : 3
i : 4
i : 5
i : 6
i : 7
i : 8
i : 9
Цикл завершений
```

Як видно з представленого циклу оператор break призводить до виходу з циклу до моменту його повного виконання при досягненні значення i = 10, хоча умова циклу передбачає його виконання до того моменту, поки i не дорівнюватиме 99.

Оператор break може застосовуватись те тільки з циклом for. Розглянемо приклад виконання аналогічної програми із застосуванням циклу while:

```
public class Main {

    public static void main (String[] args) {
        int i = 0;
        while (i< 100) {

            if (i == 10)
                break; //вихід з циклу при
досягненні значення 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Цикл завершений.");

    }
}
```

Результат роботи програми буде аналогічним попередньому прикладу.

Якщо в програмі застосовується вкладений цикл то застосування оператора break забезпечуватиме вихід тільки з внутрішнього по черговості циклу. Приклад:

```
public class Main {

    public static void main(String[] args) {
        for (int i = 0; i< 3; i++) {
            System.out.print ("Прохід " + i + ":
");

            for (int j = 0; j< 100; j++) {
                if (j == 10)
                    break; // вихід з циклу
                System.out.print (j + " ");
            }
            System.out.println ();
        }
        System.out.println ("Цикли завершені.");
    }
```

```
}
```

Результат роботи програми:

```
Прохід 0: 0 1 2 3 4 5 6 7 8 9
Прохід 1: 0 1 2 3 4 5 6 7 8 9
Прохід 2: 0 1 2 3 4 5 6 7 8 9
Цикли завершені.
```

Застосування оператора break у внутрішньому циклі може призводити тільки до виходу тільки з цього циклу. На зовнішній цикл цей оператор не матиме жодного впливу.

Важливо! Оператор break не слід використовувати в якості традиційного способу виходу з циклу. Для цього в програмуванні використовують умову виконання циклу (умовний вираз). Оператор breakслід використовувати для виходу з циклу тільки в окремих випадках.

Використання оператора breakв якості форми оператора goto. Як вже зрозуміло оператор break можна застосовувати не лише в якості оператора для виходу з switch та циклів, а ще й в якості оператора безумовного переходу goto. В мові програмування Java оператор goto відсутній, оскільки він дозволяє виконувати розгалуження програм випадковим та неструктурованим способом. Як правило, код який керується оператором goto важкий для розуміння та супроводу. Крім того цей оператор виключає можливість оптимізації коду для визначеного компілятора. Проте в деяких випадках оператор goto виявляється цілком зручним та допустимим засобом для керуванням потоком виконання програм. До прикладу, оператор goto може виявитись корисним при виході з ряду глибоко вкладених циклів. Для подібних випадків в Java визначена розширена форма оператора break. Використовуючи цю форму, можна, наприклад, організувати вихід з одного або декількох блоків коду. Крім того можна чітко вказати оператор, з якого буде продовжено виконання програми, оскільки ця форма оператора break має відповідні мітки. Оператор break з міткою надає усі переваги відомого оператора goto. Загальна форма оператора break з міткою має такий вигляд:

```
Break мітка;
```

В більшості випадків мітка – це ім’я мітки, яке зазначає блок коду. Ним може бути як самостійний блок коду так і цільовий блок

іншого оператора. За умови застосування цього оператора керування роботою програми передається до блоку кодів помічених відповідною міткою. Слід розуміти, що оператор break з міткою можна використовувати для виходу з низки вкладених блоків, проте його не використовують як «посилання» для передачі керування програмою визначеному блоку операторів.

Для того щоб помітити блок операторів достатнього позначити на його початку мітку. Мітка – це будь-який доступний в Java ідентифікатор з двокрапкою. В наведеному прикладі програма містить три вкладених блоки, кожен з яких помічений міткою. Оператор break виконує перехід в кінець блоку з міткою second, пропускаючи два виклики методу println ():

```
public class Main {

    public static void main(String[] args) {

        boolean t = true;
        first: {
            second: {
                third: {
                    System.out.println("Знаходиться
                                perед оператором
                                break.");
                    if (t) break second; // вихід з
блоку second
                    System.out.println("Цей оператор
                                не
                                виконуватиметься")
                                ;
                    System.out.println("Цей оператор
                                не
                                виконуватиметься")
                                ;
                }
            }
        System.out.println("Цей опертор слідує за
блоком second.");
        }
    }
```

```
}
```

Результат виконання програми:

```
Знаходиться перед оператором break.
Цей опертор слідує за блоком second.
```

Одним з найбільш розповсюджених способів використання оператора break з міткою є вихід із вкладених циклів. В наведеному прикладі зовнішній цикл виконуватиметься лише один раз:

```
public class Main {
    public static void main(String[] args) {

        outer: for (int i = 0; i < 3; i++) {
            System.out.print("Прохід " + i + ":
");
            for (int j = 0; j < 100; j++) {
                if (j == 10)
                    break outer; // вихід з обох
циклів
                System.out.print (j + " ");
            }
            System.out.println("Цей рядок не
виводитиметься");
        }
    System.out.println("Цикли завершені");
    }
}
```

Результат виконання програми:

```
Прохід 0: 0 1 2 3 4 5 6 7 8 9 Цикли завершені
```

Як видно, за умови виконання виходу з вкладеного у зовнішній цикл, це призводить до завершення роботи обох циклів.

Важливо! Не можна виконувати перехід до мітки, якщо вона визначена не для блоку зовнішніх операторів, а для окремо розташованої частини загального коду програми (не дотримуючись ієрархії). Для кращого розуміння розглянемо приклад програмного коду, результатом роботи якого буде помилка:

```java
public class Main {

    public static void main(String[] args) {

        one: for (int i = 0; i < 3; i++) {
            System.out.print("Прохід" + i + ":");
        }

        for (int j = 0; j < 100; j++) {
            if (j == 10) break one; // ПОМИЛКА !
            System.out.print(j + " ");
        }
    }
}
```

Блок коду помічений міткою one являється окремим блоком, який не містить в зв'язку вкладених циклів оператора break, тому і перехід до цього блоку не можливий.

Оператор continue. В низці літературних джерел зустрічається безліч тлумачень цього оператора, ми ж визначимо його призначення значно простіше. Оператор ***continue*** використовують для переходу в тілі циклу до його завершення. В циклах while та do-while оператор continue викликає передачу керування безпосередньо умовному виразу, який керує циклом. В циклі for керування передається спочатку ітераційній частині, а потім умовному виразу. У всіх трьох видах циклів любий проміжний код пропускається. Приклад:

```java
public class Main {

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            System.out.print(i + " ");
            if (i % 2 == 0) continue;
            System.out.println();
        }
    }
}
```

Результатом роботи програми буде вивід в кожному рядку по два числа послідовністю від 0 до 9. Власне з метою виводу тільки двох чисел в кожному рядку використовується оператор continue:

```
0 1
2 3
4 5
6 7
8 9
```

В наведеному прикладі оператор % використовується для перевірки парності значення змінної i. Якщо змінна приймає парне значення на черговій ітерації циклу, то виконання цього ж циклу продовжується без переходу на новий рядок (спрацьовує оператор continue та метод println () не виконується).

Як і оператор break, оператор continue може містити мітку. Розглянемо приклад такого застосунку у вигляді програми виводу трикутної таблиці множення чисел від 0 до 9:

```
public class Main {

    public static void main(String[] args) {

        outer: for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                if (j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
        }
        System.out.println();
    }
}
```

Результат роботи програми:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
```

```
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

В наведеному прикладі оператор continue перериває цикл підрахунку значень змінної j та продовжує його з наступного кроку циклу, в якому вираховуються значення змінної i.

Отже, підведемо підсумок. В тих випадках, коли при написанні програми із використанням циклів виникає необхідність реалізації більш раннього початку нового кроку циклу, оператор continue є вдалим інструментом.

Оператор return. Цей оператор слугує для виконання явного виходу з методу, тобто передає керування об'єкту, який викликав відповідний метод. Інакше кажучи, оператор return негайно зупиняє роботу методу, в тілі якого він знаходиться. Приклад:

```
public class Main {

   public static void main(String[] args) {
        boolean t = true ;
        System.out.println ("До повернення");
        if (t) return ; // повернення до коду 
об’єкту, що викликає метод
        System.out.println ("Цей оператор не 
виконуватиметься");
   }
}
```

В наведеному прикладі застосування оператора return призводить до повернення управління керуючій системі Java, оскільки саме вона викликає метод main ( ). Результатом роботи програми буде:

```
До повернення
```

Отже, заключний метод println ( ) не виконуватиметься. Відразу після виконання оператора return управління повертається об'єкту, що його викликає.

Висновок: в елементах як процедурного, так об'єктно-орієнтованого програмування оператори циклів та переходу відіграють важливу роль. Автоматизація циклічності роботи програми в більшості випадків значно спрощує процес проходження масивів та колекцій, а також значно зменшує обсяги програмного

коду. Оператори переходу є незамінними в процесі управління ходом та логікою виконання програмного коду. Саме тому вивчення та розуміння принципу роботи операторів циклу та переходу є важливим для подальшого оволодіння навиками програмування.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И. Д. Вильяме", 2015. – 1376 с.

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

Запитання до лекції

1. Розкрийте принцип роботи та наведіть приклади застосування оператора циклу *while.*
2. Розкрийте принцип роботи та наведіть приклади застосування цикла з післяумовою *do-while*.
3. Розкрийте принцип роботи та наведіть приклади застосування оператора циклу з лічильником *for*.
4. Які оператори переходу застосовуються в середовищі програмування Java?
5. Розкрийте принципи роботи та наведіть приклади застосування операторів переходу.

ЛЕКЦІЯ 8.
МАСИВИ

8.1. Одновимірні масиви
8.2. Багатовимірні масиви

8.1. Одновимірні масиви

Масиви – це група однотипних змінних, для звернення до яких використовують спільне ім'я. В Java можливе створення масивів будь-якого типу та різної величини. Доступ до елементів масиву реалізується через їх індекси. Масиви є зручним способом групування однотипної інформації.

Одновимірні масиви мають вигляд списку однотипних змінних. Щоб створити масив, необхідно оголосити змінну масиву потрібного типу та вказати ім'я масиву. Загальна форма оголошення одновимірного масиву виглядає так:

```
тип ім'я_змінної [];
```

де параметр `тип` зазначає тип усіх елементів масиву (його ще називають базовим типом). Тип масиву визначає тип даних, які буде містити масив. До прикладу, масив `month_day` буде містити елементи типу `int:`

```
int month_days [];
```

Попри те, що в наведеній стрічці коду наведено інформацію, що `month_days` оголошено як масив, жодного масиву наразі не існує. Аби зв'язати оголошений масив `month_day` з масивом цілочисельних значень типу `int`, потрібно зарезервувати область пам'яті за допомогою оператора `new` та призначити її адресу масиву. Ця операція потрібна для виділення пам'яті під масив. Загальна форма оператора `new` для одновимірних масивів виглядає так:

```
змінна_масиву = new тип [розмір];
```

де `тип` – тип даних, для яких резервується пам'ять; `розмір` – кількість елементів в масиві; `змінна_масиву` – означає змінну, зв'язану безпосередньо з масивом.

Іншими словами, щоб скористатись оператором `new` для резервування пам'яті, необхідно зазначити тип та кількість елементів,

для яких необхідно зарезервувати пам'ять. Елементи масиву, для яких було виділено пам'ять, будуть автоматично ініціалізовані або нульовим значенням (для числових типів), або значенням false (для логічного типу), або пустим значенням null (для посилкових типів даних).

В наведеному далі прикладі резервується пам'ять для 12 елементів масиву цілих значень, які зв'язані з масивом month_days. Після виконання наведеної стрічки змінна масиву month_days буде містити посилання на масив, який складається з 12 цілочисельних елементів. При цьому всі значення цього масиву будуть ініціалізовані нульовими значеннями:

```
month_days = new int [12];
```

Підведемо деякі підсумки: процес створення масиву проходить в два етапи. Спочатку необхідно оголосити змінну необхідного типу масиву. А потім за допомогою оператора new зарезервувати пам'ять для зберігання масиву і присвоїти його адресу раніше оголошеній змінній. Означені операції можна об'єднати, що значно спрощує процедуру оголошення та ініціалізації масиву даних (найбільш розповсюджений за стосунок):

```
int month_days [] = new int [12];
```

Як тільки буде створено масив та зарезервовано пам'ять під нього, з'явиться можливість звертатись до конкретного елемента масиву, зазначаючи його індекс в квадратних дужках. СЛІД ПАМ'ЯТАТИ, що індекси масиву починають відлік з нуля.

Для демонстрації процедури створення масиву та його наповнення шляхом звернення до елементів за індексами розглянемо наступний приклад. В зазначеному прикладі створюється масив, який міститиме інформацію про кількість днів у кожному місяці:

```
public class Array {

    public static void main(String[] args) {
        int month_days [] = new int [12];
        month_days [0] =31;
        month_days [1] =28;
        month_days [2] =31;
        month_days [3] =30;
        month_days [4] =31;
        month_days [5] =30;
        month_days [6] =31;
```

```
        month_days [7] =31;
        month_days [8] =30;
        month_days [9] =31;
        month_days [10] =30;
        month_days [11] =31;
    }
}
```

Для звернення до визначеного елементу масиву необхідно зазначити ім'я змінної масиву та індекс елементу, до якого реалізується запит:

```
public class Array {

    public static void main(String[] args) {
        int month_days[] = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
        month_days[10] = 30;
        month_days[11] = 31;

        System.out.println("В січні " +
month_days[0] + "день(днів)");
        System.out.println("В лютому " +
month_days[1] + "день(днів)");
        System.out.println("В березні " +
month_days[2] + "день(днів)");
        System.out.println("В квітні " +
month_days[4] + "день(днів)");
        // . . .

    }
}

Результат роботи програми:
В січні 31 день(днів)
```

```
В лютому 28 день(днів)
В березні 31 день(днів)
В квітні 31 день(днів)
```

В наведеному прикладі оголошення масиву (резервування пам'яті) та його ініціалізація проведена двома окремими етапами. При оголошенні масиву ми створили змінну, зарезервували для масиву визначений діапазон пам'яті та присвоїли її адресу створеній змінній. Процедуру ініціалізації або наповнення масиву проведено шляхом звернення до кожного елементу окремо та встановленням відповідного значення у комірку за вказаним індексом.

Проте в програмуванні мовою Java можливо оголошувати та ініціалізовувати масиви одночасно. Ця процедура аналогічна процедурі одночасного оголошення та ініціалізації звичайних змінних. Одночасна ініціалізація масиву при його оголошенні відбувається шляхом подання списку виразів, що розділені комами і об'єднані фігурними дужками. Коми розділяють значення елементів масиву. Об'єм масиву автоматично створюється такої величини, щоб вмістити усі перераховані елементи (елементи зазначаються в ініціалізаторі масиву). За умови такої ініціалізації масиву, необхідність у застосуванні оператора **new** відпадає. Наприклад, щоб зберегти кількість днів у кожному місяці, можна використати наведений код:

```
 int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31 };
```

Як перший, так і другий варіант ініціалізації (наповнення) масиву видають однаковий результат та мають право на існування. Спробуємо вивести у консоль результати запитів, як у попередньому прикладі:

```
public class AutoArray {

    public static void main(String[] args) {
        int month_days[] = { 31, 28, 31, 30, 31,
30, 31, 31, 30, 31, 30, 31 };

        System.out.println("В січні " +
month_days[0] + " день(днів)");
```

```
        System.out.println("В лютому " +
month_days[1] + " день(днів)");
        System.out.println("В березні " +
month_days[2] + " день(днів)");
        System.out.println("В квітні " +
month_days[4] + " день(днів)");
        // . . .
    }
}
```

Як бачимо, результати запитів ідентичні. На перший погляд може скластись враження, що другий спосіб ініціалізації масивів є зручнішим, простішим та ефективнішим. Можливо, в цьому і є частка істини, та такий застосунок слід реалізовувати тільки у випадках, коли чітко відомо остаточну кількість елементів масиву та те, що значення елементів масиву не будуть змінюватись (масив є статичним).

Для кращої уяви про структуру одновимірних масивів розглянемо ще один приклад програми визначення середнього значення декількох чисел:

```
public class Average {
    public static void main(String[] args) {
        double nums[] = { 10.1, 11.2, 12.3, 13.4,
14.5 };
        double result = 0;
        for (int i = 0; i < 5; i++) {
            result = result + nums[i];
        }
        System.out.print("Середнє значення рівне
"+ result / 5);
    }
}
```

```
Результат роботи:
Середнє значення рівне 12.2999
```

Слід пам’ятати, що за умови звернення до визначеного елемента масиву, керуюча система Java здійснює перевірку, чи зазначений в

запиті індекс відповідає діапазону наявних елементів. Будь-яка спроба звернутись до елемента масиву за межами його діапазону (в нашому прикладі вказати від'ємний індекс або індекс більше 11) призведе до помилки при виконанні програми.

8.2. Багатовимірні масиви

В програмуванні ***багатовимірні масиви*** являють собою масиви масивів. Зовнішній вигляд та принцип дії багатовимірних масивів ідентичний звичайним масивам проте із низкою незначних відмінностей. При оголошенні змінної багатовимірного масиву для зазначення кожного додаткового індексу використовують окремий ряд квадратних скобок. Наприклад в наведеній стрічці коду оголошується змінна двовимірного масиву `twoD`:

```
int twoD[][] = new int[4][5];
```

В наведеній стрічці коду зарезервовано пам'ять для масиву розміром 4x5, який присвоєно змінній `twoD`. Концептуальний вигляд оголошеного масиву набуває вигляд матриці (рис. 8.1).

Рисунок 8.1 – Концептуальне представлення масиву розміром 4x5

В наступному прикладі розглянемо двовимірний масив із двома процедурами: наповнення та виведення вмісту у консоль. Як перша, так і друга процедура виконуються почергово: спочатку елементи наповнюються зліва направо, а потім зверху вниз. Виведення значень реалізовано за тим самим порядком:

```
public class TwoDArray {
```

```
        public static void main(String[] args) {
            int twoD[][] = new int[4][5];
            int i, j, k = 0;
            for (i = 0; i < 4; i++) {
                for (j = 0; j < 5; j++) {
                    twoD[i][j] = k;
                    k++;
                }
            }
            for (i = 0; i < 4; i++) {
                for (j = 0; j < 5; j++) {
                    System.out.print(twoD[i][j] + " ");
                }
                System.out.println();
            }
        }
    }
```

```
Результат роботи програми :
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

При резервуванні пам’яті під багатовимірний масив обов’язково вказувати параметри тільки для першого (лівого) виміру масиву. А для кожного наступного виміру пам'ять можливо резервувати окремо. Наприклад, в наведеному нижче прикладі коду пам'ять резервується тільки для першого виміру масиву twoD при його оголошенні. А резервування пам’яті для другого виміру реалізується власноруч:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Якщо в наведеному прикладі окреме резервування пам’яті для другого виміру масиву не дає жодних переваг, то в інших випадках така процедура може бути корисною. Наприклад, за умови

власноручного резервування пам'яті для окремих вимірів масиву, зовсім не обов'язково резервувати однакову кількість елементів для кожного виміру. Розробнику надається повна свобода при визначенні довжини кожного виміру масиву, оскільки в багатовимірних масивах кожен вимір розцінюється як окремий масив в масиві (масив масивів). Для прикладу, в наступному взірці програмного коду розглядається двовимірний масив з різною довжиною другого виміру:

```
public class TwoDArray {

    public static void main(String[] args) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];

        int i, j, k = 0;

        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                twoD[i][j] = k;
                k++;
            }
        }
        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                System.out.print(twoD[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
Результат роботи програми:
0
1 2
3 4 5
6 7 8 9
```

На рисунку 8.2 представлено графічну візуалізацію ініціалізованого двовимірного масиву із довільною довжиною другого масиву.

[0] [0]			
[1] [0]	[1] [1]		
[2] [0]	[2] [1]	[2] [2]	
[3] [0]	[3] [1]	[3] [2]	[3] [3]

Рисунок 8.2 – Двовимірний масив з довільною довжиною другого виміру

Аналогічно одновимірним масивам, багатовимірні також можна ініціалізовувати як окремо після оголошення, так і під час оголошення з використанням фігурних дужок. Для цього слід включити ініціалізатор кожного виміру масиву в окремий рядок фігурних дужок. В наведеному далі прикладі створюється матриця, в якій кожен елемент містить результат добутку індексів рядка та стовбця. Слід також відзначити, що в ініціалізаторах масиву можливо використовувати не лише значення, а й математичний вираз:

```
public class TwoDArray {

    public static void main(String[] args) {
        double m[][] = {
                { 0 * 0, 1 * 0, 2 * 0, 3 * 0 },
                { 0 * 1, 1 * 1, 2 * 1, 3 * 1 },
                { 0 * 2, 1 * 2, 2 * 2, 3 * 2 },
                { 0 * 3, 1 * 3, 2 * 3, 3 * 3 }
                };

        int i, j;
        for (i = 0; i < 4; i++) {
            for (j = 0; j < 4; j++)
                System.out.print(m[i][j] + "    ");
            System.out.println();
        }
    }
}
Результат роботи програми:
0.0    0.0    0.0    0.0
```

```
0.0    1.0    2.0    3.0
0.0    2.0    4.0    6.0
0.0    3.0    6.0    9.0
```

Як бачимо з наведеного прикладу, кожна стрічка масиву ініціалізується у відповідності із значеннями, вказаними в списках ініціалізації. Розглянемо ще один приклад застосування багатовимірного масиву. Створимо тривимірний масив розміром 3x4x5, після чого кожен елемент масиву заповнимо результатом добутку його індексів. Для наочності результат роботи виведемо у консоль.

```
public class TreeDMatrix {

    public static void main(String[] args) {
        int threeD[][][] = new int[3][4][5];
        int i   , j, k;
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 4; j++) {
                for (k = 0; k < 5; k++) {
                    threeD[i][j][k] = i * j * k;
                }
            }
        }

        for (i = 0; i < 3; i++) {
            for (j = 0; j < 4; j++) {
                for (k = 0; k < 5; k++) {

 System.out.print(threeD[i][j][k] + " ");

                }
                System.out.println();
            }
            System.out.println();
        }
    }
}


Результат роботи програми:
0 0 0 0 0
```

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

І на завершення розглянемо альтернативні варіанти оголошення масивів. Оголошення може мати такий варіант

```
тип [] ім'я_змінної;
```

Наприклад, наступні два оголошення рівносильні:

```
int a1 [] = new int[3];
int [] a2 = new int[3];
```

Обидві форми оголошення також правдиві для багатовимірних масивів:

```
char twooD1 [] = new char[3][4];
char [] twooD2 = new char[3][4];
```

Альтернативну форму оголошення масивів раціонально використовувати при оголошення одразу декількох масивів, наприклад:

```
int [] nums1, nums2, nums3;
```

Така форма оголошення є тотожною для:

```
int  nums1 [], nums2 [], nums3 [];
```

Альтернативна форма оголошення масивів також зручна для передачі масиву в якості типу даних, який повертає метод. В програмуванні мовою Java застосовуються обидві форми оголошення масивів.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И. Д. Вильяме", 2015. – 1376 с.

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

Запитання до лекції

1. Що таке масив?
2. Як відбувається резервування області памяті під масив?
3. Вкажіть відмінності між одновимірними і багатовимірними масивами.

ЛЕКЦІЯ 9.
ПОТОКИ ВВЕДЕННЯ/ВИВЕДЕННЯ ТА РЯДКИ В JAVA

План:

9.1. Потоки введення та виведення. Клас Scanner
9.2. Рядки. Клас String

9.1. Потоки введення та виведення. Клас Scanner

На етапі вивчення основ програмування настав час створення програм із можливістю введення з консолі певних вхідних значень, потрібних для роботи програми. Для введення даних в Java використовують клас Scanner. Для початку роботи з класом його необхідно імпортувати в ту програму, де він буде використовуватись.

Клас містить в собі методи для зчитування чергового символу зі стандартного потоку введення, а також для перевірки існування такого символу.

Для роботи з потоком введення необхідно створити об’єкт (екземпляр) класу Scanner, вказавши на те, з яким потоком введення він працюватиме. Стандартний потік введення (клавіатура) в Java представлений об’єктом – System.in. А стандартний потік виведення (дисплей) – вже знайомим об’єктом – System.out. Існує ще один стандартний потік для виведення помилок – System.err, проте наразі ми цей потік не розглядатимемо.

Розглянемо приклад застосування класу Scanner з метою зчитування числа з потоку введення та збереження його у змінну:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner (System.in); //
створюєм об'єкт класу Scanner
        int i;
        System.out.print("Введеіть ціле число: ");
        if (sc.hasNextInt()) { // повертає істину,
якщо з потоку введення можна
                                   //зчитати значення типу int
```

```
            i = sc.nextInt(); // зчитує ціле число з
потоку введення та зберігає
                                // його у змінну
            System.out.println(i * 2); // збільшуємо
введене число удвічі
        } else {
            System.out.println("Ви ввели не ціле число");
        }
    }
}
```

Метод hasNextInt(), перевіряє чи можливо зчитати з потоку введення число типу int, а метод nextInt – зчитує його. Якщо намагатись зчитувати значення без попередньої перевірки, то під час виконання програми можлна отримати помилку. Наприклад:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        double i = sc.nextDouble(); // якщо ввести
літеру, отримаємо помилку в процесі виконання
програми
        System.out.println(i / 3);
    }
}
```

Існує також метод nextLine(), який дозволяє зчитувати послідовність символів (стрічку), а це означає, що отримане через цей метод значення потрібно зберігати в об'єкті класу String. В наступному прикладі створюється два таких об'єкти, до яких записуються введені користувачами значення, а на екран виводиться об'єднання послідовності символів:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
            String s1, s2;
            s1 = sc.nextLine();
            s2 = sc.nextLine();
            System.out.println(s1 + s2);
        }
    }
```

З метою перевірки залишку у потоці введення будь-яких символів використовують метод hasNext().

9.2. Рядки. Клас String

Рядки в Java – це послідовність символів Юнікоду ("Абвгґд"), інакше кажучи – послідовність символів. В багатьох мовах програмування рядки зберігаються у масивах. Проте в Java рядки представляють собою окремий об'єктний тип (String – це об’єкт).

Оголошення змінної і присвоєння рядка відбувається у традиційний спосіб:

```
    String str = "Це рядок";
```

Як вже відомо, літерали в стрічках беруться у подвійні лапки ("Це рядок").

Конкатенація – це поєднання двох рядків, що здійснюється за допомогою оператора "+". Приклад:

```
    String str = "Це рядок";
    String str = "Це"+"рядок";
```

В результаті програма видасть аналогічний результат при виведенні як першої, так і другої стрічки.

При використанні конкатенації рядків з іншими типами даних відбувається автоматичне приведення до типу String:

```
public class Main {

    public static void main(String[] args) {
        String str = "цифра " + 5; //String + int дає
String "цифра 5"
        System.out.println(str);
    }
}
```

Також з'єднання двох рядків можна здійснити за допомогою методу concat():

```
String strEnd = "рулить";
String str = "Java ".concat(strEnd); // результат:
"Java рулить"
```

В Java об'єкти класу String не можна змінювати. На перший погляд, це додає проблем при роботі, але насправді це не так. Не можна змінювати сам рядок в пам'яті комп'ютера, але змінній, яка посилається на певний рядок, можна призначити інший рядок. Приклад:

```
String str = "Це";
String str2 = "рядок";
String str3 = "555";
str = str3; //так можна
str = str + " " + str2;    //і так можна
```

Дію обмеження можна відчути, наприклад, якщо замінити літеру "е" іншою, або змінити її регістр. В інших мовах це можна зробити без проблем. В Java потрібно утворити новий рядок. Скопіювавши, наприклад, літеру "Ц" та додавши до неї "Е". Рядок, на який вже не посилається жодна змінна, буде видалений з пам'яті комп'ютера автоматичним прибиральником сміття Java.

Якщо все ж таки необхідна маніпуляція з рядком напряму, для таких цілей існують споріднені із String класи. Зокрема, StringBuffer — корисний при роботі з великими об'ємами текстових даних, читання з файлу тощо.

З метою здійснення різноманітних операцій зі стрічками (пошук, заміна тощо) в класі String існує чималий набір методів. Розглянемо декілька прикладів застосунку найбільш популярних методів класу.

Щоб дізнатися ***довжину рядка*** використовують метод length():

```
public class Main {

   public static void main(String[] args) {
       String str = "Це рядок";
       int strLength = str.length();
       int str2Length = "Це рядок".length();
//можна і так
       System.out.println(strLength);
```

```
            System.out.println(str2Length);
        }
    }
```

В результаті роботи програми в консоль буде виведено (в обох випадках): 8.

Отже, виклик методу відбувається використанням оператора «.»

Для того, щоб одержати частину рядка (***підрядок***), можна скористатися методом substring(pos1, pos2). Приклад:

```
String greeting = "Hello";
String s = greeting.substring(0,3); //скопіювати з greeting символи від 0 до 3: "Hel"
```

Приклад зміни рядка за допомогою методу substring та контамінації «+»:

```
public class Main {

    public static void main(String[] args) {
        String greeting = "Hello";
        String s = greeting.substring(0,3)+ "p!";
        System.out.println(s);
    }
}
```

В результаті рядок «Hello» буде видозмінено на «Help!». В прикладі відбувається не заміна рядка, а присвоєння змінній іншого рядка.

З метою ***порівняння рядків*** на рівність використовують метод equals(). Приклад:

```
public class Main {

    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = str1.substring(0,3)+ "p!";
        System.out.println(str1.equals(str2));
    }
}
```

Метод equals() повертає true, якщо стрічки однакові (рівні). В наведеному прикладі у результаті порівняння рядків в консоль буде виведено false.

Замість змінних str1 та str2 можливе використання рядкових констант. Наприклад:

```
"Hello".equals(str2);
```

Згаданий метод порівнює рядки з врахуванням регістру символів. Для того, щоб при порівнянні не враховувався регістр, існує метод equalsIgnoreCase():

```
"Hello".equalsIgnoreCase("hello");
```

Не рекомендовано використовувати оператор == з метою перевірки рядків на рівність. Застосування цього оператора дозволятиме лише перевірити адресу розташування рядків в пам’яті. Результат може бути неоднозначним.

Часто постає необхідність ***явного приведення типу*** String до інших типів і навпаки. Наприклад, ви вводите з клавіатури певне число у вигляді рядка символів, а для проведення обчислень його потрібно привести або до типу int (якщо це цілочисельне число), або ж до float чи double (якщо дробове). Для таких випадків існує ряд статичних методів valueOf(), які наявні в класі String, а також в класах, які реалізовують числові типи Float, Double та ін.

Наступна програма вимагає введення числа з клавіатури і виводить результат множення введеного числа на 36:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
         int intNumber = 36;

            System.out.print("Введіть число: ");
            //зчитуємо число з клавіатури
            Scanner in = new Scanner(System.in);
            String doubleStr = in.next();
            System.out.println("Ви ввели: " +
doubleStr);
         //Java не розуміє дробових значень, записаних
         через «,» //шукаємо «,»
```

```
            int index = doubleStr.indexOf(",");
            if (index >= 0) {
                System.out.println("Кома у позиції: "
+ index);
                //замінити кому крапкою
             doubleStr = doubleStr.replace(',', '.');
            }
            //Перетворюємо int у рядок тексту
            String strNumber = String.valueOf(intNumber);
            //Приєднуємо число до рядка через метод
concat (хоча можна і
            //оператором "+")
            String strOut = "*".concat(strNumber) + "=";
            //Перетворюємо введений рядок тексту у число
            double number = Double.valueOf(doubleStr);
            number = number * intNumber; //множимо
введене число на 36
            System.out.println(doubleStr + strOut +
number);
            }
    }
```

Результат роботи програми:

```
Введіть число: 2,5
Ви ввели: 2,5
Кома у позиції: 1
2.5*36=90.0
```

У наведеному прикладі для знаходження коми використано метод int indexOf (String str), а для заміни коми на крапку метод String replace (char oldChar, char newChar). Якщо не зробити таку заміну, при приведенні типу String до Double виникне помилка виконання і програма завершиться аварійно.

В мові Java рядки організовані як послідовність значень типу char. Значення char представляються в Unicode 16 і являють собою ***кодову одиницю*** (Code Units). Найчастіше вживані символи Unicode подаються однією кодовою одиницею і також являють собою кодові

точки (Code Points). В той же час специфічні рідковживані символи представляються двома кодовими одиницями. Ці дві кодові одиниці будуть являти собою не дві, а одну кодову точку. Якщо в рядку є такі символи, то метод length() повертатиме неправильну символьну довжину: фактично повертатиметься кількість кодових одиниць, а не кількість справжніх символів (кодових точок).

Тому, якщо такі символи наявні в рядках, необхідно використовувати методи для роботи з кодовими точками (такі методи наведені в таблиці 9.1). Наприклад, щоб визначити справжню кількість символів, потрібно визначити кількість кодових точок в рядку методом codePointCount:

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

Клас String в Java містить понад 50 методів, які можуть бути корисні при створення програм. Нижче наведено список найбільш корисних методів.

Таблиця 9.1

Методи класу String

Тип повернення	**Назва методу та його аргументи**	**Опис**
1	2	3
char	charAt(int index)	Повертає char значення за вказаним індексом
int	codePointAt(int index)	Повертає символ (Unicode code point) за вказаним індексом
int	codePointBefore(int index)	Повертає символ (Unicode code point) перед вказаним індексом
int	codePointCount(int beginIndex, int endIndex)	Повертає кількість кодових точок Unicode у зазначеному інтервалі в рядку
int	compareTo(String anotherString)	Порівнює два рядки лексикографічно
int	compareToIgnoreCase(String str)	Порівнює два рядки лексикографічно, ігноруючи різницю в регістрах літер
String	concat(String str)	Приєднує зазначений рядок str в кінець рядка
boolean	contains(CharSequence s)	Повертає true, тільки якщо рядок містить зазначену послідовність значень char

Продовження таблиці 9.1

1	2	3
boolean	contentEquals(CharSequence cs)	Порівнює рядок із зазначеною послідовністю символів(CharSequence)
boolean	endsWith(String suffix)	Перевіряє чи рядок закінчується зазначеним суфіксом
boolean	equals(Object anObject)	Порівнює рядок із зазначеним об'єктом
boolean	equalsIgnoreCase(String anotherString)	Порівнює рядок з іншим рядком, ігноруючи регістр
byte[]	getBytes()	Кодує рядок у послідовність байт, використовуючи символьний набір(charset) по замовчуванню, результат зберігається у новому байтовому масиві
byte[]	getBytes(Charset charset)	Кодує рядок у послідовність байт, використовуючи наданий символьний набір(charset), результат зберігається у новий байтовий масив
byte[]	getBytes(String charsetName)	кодує рядок у послідовність байт, використовуючи названий символьний набір, результат зберігається у новому байтовому масиві
void	getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Копіює символи з рядка у символьний масив
int	hashCode()	Повертає ХЕШ-код для рядка
int	indexOf(int ch)	Повертає індекс першого входження зазначеного символу в рядок
int	indexOf(int ch, int fromIndex)	Повертає індекс символу у рядку, починаючи пошук із зазначеного індексу
int	indexOf(String str)	Повертає індекс першого знаходження підрядка у рядку
int	indexOf(String str, int fromIndex)	Повертає індекс в рядку підрядка, починаючи пошук із зазначеної позиції
boolean	isEmpty()	Повертає true, тільки тоді, якщо довжина(length()) становить 0.
int	lastIndexOf(int ch)	Повертає індекс останнього входження зазначеного символу в рядку

Продовження таблиці 9.1

1	2	3
int	lastIndexOf(int ch, int fromIndex)	Повертає індекс останнього входження зазначеного символу, шукаючи його із зазначеної позиції в рядку
int	lastIndexOf(String str)	Повертає індекс останнього входження зазначеного підрядка
int	lastIndexOf(String str, int fromIndex)	Повертає індекс останнього входження зазначеного підрядка, шукаючи його із зазначеного індексу у рядку
int	length()	Повертає довжину даного рядка
boolean	matches(String regex)	Повідомляє чи відповідає даний рядок заданому регулярному виразу
String	replace(char oldChar, char newChar)	Повертає новий рядок, заміняючи усі входження символу(oldChar) в рядку на новий символ (newChar)
String	replace(CharSequence target, CharSequence replacement)	Заміняє в рядку підрядок target новою послідовністю replacement
String	replaceAll(String regex, String replacement)	Заміняє кожен підрядок в рядку, що співпадає з регулярним виразом(regex), новим підрядком(replacement)
String	replaceFirst(String regex, String replacement)	Заміняє перший підрядок, що відповідає заданому регулярному виразу, на підрядок для заміни
String[]	split(String regex)	Розбиває рядок за певним правилом, поданим у регулярному виразі
String[]	split(String regex, int limit)	Розбиває рядок за певним правилом, поданим у регулярному виразі
boolean	startsWith(String prefix)	Перевіряє чи поточний рядок починається з заданого префікса
String	substring(int beginIndex)	Повертає підрядок з поточного рядка
String	substring(int beginIndex, int endIndex)	Повертає підрядок з поточного рядка
char[]	toCharArray()	Перетворює рядок у новий символьний масив
String	toLowerCase()	Перетворює усі символи рядка у нижній регістр, використовуючи locale правило за замовчуванням

Продовження таблиці 9.1

1	2	3
String	toLowerCase(Locale locale)	Перетворює усі символи рядка у нижній регістр, використовуючи правило Locale.
String	toUpperCase()	Конвертує всі символи рядка у верхній регістер, використовуючи locale правило за замовчуванням
String	toUpperCase(Locale locale)	Перетворює усі символи рядка у верхній регістр, використовуюче правило, подане у Locale.
String	trim()	Повертає копію рядка, усуваючи пробіли спереду і ззаду рядка
static String	valueOf(boolean b)	Повертає рядкове представлення аргументу boolean типу
static String	valueOf(char c)	Повертає рядкове представлення char аргументу
static String	valueOf(char[] data)	Повертає рядкове представлення масиву типу char
static String	valueOf(double d)	Повертає рядкове представлення double аргументу
static String	valueOf(float f)	Повертає рядкове представлення float аргументу
static String	valueOf(int i)	Повертає рядкове представлення int аргументу.
static String	valueOf(long l)	Повертає рядкове представлення аргументу типу long
static String	valueOf(Object obj)	Повертає представлення об'єкту у вигляді рядка

Для наочності розглянемо приклади застосування деяких із наведених методів:

```
public class Main {
    public static void main(String[] args) {
        String s1 = "firefox";
        System.out.println(s1.toUpperCase()); // виведе
"FIREFOX"
        String s2 = s1.replace('o', 'a');
        System.out.println(s2); // виведе «firefax»
        System.out.println(s2.charAt(1)); // виведе «i»
        int i;
        i = s1.length();
```

```
        System.out.println(i); // виведе 7
        i = s1.indexOf('f');
        System.out.println(i); // виведе 0
        i = s1.indexOf('r');
        System.out.println(i); // виведе 2
        i = s1.lastIndexOf('f');
        System.out.println(i); // виведе 4
    }
}
```

Висновок: оволодіння матеріалом лекції необхідне для отримання фундаментальних знань для подальшого вивчення складніших питань, пов'язаних з динамічною ініціалізацією змінних, а також роботи із стрічками та класами. В лекції вперше розглянуто питання екземпляру класу та виклику методів до відповідного об'єкту, що необхідно для подальшого вивчення основ об'єктно-орієнтованого програмування.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильямс", 2015. – 1376 с.
2. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java
3. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java
4. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java

Запитання до лекції

1. З якою метою використовують клас Scanner?
2. Які методи містить в собі клас Scanner?
3. Що являють собою рядки в Java?
4. Які методи містить в собі клас String?

ЛЕКЦІЯ 10.
ЧИСЛОВІ МЕТОДИ В JAVA

10.1. Математичний клас Math та його методи
10.2. Псевдовипадкові числа
10.3. Методи класу чисел

10.1. Математичний клас Math та його методи

Розробник програмного забезпечення мовою Java має доступ до множини готових (бібліотечних) класів та їх методів, корисних в процесі реалізації різноманітних математичних операцій. Наявність бібліотечних рішень дозволяє швидко та якісно вирішувати різноманітні типові задачі.

Основним класом, який містить в собі низку корисних числових методів (математичних функцій) є клас Math.

Для виклику методів класу Math потрібно вказати ключове слово класу та застосувати оператор «.», після чого користувачеві буде доступний перелік всіх методів. Приклад:

```
public class Main {

    public static void main(String[] args) {
        int n = 5;
        Math.abs(n);
    }
}
```

Далі розглянемо основні методи згаданого класу та їх призначення.

1. Math.abs(n) — повертає модуль числа n.

```
public class Main {

    public static void main(String[] args) {

        int n = -5;
        System.out.println(Math.abs(n)); //
результат 5
    }
}
```

2. Math.round(n) — повертає ціле число, найближче до числа n (округлює n).

```
public class Main {
    public static void main(String[] args) {
        double n = 5.5;
        System.out.println(Math.round(n)); // 
результат 6
    }
}
```

3. Math.rint(n) — повертає дробове число, найближче до числа n та рівне математичному цілому числу.

```
public class Main {
    public static void main(String[] args) {
        double n = 5.5;
        System.out.println(Math.rint(n)); // 
результат 6.0
    }
}
```

4. Math.ceil(n) — повертає найближче до числа n праве число з нульовою дробовою частиною.

```
public class Main {
    public static void main(String[] args) {
        double n = 5.4;
        System.out.println(Math.ceil(n)); // 
результат 6.0
    }
}
```

5. Math.toRadians — повертає число n, задане в градусах, у радіанах (переводить градуси в радіани).

```
public class Main {
    public static void main(String[] args) {
        double degrees = 30.0;
```

```
            double radians = Math.toRadians(degrees);
            System.out.println(radians); //результат 
0.5235987755982988
        }
    }
```

6. Math.cos(n), Math.sin(n), Math.tan(n) — тригонометричні функції sin, cos та tg аргументу n, ***вказаного в радіанах*** .

```
    public class Main {

        public static void main(String[] args) {

            double degr = 30.0;
            double radians = Math.toRadians(degr);
            System.out.println(Math.sin(radians)); // 
результат 0.5
            System.out.println(Math.cos(radians)); // 
результат 0.8660
            System.out.println(Math.tan(radians)); // 
результат 0,5774
        }
    }
```

7. Math.acos(n), Math.asin(n), Math.atan(n) — зворотні тригонометричні функції, повертають значення кута в радіанах.

```
    public class Main {

        public static void main(String[] args) {

            double radiansSin = 0.4999;
            double radiansCos = 0.8660;
            double radiansTan = 0.5774;
            System.out.println(Math.asin(radiansSin)); 
// результат 0.523
            System.out.println(Math.acos(radiansCos)); 
// результат 0.523
            System.out.println(Math.atan(radiansTan)); 
// результат 0,523
        }
    }
```

8. Math.toDegrees() — повертає в градусах значення кута, заданого у радіанах (переводить радіани у градуси).

```
public class Main {

    public static void main(String[] args) {

        double radiansSin = 0.4999;
        double radiansCos = 0.8660;
        double radiansTan = 0.5774;
        double sin = Math.asin(radiansSin);
        double cos = Math.acos(radiansCos);
        double tan = Math.atan(radiansTan);
        System.out.println(Math.toDegrees(sin));
// наближено 30
        System.out.println(Math.toDegrees(cos));
// наближено 30
        System.out.println(Math.toDegrees(tan));
// наближено 30
    }
}
```

9. Math.sqrt(n) — повертає квадратний корінь числа n.

```
public class Main {

    public static void main(String[] args) {

        int i = 16;
        int j = 9;

        System.out.println(Math.sqrt(i)); //
результат 4.0
        System.out.println(Math.sqrt(j)); //
результат 3.0
    }
}
```

10. Math.pow(n, b) — повертає значення числа n в степені b.

```
public class Main {

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

```
            for (int i = 1; i <= 10; i++) {
                System.out.println(i + " в квадраті
рівне " + Math.pow(i, 2));
            }
        }
    }
```

Результат:

```
1 в квадраті рівне 1.0
2 в квадраті рівне 4.0
3 в квадраті рівне 9.0
4 в квадраті рівне 16.0
5 в квадраті рівне 25.0
6 в квадраті рівне 36.0
7 в квадраті рівне 49.0
8 в квадраті рівне 64.0
9 в квадраті рівне 81.0
10 в квадраті рівне 100.0
```

11. Math.log(n) — повертає значення натурального логарифма числа n. Math.log10(n) — повертає значення десяткового логарифма числа n.

```
    public class Main {

        public static void main(String[] args) {

            double n = 11.635;
            System.out.println(Math.log(n)); //
результат 2,454
        }
    }
```

12. Math.exp(n) — повертає натуральний логарифм числа n за основою числа e.

```
    public class Main {

        public static void main(String[] args) {

            double n = 11.635;

            double m = Math.log(n);
```

```
            System.out.println(Math.exp(m)); //
результат 11,635
        }
    }
```

13. Math.min(n, m), Math.max(n, m) — повертає більше (менше) з двох аргументів.

```
    public class Main {
        public static void main(String[] args) {
            System.out.println(Math.min(-12, -6)); //
результат -12
            System.out.println(Math.max(5, 7.58)); //
результат 7.58
        }
    }
```

Крім функцій в класі Math часто використовують дві константи:

1. Math.PI – число «Пі».
2. Math.E – число Неппера (основа експоненціальної функції).

10.2. Псевдовипадкові числа

Метод Math.random() в Java використовують для генерації випадкових чисел в діапазоні від 0.0 до 1.0. Стандартне застосування методу повертає double додатнє значення, в межах 0.0 <= Math.random() < 1.0. Приклад:

```
    public class Main {
        public static void main(String[] args) {
        System.out.println("1 випадкове число: " +
Math.random()); // 0.9362192604319092
        System.out.println("2 випадкове число: " +
Math.random()); // 0.12966189312475418
        System.out.println("3 випадкове число: " +
Math.random()); // 0.506620737051696
        }
    }
```

Можна змінювати межі діапазону для генерування псевдовипадкових чисел. Для прикладу, щоб згенерувати випадкове число в діапазоні від 0 до 10 необхідно представити метод в наступному вигляді:

```
System.out.println((Math.random() * 10));
```

Приклад:

```
public class Main {
    public static void main(String[] args) {
        System.out.println((Math.random() * 10)); // 
2.268517446966837
        System.out.println((Math.random() * 10)); // 
3.893158265273526
        System.out.println((Math.random() * 10)); // 
0.40220587435187194
    }
}
```

Кожного разу при застосуванні методу random генеруватимуться нові числа із вказаного проміжку чисел.

Щоб згенерувати псевдовипадкове число в межах одиниці, починаючи з цілого числа 3 (від 3 до 4), слід застосувати метод в наступному вигляді:

```
System.out.println(Math.random()+3);
```

Аби збільшити проміжок, до прикладу, від 0 до 5, необхідно застосувати:

```
System.out.println(Math.random()*5);
```

Для виведення псевдовипадкового цілого числа слід застосовувати метод з явним приведенням:

```
System.out.println((int)(Math.random()*5));
```

Щоб змістити діапазон псевдовипадкових чисел, до прикладу, в проміжок від 3 до 8, необхідно застосувати:

```
System.out.println(Math.random()*5+3);
```

Виведення цілого числа з проміжку від –5 до 5 проводиться наступним чином:

```
System.out.println((int)(Math.random()*11) - 5);
```

Для кращої наочності розглянемо послідовність перетворень, необхідних для отримання цілого випадкового числа з відрізку [-1;3]:

`Math.random()`		
	[0; 1)	
`Math.random()*5`	↓	Перемноження кінців на 5 (розширення діапазону)
	[0; 5)	
`(int)(Math.random()*5)`	↓	Відкидаємо цілу частину (всі значення >4)
	[0; 4]	
`(int)(Math.random()*5)-1`	↓	Пересуваємо обидва кінці на 1 вліво
	[-1; 3]	

10.3. Методи класу чисел

При написанні програм інколи виникає необхідність переведення числового значення у рядок і навпаки – переведення рядку в числове значення, або ж порівняння двох числових значень тощо. Розглянутий клас Math не наділений такими методами. Для цього в Java існують методи класу чисел. Давайте з'ясуємо, що це за клас.

Насправді, усі примітивні типи даних, якими ми користуємось на практиці, належать класам-оболонкам Integer, Long, Byte, Double, Float, Short. В свою чергу, класи-оболонки є підкласами абстрактного класу чисел Number.

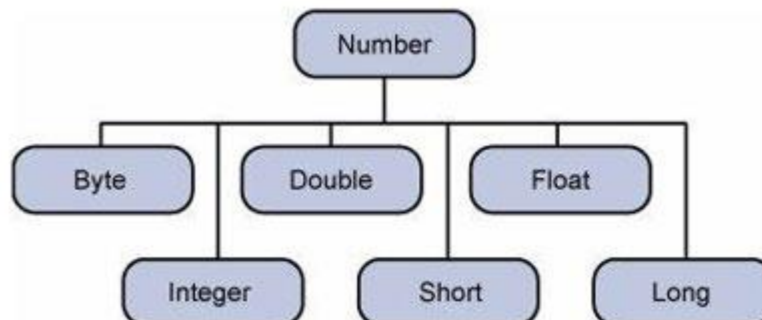

Рисунок 10.1 – Структура абстрактного класу Number

Об'єкт класу-оболонки містить в собі відповідний примітивний тип даних. В програмуванні інколи виникає необхідність використання об'єктів замість примітивних типів даних. Конвертування примітивних типів в об'єкт називається упакуванням. Для цього необхідно передати його конструктору значення примітивного типу даних. Об'єкт оболонки також можна перетворити назад в примітивний тип даних (розпакування). Клас Number є частиною пакету java.lang.

Далі розглянемо методи класу чисел.

1. xxxValue() — перетворює числове значення об'єкта, який викликає метод, в примітивний тип даних, який повертається з методу.

```
public class Main {
    public static void main(String[] args) {
        Integer x = 5;
        System.out.println(x.byteValue()); // перетворює int в byte та повертає примітивний тип даних byte
        System.out.println(x.doubleValue()); // перетворює int в double та повертає примітивний тип даних double
        System.out.println(x.longValue()); // перетворює int в long та повертає примітивний тип даних long
    }
}
```

Результат:

```
5
5.0
5
```

2. compareTo() — порівнює числовий об'єкт з аргументом. Якщо Integer рівний аргументу, то повертається 0, якщо Integer менший за аргумент – повертається –1, якщо ж Integer більший за аргумент – 1.

```
public class Main {
    public static void main(String[] args) {
```

```
            Integer x = 5;
            System.out.println(x.compareTo(3)); //
результат 1
            System.out.println(x.compareTo(5)); //
результат 0
            System.out.println(x.compareTo(8)); //
результат -1
        }
    }
```

3. equals() – визначає чи дорівнює цілочисельний об'єкт аргументу.

```
    public class Main {
        public static void main(String[] args) {
            Integer x = 5;
            Integer y = 10;
            Integer z = 5;
            Short a = 5;

            System.out.println(x.equals(y)); //
результат false
            System.out.println(x.equals(z)); //
результат true
            System.out.println(x.equals(a)); //
результат false
        }
    }
```

4. valueOf() — перетворює аргумент в потрібний тип даних.

```
    public class Main {
        public static void main(String[] args) {
            Integer x = Integer.valueOf(9);
            Double c = Double.valueOf(5);
            Float a = Float.valueOf("80");
            Integer b = Integer.valueOf("444", 16);

            System.out.println(x); // результат 9
            System.out.println(c); // результат 5.0
            System.out.println(a); // результат 80.0
```

```
            System.out.println(b); // результат 1092
        }
    }
```

5. toString() — перетворює число в рядок.

```
    public class Main {
        public static void main(String[] args) {
        Integer x = 5;
        System.out.println(x.toString()); //
перетворення числа x в рядок
        System.out.println(Integer.toString(12)); //
перетворення int в string
        }
    }
```

6. Math.parseInt() — перетворює рядок в число.

```
    public class Main {
        public static void main(String[] args) {
            int x = Integer.parseInt("9");
            double c = Double.parseDouble("5");
            int b = Integer.parseInt("444", 16);
            System.out.println(x); // результат 9
            System.out.println(c); // результат 5.0
            System.out.println(b); // результат 1092
        }
    }
```

Висновок: оволодіння знаннями та практичними навиками щодо застосування стандартних числових методів у Java значно спрощує та пришвидшує процес створення програм. Стандартні методи існують для того, щоб створювати просту та ефективну логіку програмованих додатків.

Література:

1. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java

2. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java.

Запитання до лекції

1. Опишіть призначення класу Math та його методи
2. Як відбувається генерування випадкових чисел в Java?
3. Опишіть структуру класу Number.
4. Які методи містить в собі клас Number?

ЛЕКЦІЯ 11.
ВВЕДЕННЯ В КЛАСИ

11.1. Знайомство з класами
11.2. Конструктори

11.1. Знайомство з класами

Класи використовувались в прикладах програм ще з початку вивчення мови програмування Java. Але в розглянутих прикладах демонструвались найпримітивніші форми класів. Ці класи слугували лише контейнерами для методу `main()`, щоб ознайомити нас з синтаксисом та основами мови Java. Найважливішою особливістю класу є те, що він є шаблоном для створення об'єктів (екземплярів класу). Існує чудовий приклад цієї особливості: клас – це ніби форма для випічки печива. Оскільки об'єкт є екземпляром класу, то в Java поняття об'єкт та екземпляр є тотожними.

При визначенні класу оголошується його конкретна форма та сутність. Для цього необхідно вказати дані, що міститиме клас, а також код, який використовуватиме ці дані. Можуть зустрічатись класи, які містять лише код або лише дані, проте більшість класів в реальних програмах містять обидва компоненти.

Якщо говорити термінологічною мовою, то дані – це змінні екземпляра класу (поля класу), а код – це методи класу. ***Методи визначають те, що клас вміє, а змінні екземпляра класу (поля) – що він знає.***

Для оголошення класу слугує ключове слово `class`. Розглянемо загальну форму класу із усіма компонентами:

```
class Імя_класу {
    тип змінна_екземпляра1;
    тип змінна_екземпляра2;
    //...

    тип імя_методу1 (список_парметрів) {
        // тіло методу
    }
    тип імя_методу2 (список_парметрів) {
        // тіло методу
    }
```

```
    //...
}
```

В більшості класів дії над змінними екземпляра і доступ до них реалізують методи, визначені в класі. Саме методи, як правило, визначають порядок використання змінних екземпляра класу. Змінні, оголошені в класі, мають назву змінних екземпляра класу, так як кожен екземпляр (об'єкт) містить власні копії оголошених змінних.

Для кращого розуміння розглянемо поняття класу на основі простого прикладу. Розглянемо код класу Box, який визначає три змінні екземпляра: `width`, `height`, `depth`. На цьому етапі клас Box не містить жодних методів:

```
class Box {
    double width;
    double height;
    double depth;
}
```

Клас визначає новий тип даних. В наведеному прикладі новий тип даних має назву `Box`. Це ім'я використовуватиметься для оголошення об'єктів типу `Box`. Слід зауважити, що застосування ключового слова **`class`** створює лише шаблон, проте не конкретний об'єкт. Щоб дійсно створити об'єкт (екземпляр) класу `Box`, потрібно застосувати такий оператор:

```
Box mybox = new Box() ; // створити об’єкт mybox класу Box
```

Після виконання подібного оператора об'єкт `mybox` набуде реальної сутності та стане першим створеним екземпляром класу `Box`.

Кожен об'єкт, створений на основі класу, міститиме власну копію усіх змінних, визначених в класі. В нашому прикладі, кожен об'єкт класу `Box` міститиме змінні екземпляра `width`, `height`, `depth`. Для доступу до змінної екземпляра застосовують оператор «.», який зв'язує ім'я об'єкту з ім'ям змінної екземпляра класу. Наприклад, щоб присвоїти змінній width значення 100, потрібно виконати оператор:

```
mybox.width = 100;
```

Оператор «.» слугує доступом як до змінних екземпляра, так і до методів класу.

Поєднаємо усі розглянуті приклади у вигляді повноцінної програми з використанням класу Box:

```
class Box {
    double width;
    double height;
    double depth;
}

// Клас із оголошенням об’єкту типу Box (екземпляра
класу Box)
class BoxDemo {

    public static void main(String[] args) {

        Box mybox = new Box() ;

        double vol ; // змінна для запису результату
розрахунку об’єму

        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        vol = mybox.width * mybox.height *
mybox.depth; // розрахунок об’єму

        System.out.println ( "Об'єм становить: " + vol
) ;
    }
}
```

Важливо! Після компіляції цієї програми буде створено два файли з вихідним кодом та розширенням *.class: один для класу `Box`, інший – `BoxDemo`. Два класи можуть розміщуватись як в одному, так і в двох окремих файлах з розширенням *.java. На результат виконання програми це не впливатиме. Слід зауважити, що за умови рознесення класів різними файлами, для запуску програми необхідно запускати той файл, який містить `main()`-метод. А за умови розміщення класів в одному файлі, його слід називати ім’ям класу з `main()`-методом, тобто BoxDemo.java.

За умови наявності декількох екземплярів класу, коригування змінних одного екземпляра не впливатиме на змінні іншого екземпляра класу. Приклад:

```
class Box {
    double width;
    double height;
    double depth;
}
class BoxDemo {
    public static void main(String[] args) {
        Box mybox1 = new Box() ;
        Box mybox2 = new Box() ;
        double vol ;
        // присвоєння значень змінним першого екземпляру
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // присвоєння значень змінним другого екземпляру
        mybox2.width = 5;
        mybox2.height = 10;
        mybox2.depth = 7.5;
        vol = mybox1.width * mybox1.height *
mybox1.depth;
        System.out.println ( "Об'єм 1 коробки
становить: " + vol ) ;
        vol = mybox2.width * mybox2.height *
mybox2.depth;
        System.out.println ( "Об'єм 2 коробки
становить: " + vol ) ;
    }
}
```

Результат роботи програми становить:

```
Об'єм 1 коробки становить: 3000.0
Об'єм 2 коробки становить: 375.0
```

Таким чином, змінні одного екземпляру класу ізольовані від змінних іншого екземпляру.

Оголошення об'єктів та посилання на об'єкти. Ми вже знаємо, що під час створення нового класу створюється новий тип даних (посилкового типу), який можна використовувати для оголошення об'єктів даного типу. Але створення об'єктів класу є двоетапним процесом. Спочатку слід оголосити змінну типу класу (в наведеному прикладі це змінну типу Box). Ця змінна не визначає об'єкт, вона є лише змінною, що може посилатись на об'єкт. Після створення змінної необхідно отримати конкретну фізичну копію об'єкта та присвоїти її оголошеній змінній. Це можливо виконати з допомогою оператора new. Цей оператор резервує пам'ять для об'єкта та повертає посилання на нього. Посилання – це адреса об'єкта в пам'яті. Отримане посилання зберігається в оголошеній змінній. Таким чином, оперативна пам'ять під об'єкти в Java виділяється динамічно. Розглянемо описану процедуру детальніше.

Як нам вже відомо, стрічка, наведена нижче, слугує для оголошення об'єкту типу Box з ім'ям mybox:

```
Box mybox = new Box();
```

В наведеній стрічці об'єднуються два етапи щойно описаного процесу. Щоб кожен етап став очевидним, наведену стрічку можна переписати наступним чином:

```
Box mybox; // оголошення змінної типу Box
mybox = new Box() ; // виділення пам'яті для об'єкта типу Box та присвоєння посилання на об'єкт (адреси в пам'яті) змінній mybox
```

В першій стрічці змінна mybox оголошується як шаблон посилання на об'єкт типу Box. При цьому змінна mybox не має посилання на конкретний об'єкт. В наступній стрічці коду виділяється пам'ять під конкретний об'єкт, а змінній mybox присвоюється посилання на цей об'єкт. Після виконання другої стрічки коду, змінну mybox можна використовувати так, наче вона є об'єктом типу Box. Але насправді змінна mybox лише містить адресу комірки в пам'яті конкретного об'єкту типу Box.

Далі розглянемо процеси присвоєння змінним посилань на об'єкти. Для наочності розглянемо приклад:

```
Box b1 = new Box();
Box b2 = b1;
```

Після виконання представленого фрагменту коду дві змінні b1 та b2 будуть мати посилання на один об'єкт. Присвоєння однієї змінної іншій не супроводжується виділенням пам'яті або копіюванням об'єкту. Таке присвоєння призведе лише до того, що змінна b2 матиме посилання на той самий об'єкт, що й змінна b1.

Крім того, що змінні b1 та b2 посилаються на один об'єкт, вони більше не зв'язані жодним чином. Наприклад, якщо в наведеному нижче прикладі присвоїти пусте значення (null) змінній b1, то в результаті лише втратиться зв'язок змінної з об'єктом, не створивши жодного впливу на сам об'єкт та змінну b2:

```
Box b1 = new Box();
Box b2 = b1;
b1 = null;
```

Підсумовуючи наведений матеріал, розглянемо відмінність поняття класу від об'єкта. Клас створює новий тип даних (посилковий), який можливо використовувати для створення об'єктів. Це означає, що клас створює логічний каркас, який визначає взаємозв'язок між його членами. При оголошенні об'єкта класу створюється екземпляр цього класу. Таким чином, клас – це логічна конструкція, а об'єкт володіє фізичною сутністю, тобто займає конкретну область оперативної пам'яті.

11.2. Конструктори

Ініціалізація усіх змінних класу при створенні його екземплярів може виявитись виснажливим та кропітким процесом. Навіть за умови використання службових методів, таких як `setBox`, простіше та ефективніше було б реалізовувати ініціалізацію змінних під час першого згадування про об'єкт (під час його створення). В зв'язку з тим, що необхідність в ініціалізації виникає часто, об'єктам в Java дозволено виконувати власну ініціалізацію при створенні. Автоматична ініціалізація реалізується з допомогою конструктора.

Конструктор ініціалізує змінні екземпляра класу безпосередньо під час створення об'єкту.

Синтаксис конструктора аналогічний синтаксису методів. Ім'я конструктора співпадає з ім'ям класу. Якщо конструктор визначений в класі, то він автоматично викликатиметься при створенні кожного екземпляра після завершення виконання оператора `new`. На відміну від методів, конструктори не мають типу даних, що повертається,

навіть типу **void**. Конструктор ініціює внутрішній стан об’єкта таким чином, щоб код, який створює екземпляр, з самого початку містив повністю ініціалізований та придатний до використання об’єкт.

Розглянутий раніше приклад класу Box можливо переробити із використанням конструктора. В такому разі конструктор буде заміною методу setBox. В першому прикладі розглянемо простий конструктор, який встановлює сталі значення для змінних екземпляра класу.

```
public class Box {
    double width;
    double height;
    double depth;

    // Створюємо конструктор для класу Box та 
встановлюємо сталі
    // значення для усіх параметрів
    Box() {
        width = 10;
        height = 10;
        depth = 10;
    }

    // Створюємо метод для визначення об'єму
    double volume (){
        return width*height*depth;
    }
}

public class BoxDemo {

    public static void main(String[] args) {
        Box mybox1 = new Box ();
        Box mybox2 = new Box ();

        // отримати об’єм першої коробки
        System.out.println(mybox1.volume());

        // отримати об’єм другої коробки
        System.out.println(mybox2.volume());
    }
}
```

Результатом роботи програми буде виведення у консоль:

```
Об'єм складає 1000.0
```

```
Об'єм складає 1000.0
```

В наведеному прикладі, за будь-якого виклику методу `volume` результат буде сталий. Адже конструктором класу `Box` визначені сталі значення усіх змінних. Проте цей приклад наведений лише для наочності того, що за умови застосування конструктора зникає необхідність ініціалізації (присвоєння значень) параметрів об'єкта після його створення в `main()`методі.

Повернемось до оператора створення екземпляра класу. Після ключового слова **new** та ім'я класу, необхідно вказати круглі дужки. Власне, ця операція завжди викликає конструктор класу.

```
Box myboxl = new Box ();
```

Оператор **new** `Box ()` викликає конструктор `Box ().` Якщо конструктор класу не визначено явно, то Java створює для будь-якого класу конструктор за замовчуванням (пустий конструктор). Конструктор створюється автоматично для усіх класів, саме тому за відсутності його явного приведення усі змінні класів (поля) можна ініціалізувати та присвоювати їм значення. Конструктор за замовчуванням ініціалізує всі змінні екземпляра, встановлюючи їм значення. Всі значення за замовчуванням можуть бути нульовими (пустими): для чисельних типів – null, для логічних – false. Зазвичай використання конструкторів за замовчуванням є достатнім для простих класів, проте не для складних. Як тільки в класі буде визначено власний конструктор, конструктор за замовчуванням більше не використовуватиметься.

Більшість конструкторів в програмуванні не встановлюють жодних даних, а лише виконують ініціалізацію об'єкта. Їх називають параметризованими.

В попередньому прикладі конструктор класу `Box()` ініціалізував об'єкт, проте користі від цього було мало, оскільки всі об'єкти отримували однаковий розмір. Тому в Java передбачено спосіб створення об'єктів класу з різними розмірами. Для цього достатньо ввести в конструктор параметри (параметризовані конструктори). Наприклад, в наведеному нижче прикладі визначається параметризований конструктор, який визначає розміри за значеннями параметрів конструктора. Зверніть увагу на порядок створення екземпляра класу (об'єкту):

```
public class Box {
```

```
        double width;
        double height;
        double depth;
        // Створюємо параметризований конструтор
        Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
        }
        // Створюємо метод для визначення об'єму
        double volume (){
            return width*height*depth;
        }
    }
    public class BoxDemo {
        public static void main(String[] args) {
            Box myboxl = new Box (10,20,15);
            Box mybox2 = new Box (3,6,9);
            // отримати обєм першої коробки
            System.out.println("Об'єм складає " +
myboxl.volume());
            // отримати об’єм другої коробки
            System.out.println("Об'єм складає " +
mybox2.volume());
        }
    }
```

Результат роботи програми:

```
Об'єм складає 3000.0
Об'єм складає 162.0
```

Як бачимо, ініціалізація параметрів класу відбувається під час створення екземпляру за допомогою оператора **new**:

```
Box myboxl = new Box (10,20,15);
```

За такого варіанту копії змінних `width`, `height`, `depth` будуть містити значення 10, 20, та 15 відповідно.

Досконалості немає меж, тому йдемо далі. Як відомо, в Java не допускається оголошення двох локальних змінних з однаковими іменами в тій самій області. Тим не менше, допускається існування локальних змінних, імена яких співпадають з іменами змінних екземпляра класу. У випадку, коли ім'я локальної змінної співпадає з ім'ям змінної екземпляра класу, локальна змінна ***перекриває*** змінну екземпляра. Саме тому іменами `width, height та depth` в класі `Box` не були названі параметри конструктора `Box()`. В зворотному випадку ім'я `width`, наприклад, мало б значення формального параметру та перекривало б змінну екземпляру `width`. І навіть попри те, що просто обрати різні імена для локальних змінних і змінних екземпляра класу, існує інший спосіб, більш цивілізований. Ключове слово **`this`** дозволяє посилатись безпосередньо на об'єкт, тому його можна використовувати для вирішення різних конфліктів, які можуть виникати між змінними екземпляру та локальними змінними. В якості прикладу розглянемо конструктор із застосуванням ключового слова **`this`**. В прикладі імена `width, height та depth` слугують для визначення параметрів, а ключове слово **`this`** – для звернення до змінних екземпляра за цими ж іменами.

```
Box(double width, double heighth, double depth) {
    this.width = width;
    this.height = heighth;
    this.depth = depth;
    }
```

В більшості професійно написаних програм розробники дотримуються правила щодо використання в конструкторі слова **`this`**. Таким чином, в програмі використовуються одні і ті ж імена для оголошення одного і того ж параметру, що дозволяє внести більшу ясність в код програми.

До речі, конструктор подібної архітектури також генерується IDE Eclipse, що підтверджує актуальність застосування ключового слова **`this`**. Для автоматичної генерації конструктора після оголошення класу та змінних його екземпляру (полів класу), необхідно натиснути на комбінацію клавіш Shift+Alt+S. Після виконання цих операцій з'явиться контекстне меню, на якому слід обрати Generate Constructor using Fields ...

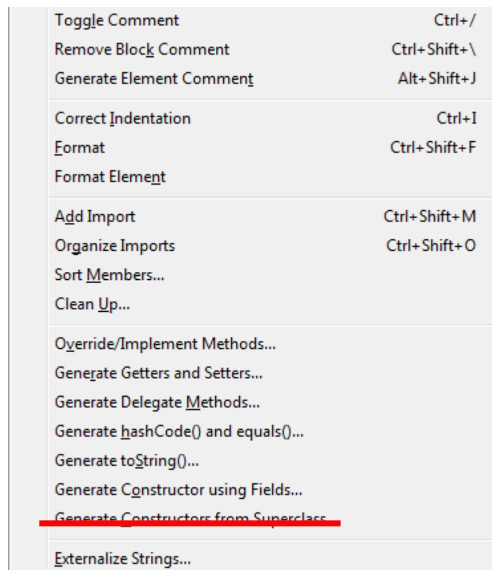

Рисунок 11.1 – Контекстне меню для автоматичного генерування конструктора

Після вибору відповідного меню середовище розробки виводить діалогове вікно з переліком змінних екземпляру класу, можливістю вибору модифікатора доступу, точки вставки (після методів, спочатку тощо) та іншими опціями.

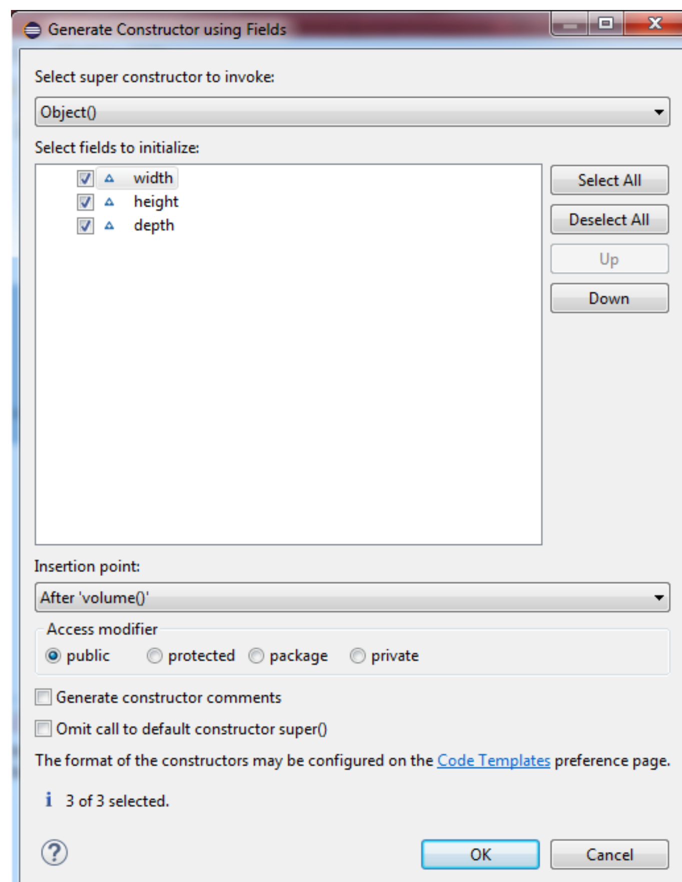

Рисунок 11.2 – Діалогове вікно з параметрами формування конструктора

Після виконання вказаних операцій клас `Box` набуде такого вигляду:

```
public class Box {
    double width;
    double height;
    double depth;

    public Box(double width, double height, double
depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
        double volume() {
        return width * height * depth;
    }
}
```

Слід згадати кількома словами ***збірник сміття***. У випадку відсутності будь-яких посилань на об'єкт, вважається, що цей об'єкт більше не потрібен та пам'ять, яку він займає, можна звільнити. В інших мовах програмування таку операцію потрібно проводити власноруч, а в Java це відбувається автоматично.

Висновок: більшість сучасних додатків (прикладних, мобільних, системних) розробляють із застосуванням об'єктно-орієнтованого підходу, саме тому цей підхід вважається основним в прикладному програмуванні. Зважаючи на це, викладений матеріал є надзвичайно актуальним для початківців.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с.
2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.
3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java
4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java.

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java.

Запитання до лекції

1. Дайте визначення класу. Поняття даних та коду.
2. Як створити новий клас?
3. Що таке конструктор? Опишіть принцип роботи конструктора.
4. Опишіть механізм генерування конструктора.

ЛЕКЦІЯ 12.
ВВЕДЕННЯ В МЕТОДИ

12.1. Типізовані та void-методи
12.2. Методи, що приймають параметри
12.3. Використання об'єктів в якості параметрів методу

12.1. Типізовані та void-методи

Нам вже відомо низку методів, які володіють різноманітними можливостями та функціями. Зокрема, нами розглянуто реалізовані в Java методи класів `Math`, `String` тощо. Проте, частіше всього, в програмуванні на Java виникає необхідність створення власних методів, які реалізують унікальну логіку, не передбачену стандартними методами.

Усі методи, які створюються розробниками програмного забезпечення, володіють тією ж формою, що і метод `main()`, який широко використовувався в наведених раніше прикладах. Слід зауважити, що більшість методів, написаних власноруч, не оголошуватимуться як **`static`** (статичні) та **`public`** (публічними). До речі, метод `main()` не є обов'язковим для загальної форми класу. Класи в Java можуть і не мати `main()` метода, він потрібен лише в тих випадках, коли клас або є відправною точкою (запуском) для виконання усієї програми загалом, або для тестування окремих частин коду під час його написання.

Аби написати метод у класі, потрібно застосувати загальну форму його оголошення, яка виглядає так:

```
тип імя_методу (список_парметрів) {
     // тіло методу
 }
```

де **`тип`** означає визначений тип даних, який повертає (з якими працює) метод. Якщо метод не повертає жодного значення, то його типом має бути **`void`**. «Метод не повертає жодного значення» означає, що метод виконує певний перелік операцій, визначений логікою програмного коду, та не здійснює повернення результату своєї роботи (результат є, але методом не повертається).

За назву методу слугує його ідентифікатор `імя_методу.` А `список_параметрів` розглянемо згодом. Якщо в метода відсутні параметри, то їх список залишається пустим (пусті дужки).

Давайте далі розглянемо визначені моменти на реальних прикладах. Методи, що повертають значення після своєї роботи, подаються з використанням оператора **`return`** за наступною формою:

```
int імя_методу () {
    int a = 5;
    int b = 5;
    int c = a + b;
    return c; // повертає результат роботи методу – 10
}
```

Методи, що не повертають значення, подаються без оператора **`return`** за наступною формою:

```
void імя_методу () {
    int a = 5;
    int b = 5;
    int c = a + b;
    System.out.print (c); // в консоль буде
виведено - 10
}
```

Для того, щоб викликати метод для визначеного об'єкту, необхідно зазначити назву об'єкта та, використовуючи оператор «.», здійснити виклик самого методу, наприклад:

```
myboxl.volume ();
```

Розглядаючи приклади попередньої теми, можна стверджувати, що клас `Box` може краще реалізовувати розрахунок об'єму, аніж клас `BoxDemo,` адже не нагромаджує `main()` – методу. Логічніше, якщо б такий розрахунок виконувався в класі `Box`, оскільки об'єм залежить від розмірів, що представлені як змінні в тому ж класі `Box`. Для цього в клас `Box` необхідно ввести метод `volume()`. Приклад:

```
// файл Box.java
class Box {
    double width;
    double height;
```

```
        double depth;
        void volume() {  // метод розрахунку та виводу
    об’єму
            System.out.print("Об'єм становить ");
            System.out.println(width * height *
    depth);
        }
    }

    // файл BoxDemo.java
    class BoxDemo {
        public static void main(String[] args) {
            Box myboxl = new Box();
            Box mybox2 = new Box();
            myboxl.width = 10; // ініціалізація
    змінних екземпляра
            myboxl.height = 20; // класу myboxl
            myboxl.depth = 15 ;
            mybox2.width = 3; // ініціалізація змінних
    екземпляра
            mybox2.height = 6; // класу mybox2
            mybox2.depth = 9;
            myboxl.volume ();//виклик методу для
    розрахунку 1 об’єму
            mybox2.volume ();//виклик методу для
    розрахунку 2 об’єму
        }
    }
```

Результат:

```
Об'єм становить 3000.0
Об'єм становить 162.0
```

Розглянемо детальніше дві останні стрічки коду. В першій стрічці відбувається виклик методу `volume()` для об’єкту `mybox1`, а в другій стрічці – для об’єкту `mybox2`. ***Особливість такого застосунку полягає в тому, що один і той самий метод, логіку***

якого викладено в класі, можна застосовувати для n кількості його екземплярів (об'єктів). За умови застосування подібного підходу нівелюється необхідність кожного разу при створенні нового об'єкту писати код для визначення його об'єму в `main()` – методі.

Враховуючи те, що метод `volume ()` використовує значення змінних екземпляра класу, до якого викликано метод, результат роботи для двох об'єктів отримано різний. Перевага такого застосунку очевидна: логіку методу, який написано один раз, можна використовувати багато разів без його переписування.

Після виклику методу `volume()` керуюча система Java передає керування коду визначеному в тілі методу `volume()`. Після закінчення виконання усіх операторів в тілі методу `volume()`, керування повертається до тієї частини програми, де реалізувався виклик методу, після чого її виконання продовжується з наступної стрічки.

В методі `volume()` слід зауважити ще одну особливість: посилання на змінні екземпляра `width, height, depth` реалізовані без прив'язки до конкретного об'єкту.

Існує й інший спосіб представлення методу `volume()`, з обов'язковим поверненням значення, проте без виводу у консоль (на той випадок, коли потрібно визначити об'єм без його виводу на екран):

```
double volume() {
    return width * height * depth;
}
```

З наведеного прикладу може виникнути питання: як дістатися до значення, що повертає метод `volume()`? Для цього необхідно ввести додаткову змінну за прикладом:

```
// …
double volume() {
    return width * height * depth;
}

double vol; // змінна, яка приймає значення, що
повертає метод
vol = mybox.volume ();
// …
```

Або цю процедуру можна виконати безпосередньо в самому методі `volume()`:

```
double volume() {
double vol;
vol = width * height * depth;
return vol;
}
```

Наведені приклади є автентичними!

При роботі з методами, що повертають значення, слід пам'ятати:

1. Тип даних, який повертає метод має бути сумісним з типами, що використовуються у методі.
2. Змінна, що приймає значення роботи методу, також має бути сумісна з типом методу.

Узагальнення: усі данні, що використовуються в методі, мають бути одного типу!

12.2. Методи, що приймають параметри

Нагадаємо приклад щодо форми написання методу:

```
тип імя_методу (список_парметрів) {
    // тіло методу
}
```

де `список_парметрів` означає послідовність пар «тип даних – назва», розділених комами. Параметри – це змінні, які приймають значення аргументів. Аргументи – це дані, що передаються методу під час його виклику. Іншими словами, метод приймає параметри для їх опрацювання під час його виклику.

Методи, які приймають під час виклику параметри та виконують над ними певні операції, називають ***параметризованими***. В якості прикладу розглянемо метод, що повертає квадрат числа 10 без прийняття параметрів:

```
int square ()  {
   return 10 * 10;
}
```

Недолік цього методу очевидний: він може визначати квадрат лише числа 10. А якщо потрібен метод піднесення до квадрату будь-якого числа? В такому разі слід написати метод із прийняттям відповідного параметру:

```
int square(int i) {
    return i * i;
}
```

Застосуємо даний метод для умовного екземпляра класу `two`:

```
…
two.square(2);
…
```

```
Результат роботи: 4.
```

Під час роботи з методами, які приймають параметри, слід розуміти два терміни: ***параметр*** та ***аргумент***.

Параметр – це визначена в методі змінна, яка приймає значення під час виклику методу. Наприклад, в методі `square()` параметом є `i`.

Аргумент – це значення, яке передається методу під час його виклику. В наведеному прикладі аргументом є число 2.

Параметри зазначаються в дужках при написанні методу, а аргументи – при його виклику!

Використовуючи методи з параметрами можна удосконалити розглянутий раніше клас `Box`. До цього моменту значення кожної змінної екземпляра класу необхідно було встановлювати індивідуально, використовуючи послідовність операторів:

```
myboxl.width = 10;
myboxl.height = 20;
myboxl.depth = 15;
```

Це працездатний код, проте через його громіздкий вигляд цілком реально допустити помилку, або не врахувати певного значення. Крім того, в правильно написаних програмах на Java доступ до змінних екземпляра має реалізовуватись лише через методи, визначені в їх класі. В подальшому поведінку методу можна буде перевизначити, проте неможливо буде змінити поведінку змінної екземпляра.

Відповідно, більш раціональним способом ініціалізації змінних (не беручи до уваги конструктори, розглянуті в попередній темі) є створення методу, який в якості параметрів приймає розміри та встановлює їх значення для кожної змінної екземпляра. Приклад:

```
// файл Box.java
class Box {
    double width;
    double height;
    double depth;

    double volume() {
        return width * height * depth;
    }

    void setBox(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
// файл BoxDemo.java
class BoxDemo {
    public static void main(String[] args) {

        Box myboxl = new Box() ;
        Box mybox2 = new Box() ;
        myboxl.setBox(10, 20, 15);
        mybox2.setBox(3, 6, 9) ;

        System.out.println(myboxl.volume ());
        System.out.println(mybox2.volume ());
    }
}
```

Як видно з наведеного прикладу, метод `setBox` використано для встановлення розмірів кожного параметру. Наприклад, при виконанні стрічки `myboxl.setBox(10, 20, 15);` аргумент 10 присвоюється в параметр `w`, 20 – `h`, 15 – `d`. Після чого в тілі методу `setBox` значення параметрів `w, h, d` присвоюються змінним `width, height` та `depth` відповідно.

Так, метод `setBox()` повністю виконує функціонал конструкторів класу, які ми вже розглядали. Проте в даному випадку такий приклад застосований для наочності процедури передачі параметрів для методу.

12.3. Використання об'єктів в якості параметрів методу

У попередніх прикладах в якості параметрів розглядалися лише примітивні типи даних. Проте передача методам об'єктів в якості параметрів є не лише доступною, але й доволі розповсюдженою практикою. Розглянемо приклад:

```
public class Test {
    int a, b;
 // створюємо конструктор
    Test (int i, int j){
        a = i;
        b = j;
    }
// передаємо об'єкт в якості параметру
    boolean equals (Test o){
        if (o.a == a && o.b == b) return true;
        else return false;
    }
}
public class PassOb {
    public static void main(String[] args) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " +
ob1.equals(ob2));
        System.out.println("ob1 == ob3: " +
ob1.equals(ob3));
    }
}
```

Результат роботи програми:

```
ob1 == ob2: true
ob1 == ob3: false
```

В наведеному прикладі метод `equals ()` перевіряє в класі `Test` на рівність два об’єкти та повертає логічний результат. Метод порівнює об’єкт, до якого його викликано, із об’єктом, переданим йому в якості аргументу. У випадку, якщо два об’єкти мають однакові значення, метод повертає логічне значення `true`, в іншому випадку – `false`.

Особливу увагу слід звернути на те, що в якості типу даних для параметра «о» в методі `equals ()` вказано клас `Test`. Ця процедура і відображає усю сутність передачі об’єктів в якості параметрів.

В якості параметрів об’єкти найчастіше зустрічаються в конструкторах. Іноді виникає необхідність створити новий екземпляр класу, ідентичний до вже існуючому екземпляру (створити клон). Для цього необхідно визначити конструктор, що приймає в якості параметра об’єкт (екземпляр) свого класу. Інакше кажучи, один об’єкт ініціалізується іншим об’єктом. Розглянемо приклад:

```
public class Box {
    double width;
    double height;
    double depth;

    // конструктор, в якому параметр - об'єкт типу
    Box
    Box(Box ob) {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор для ініціалізації усіх розмірів
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
```

```
    }
}
public class BoxDemo {
    public static void main(String[] args) {
        Box mybox1 = new Box(5, 10, 15); //
використання стандартного конструктора
        Box myclone = new Box(mybox1); //
створення клону
        System.out.println(mybox1.volume());
        System.out.println(myclone.volume());
    }
}
```

```
Результат роботи програми
750.0
750.0
```

Почавши розробляти власні класи, отримуємо розуміння того, що на «озброєнні» необхідно мати декілька форм конструкторів. Вони дозволяють зручно та ефективно створювати потрібні екземпляри класів. Це питання детальніше буде розглянуто в наступних темах (поняття поліморфізму).

Для передачі аргументів методу існує ДВА способи: виклик за значенням та виклик за посиланням. Коли методу передається аргумент примітивного типу, його передача відбувається за значенням. Як наслідок, створюється копія аргументу та всі зміни, що проводяться в подальшому із параметром, який приймає цей аргумент, не завдають жодного впливу за межами викликаного методу. В якості прикладу розглянемо наступну програму:

```
//аргументи примітивних типів передаються за
значенням
public class TestMethod {
    void meth (int i, int j){
        i *= 2;
        j /=2;
    }
}
```

```
public class TestMain {
    public static void main(String[]args) {
        TestMethod ob = new TestMethod();

        int a = 15, b = 20;
        System.out.println("a та b до виклику: " +
a + " " + b);

        ob.meth(a, b);
        System.out.println("a та b після виклику:
" + a + " " + b);
    }

}
```

```
Результат роботи програми:
a та b до виклику: 15 20
a та b після виклику: 15 20
```

Як видно з наведеного прикладу, операції, що прописані в методі `math()`, не мають жодного впливу на значення змінних a та b, які використовуються при виклику цього методу. Їх значення не змінилось на 30 та 10 відповідно.

При передачі об'єкта в якості аргументу для метода, ситуація змінюється кардинально, оскільки об'єкти, за своєю сутністю, передаються при виклику за посиланням. При оголошенні змінної типу класу (посилкового типу) створюється лише посилання на об'єкт цього класу. Таким чином, при передачі цього посилання методу, параметр, що приймає посилання, буде посилатись на той самий об'єкт, на який посилається аргумент. Об'єкти діють так, наче вони передаються методам за посиланням. Будь які зміни об'єкта в тілі методу мають вплив на об'єкт, який передано в якості аргументу. Для наочності розглянемо приклад:

```
public class TestMethod {
    int a, b;

    TestMethod(int i, int j) {
        a = i;
        b = j;
    }

    // передаємо об'єкт
```

```
    void meth(TestMethod o) {
        o.a *= 2;
        o.b /= 2;
    }
}
public class TestMain {
    public static void main(String []args ){
        TestMethod ob = new TestMethod(15, 20);

        System.out.println("a та b до виклику: " +
ob.a + " " + ob.b);

        ob.meth(ob);
        System.out.println("a та b після виклику:
" + ob.a + " " + ob.b);
    }
}
```

```
Результат роботи програми:
a та b до виклику: 15 20
a та b після виклику: 30 10
```

Як бачимо з наведено прикладу, дії, які виконуються в тілі методу `meth()`, мають вплив на об'єкт, використаний як аргумент.

Як відомо, типізовані методи можуть повертати значення будь-якого типу даних. Це також стосується і посилкових типів (типів певного класу). В наведеному далі прикладі метод `incrByTen ()` повертає об'єкт, в якому значення змінної збільшується на 10 у порівнянні з переданим аргументом.

```
public class TestMethod {
    int a;

    TestMethod(int i) {
        a = i;
    }

    TestMethod incrByTen() {
        TestMethod temp = new TestMethod(a + 10);
        return temp;
    }
}
```

```
public class TestMain {
    public static void main(String []args) {
        TestMethod ob1 = new TestMethod(2);
        TestMethod ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a після повторного 
збільшення: " + ob2.a);
    }
}
```

```
Результат роботи:
ob1.a: 2
ob2.a: 12
ob2.a після повторного збільшення: 22
```

Отже, при кожному виклику метода `incrByTen ()` створюється новий об'єкт, а посилання на нього повертається тій частині програми, де відбувається виклик. В наведеному прикладі можна побачити ще один цікавий момент. Пам'ять виділяється для всіх об'єктів динамічно з допомогою оператора new, відповідно, розробнику не потрібно вживати заходів, аби об'єкт не вийшов за межі області своєї роботи. Це можливо за рахунок того, що виконання методу, де створюється об'єкт, обов'язково закінчує свою роботу. Об'єкт буде існувати , доки буде існувати посилання на нього в будь-якому іншому місці програми. А за відсутності будь-яких посилань на об'єкт він буде знищений при наступному збиранні «сміття».

Висновок: основний функціонал (логіка) будь-якого додатку подається у методах, які, використовуючи значення змінних екземплярів класу, виконують основну роботу програми. Як відомо, клас є оболонкою для написання програми, а методи – безпосередні «виконавці» цієї програми. Саме тому знання особливостей написання методів, їх структури та порядку застосування є одними із ключових моментів для розв'язання прикладних завдань з програмування на практиці.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с.

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java.

4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java.

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java (наявне в віртуальному середовищі)

Запитання до лекції

1. Які методи називають типізованими?
2. Як написати метод у класі?
3. Що таке void метод?
4. Дайте визначення та наведіть приклади параметризованих методів.
5. Як передати об’єкт в якості параметру методу?

ЛЕКЦІЯ 13.
ПОЛІМОРФІЗМ

13.1. Знайомство з поліморфізмом
13.2. Перевизначення методів
13.3. Перевизначення конструкторів
13.4. Рекурсія

13.1. Знайомство з поліморфізмом

Поліморфізм (від грец. «багато форм») – один із принципів ООП, який дозволяє використовувати спільний інтерфейс для загального класу дій, де кожна дія залежить від конкретної ситуації. Для прикладу розглянемо стек (останній прийшов, перший пішов). Припустимо, що для реалізації програми необхідно реалізувати три стеки: для цілочисельних типів даних, даних із плаваючою крапкою та для символів. Алгоритм реалізації усіх стеків є аналогічним, незважаючи на різні типи даних. Якщо не використовувати принцип поліморфізму (створювати програму не об'єктно-орієнтованою мовою), то для створення стеків необхідно було б реалізовувати три окремі програми з різними іменами, але аналогічною логікою. А використовуючи поліморфізм, для роботи з трьома стеками можна створити одну підпрограму та двічі її перевизначити, при цьому назва методу (підпрограми) у всіх трьох випадках буде ідентичною.

В більш загальному розумінні принцип поліморфізму можна виразити означенням «один інтерфейс, багато методів). Це означає, що можливо розробити один інтерфейс для групи зв'язаних дій. Такий підхід дозволяє спростити програму, оскільки спільний інтерфейс слугує для загального класу дій. А вибір конкретної дії (тобто метода) здійснюється окремо для конкретної ситуації та входить в обов'язки компілятора. Залучення компілятора звільняє програміста від необхідності здійснювати такий вибір вручну, йому необхідно лише пам'ятати про загальний інтерфейс та правильно його застосовувати.

Для кращого уявлення розглянемо декілька прикладів. В першому прикладі задекларуємо, що собачий нюх – це поліморфна властивість. Якщо пес відчує запах кішки, то рефлекторно почне гавкати. А при запаху їжі – відбуватиметься виділення слини. Отже, процес один – сприйняття запаху. Але, залежно від умов, де він застосовується, реалізується інша логіка. Різницю в описаних

ситуаціях створює предмет, який видає запах. В мові програмування ця відмінність проявляється між параметрами та типами даних, які передаються методу для опрацювання.

Ще один приклад – автомобіль. Поліморфізм в цьому випадку полягає в здатності виробників автомобілів пропонувати кілька різновидів автомобілів під однією маркою. Так, автомобіль може бути обладнаним звичайною гальмівною системою, або системою ABS, кермом із гідропідсилювачем або із звичайною рейкою, 4-х або 6-тициліндровим двигуном тощо. Але в будь-якому випадку для гальмування необхідно натиснути на педаль гальм, обертати кермо аби здійснювати маневри та натискати на педаль акселератора, щоб збільшити швидкість руху. Однаковий інтерфейс може використовуватись для керування різноманітними реалізаціями.

Розглянувши загальні поняття поліморфізму, перейдемо до реальних інструментів його реалізації

13.2. Перевизначення методів

В Java дозволяється в одному класі визначати декілька методів із одним іменем, за умови якщо оголошення їх параметрів відрізняється. В такому випадку методи отримують назву перевизначених. Перевизначення методів є одним із способів підтримки поліморфізму в Java.

При виклику перевизначеного методу, для визначення потрібного його варіанту в Java використовують тип та (або) кількість аргументів методу. Відповідно, перевизначені методи мають відрізнятись за типом та кількістю параметрів.

Типи перевизначених методів можуть відрізнятись, проте одного лише типу явно не достатньо для того, щоб відрізняти два різні варіанти методу. Коли в середовищі Java зустрічається виклик перевизначеного методу, виконується той варіант, параметри якого відповідають аргументам, що були вказані під час виклику методу.

Прилад перевизначення методу:

```
class Overload {
    // основний метод
    void test() {
        System.out.println("Параметри відсутні");
    }
```

```
    // перевизначений метод, перевірка наявності 1 
числа int
    void test(int a) {
        System.out.println("a : " + a);
    }

    // перевизначений метод, перевірка наявності 2 
чисел int
    void test(int a, int b) {
        System.out.println("a та b : " + a + " та 
" + b);
    }

    // перевизначений метод, перевірка наявності 1 
числа double
    double test(double a) {
        return a;
    }
}

public class OverloadMain {

    public static void main(String[] args) {
        Overload ov = new Overload();

        ov.test();
        ov.test(5);
        ov.test(5, 10);
        System.out.println("a : " + ov.test(5.5));

    }
}
```

```
Результат роботи програми:
Параметри відсутні
a : 5
a та b : 5 та 10
a : 5.5
```

Як бачимо, метод `test` було перевизначено чотири рази. Перший варіант взагалі не приймає параметрів, другий приймає один цілочисельний параметр, третій – два, а четвертий варіант приймає одне дробове число. З наведеного прикладу наочно видно, що

компілятор Java самостійно обирає на виконання той метод, параметри якого відповідають заданим аргументам.

Під час виклику перевизначеного метода відбувається співставлення типів даних аргументів, які використовуються для виклику метода, та типів даних параметрів метода. Проте співпадіння типів даних не завжди є обов'язковим. Інколи важливу роль в перевизначені методів може відігравати автоматичне приведення типів. Для наочності розглянемо наступний приклад:

```
class Overload {
    // основний метод
    void test() {
        System.out.println("Параметри відсутні");
    }
    // перевизначений метод, перевірка наявності 2
чисел int
    void test(int a, int b) {
        System.out.println("a та b : " + a + " та
" + b);
    }
    // перевизначений метод, перевірка наявності 1
числа double
    double test(double a) {
        return a;
    }
}

public class OverloadMain {

    public static void main(String[] args) {
        Overload ov = new Overload();
        int i = 88;

        ov.test();
        ov.test(5, 10);

        // викликається варіант методу test
(double)
        System.out.println("a : " + ov.test(5.5));

        // і тут викликається варіант методу test
(double)
```

```
            System.out.println("a : " + ov.test(i));
        }
    }
```

```
Результат роботи програми:
Параметри відсутні
a та b : 5 та 10
a : 88.0
a : 5.5
```

Як видно з наведеного прикладу, перевизначений метод `test(int)` не визначено. Тому при виклику методу `test()` з цілочисельним аргументом в класі відсутній відповідний метод. Але Java здійснює автоматичне перетворення типу `int` в `double`, щоб дозволити здійснити виклик необхідного варіанту методу. Так, якщо варіант метода `test(int)` не буде визначено, тип змінної «i» автоматично приводиться до `double`, після чого викликається метод `test(double)`. Переконатись в цьому досить просто, ініціалізація змінної проведена числом 88, а в результаті роботи програми ми отримали дробове значення 88,0. Звичайно якщо б варіант методу `test(int)` існував, то в описаному випадку проводився б виклик саме цього методу. Автоматичне приведення типів в Java проводиться лише тоді, коли не знайдено варіанту повного співпадіння.

Перевизначення методів підтримує поліморфізм, оскільки це один із способів реалізації принципу ООП «один інтерфейс, декілька методів». Проведемо роз'яснення цього тлумачення детальніше. В мовах програмування, що не підтримують принципів ООП, кожному новому методу має присвоюватись унікальне ім'я. Але найчастіше виникає потреба реалізації одного і того ж методу для різних типів даних. Розглянемо в якості прикладу функцію, яка визначає модуль числа. В мовах програмування, де перевизначення методів не підтримується, існує три або більше варіантів цієї функції. До прикладу, в мові програмування C функція `abs()` повертає число по модулю із типом даних `integer`, функція `labs()` – значення типу `long integer`, функція `fabs()` – значення із плаваючою крапкою. В мові C перевизначення методів не підтримується, тому в кожної з вказаних функцій має бути унікальне ім'я, не дивлячись на те, що всі три функції виконують одну і ту задачу. На виході ситуація стає принципово складнішою, аніж могла бути. Хоча кожна функція побудована за одним принципом, програмісту на мові C необхідно

пам’ятати три різні назви одного і того ж методу. В об’єктно-орієнтованих мовах, які підтримують поліморфізм, така ситуація не виникає, оскільки усі методи визначення модуля числа для різних типів даних можуть називатись однаково. В мові програмування Java метод `abs()` входить до стандартної бібліотеки та доступний через клас Math. В залежності від типу аргументу, в Java викликається необхідний варіант методу.

Перевизначення методів цінне тим, що дозволяє звертатись до ідентичних методів за загальним іменем. Відповідно, метод `abs()` представляє лише загальну дію, яка має виконуватись, а вибір відповідного метода покладається на компілятор.

При перевизначенні методів кожен його варіант має виконувати поставлені перед ним завдання. Не існує правила, відповідно до якого перевантажені методи мають бути зв’язані один з одним. Хоча, із стилістичної точки зору, перевизначення методів передбачає їх певний зв'язок. Тому на практиці перевизначати слід тільки тісно пов’язані методи (за логікою та стилістикою).

Розглянемо ще один, не менш важливий приклад перевантаження методів різних класів, які виконують спільну логіку. Такий спосіб перевизначення дозволяє зменшити трудомісткість написання коду, позбавляючи необхідності переписувати методи n-ну кількість разів.

Припустимо, маємо клас `Pet`, в якому створено пустий метод `voice ()`:

```
public class Pet {
    public void voice ();
}
```

Створюємо два класи `Dog` та `Cat`, які є дочірніми від класу `Pet` та наслідують усі його методи (наслідування вивчатимемо дещо пізніше). Відповідно, у кожному дочірньому класі метод `voice()` виконуватиме свою унікальну логіку, хочаноситиме спільне ім’я:

```
public class Dog extends Pet {
    @Override
    public void voice() {
        System.out.println("Я пес - Гав-Гав");
    }
}
```

```
public class Cat extends Pet{

    @Override
    public void voice() {
        System.out.println("Я кіт - Няв-Няв");
    }
}
```

Цікавим є те, що для наочності перевизначення методу перед ним генерується позначення `@Override`.

13.3. Перевизначення конструкторів

Крім перевизначення звичайних методів, перевизначенню також піддаються і конструктори. На практиці перевизначення конструкторів вважається нормою. Для наочності розглянемо приклад:

```
public class Box {
    double width;
    double height;
    double depth;

    public Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}
```

В такому варіанті виконання програми під час створення екземпляру класу `Box` компілятор захоче відразу ініціалізувати усі три змінні екземпляра класу (задати аргументи для параметрів w, h, d).

```
Box myboxl = new Box (10,20,15);
```

Очевидно, що ініціалізація конструктора може здійснюватись лише так і ніяк інакше. При будь-якій іншій ініціалізації компілятор видаватиме помилку. А що робити, коли виникає необхідність не задавати жодного аргументу (пустий конструктор), або задати лише один аргумент, який автоматично буде ініціалізовано у всі три змінні

екземпляру? В такому випадку необхідно перевизначати конструктор:

```
public class Box {
    double width;
    double height;
    double depth;

    // конструктор для ініціалізації усіх розмірів
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, коли не задається жодного
розміру
    Box() {
        width = 0;
        height = 0;
        depth = 0;
    }

    // конструктор для створення куба (усі ребра
рівні)
    Box(double len) {
        width = height = depth = len;
    }

    double volume() {
        return width * height * depth;
    }
}

ublic class BoxDemo {

    public static void main(String[] args) {
        Box mybox1 = new Box(5, 10, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        System.out.println(mybox1.volume());
        System.out.println(mybox2.volume());
        System.out.println(mycube.volume());
    }
```

```
}
```

```
Результат роботи програми:
750.0
0.0
343.0
```

Отже, відповідний конструктор викликається залежно від аргументів, які вказано при створенні екземпляра класу.

13.4. Рекурсія

З точки зору знання рекурсії, класифікують три типи програмістів: ті, що не розуміють рекурсію, ті, що її розуміють, та ті, що її використовують! Відтак, розглянемо поняття рекурсії з метою її всебічного використання в процесі створення власних додатків.

Рекурсія – це засіб, який дозволяє методу викликати самого себе. Відповідно, метод побудований за таким принципом, називають *рекурсивним*.

Класичний приклад рекурсії – це визначення факторіалу числа n. Розглянемо даний приклад:

```
public class Recursion {
    int fact(int n) {
        if (n == 1) {
            return 1;
        } else {
            int res = fact(n - 1) * n;
            return res;
        }
    }
}

public class RecursionMain {
    public static void main(String[] args) {
        Recursion rec = new Recursion();
        System.out.println(rec.fact(4));
    }
}
```

```
Результат роботи програми:
24
```

З наведеного прикладу не цілком зрозуміло принцип роботи рекурсії. Тому опишемо цей процес дещо детальніше. Коли метод `fact()` в класі `Recursion` викликається із аргументом 1, то спрацьовує блок **`if`** та результат роботи програми повертає 1. В іншому випадку в роботу вступає виконання блоку **`else`**, де однією з перших процедур є повторний виклик методу `fact()` із значенням аргументу `n - 1` (в нашому випадку вже 3). Ця процедура повторюється до того моменту, поки значення n не дорівнюватиме 1, після чого відбувається зворотне повернення послідовно викликаних методів `fact()` із виконанням операції множення. Спробуємо розібрати означену процедуру за допомогою графічної моделі.

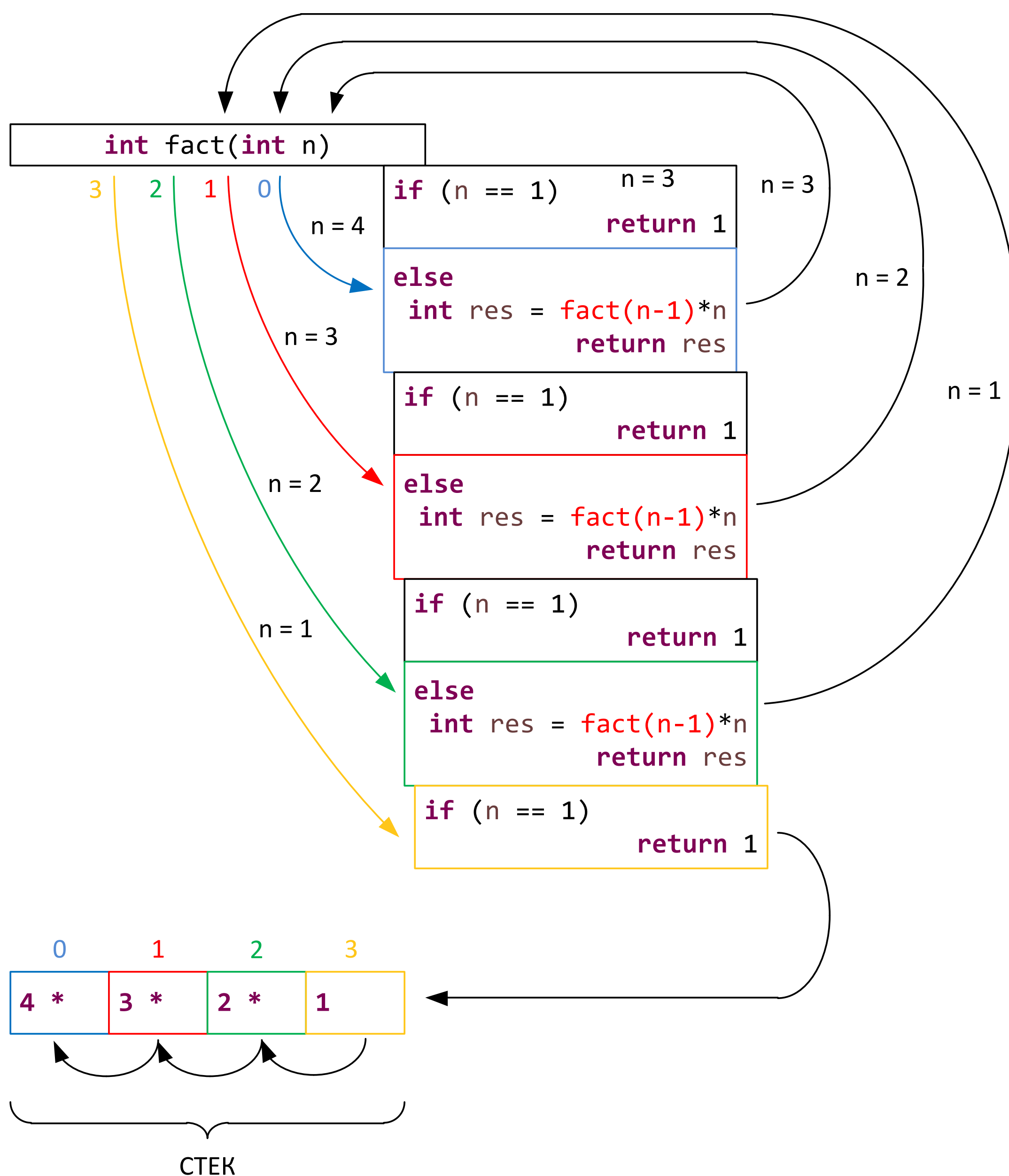

Рисунок 13.1 – Візуалізація рекурсивного методу визначення факторіалу числа n

Для розрахунку факторіала числа 4 після першого виклику методу `fact()` виконується другий виклик цього методу із значенням аргументу 3. Це, в свою чергу, призводить до третього виклику методу `fact()` із значенням аргументу 2, та четвертого виклику із значенням 1. Після досягнення n значення 1, метод повертає процедуру почергового добутку прийнятих аргументів між собою в зворотному напрямку, а саме 1 · 2 = **2**; 2 · 3 = **6**; 6 · 4 = **24**. Для того, щоб переконатись, що рекурсивна процедура визначення факторіалу числа n працює саме за наведеною моделлю, в метод `fact()` можна ввести процедуру `println()`, що виводить у консоль результат визначення факторіала на кожному рівні:

```
int fact(int n) {
    if (n == 1) {
        return 1;
    } else {
        int res = fact(n - 1) * n;
        System.out.println(res);
        return res;

    }
}
```

```
Результат роботи програми:
2
6
24
```

Може виникнути питання: чому обчислення добутку викликаних аргументів проводиться у зворотному порядку? Це пов’язано з тим, що аргументи після кожного рекурсивного виклику розміщуються в стеку (перший прийшов – останній вийшов). Рекурсивні методи виконують дії, які можливо порівняти з розкладанням та складанням телескопа.

За рахунок повторного виклику, рекурсивні методи можуть виконувати певні процедури значно повільніше, аніж їх ітераційні аналоги. Занадто велика кількість викликів рекурсивного методу може спричинити переповнення стеку, оскільки у ньому зберігаються параметри та локальні змінні, а при кожному новому виклику створюються нові копії цих значень. Застосування рекурсії в процесі написання методів має бути виваженим та обґрунтованим. До прикладу, алгоритм швидкого сортування, а також деякі алгоритми пов’язані з штучним інтелектом, дуже важко реалізувати ітераційним способом.

При написанні рекурсивних методів слід передбачити, щоб у певному місці методу був присутній умовний оператор **if**, який забезпечуватиме повернення з методу без його рекурсивного виклику. В протилежному випадку повернення із рекурсивного методу так і не відбудеться, що призведе до переповнення стеку. Це розповсюджена помилка, тому на стадії розробки рекурсивного методу рекомендується частіше здійснювати виклик методу `println()` з метою відслідковування роботи методу та його припинення у разі помилки.

Розглянемо ще один приклад організації рекурсії. В цьому прикладі рекурсивний метод `printArray()` виводить перші n елементів з масиву `values`.

```
class RecTest {
    int values [];

    RecTest (int n){
        values = new int [n];
    }

    // метод наповнення масиву
    void addValues(int n){
        for (n = 0; n < 10; n++){
            values [n] = n;
            System.out.print(values [n] + " ");
        } System.out.println();
    }

    // метод рекурсивного виведення елементів
масиву
    void printArray (int n){
        if (n == 0){
            return;
        }else{
            printArray (n-1);
            System.out.println("[" + (n-1) +
"]"+values[n-1]);
        }
    }
}
```

```
class RecTestMain {
    public static void main(String[] args) {
        RecTest rt = new RecTest(10);
        rt.addValues(9);
        rt.printArray(5);
    }
}
```

```
Результат роботи програми:
0 1 2 3 4 5 6 7 8 9
[0]0
[1]1
[2]2
[3]3
[4]4
```

Висновок: Як відомо, найбільшої популярності за останні роки набувають саме об'єктно-орієнтовані мови програмування, які підтримують основні принципи цієї парадигми. Що стосується роботи з методами, то принципи поліморфізму є одними із найбільш вживаних та дозволяють передбачати різні сценарії виконання однієї процедури, не розмножуючи велику кількість методів з різними інтерфейсами. Не менш важливою при роботі з методами є рекурсія. Розуміння цих положень надасть розробнику можливість створювати прості за архітектурою та високоефективні програмні коди.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с.

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java.

4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java.

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java.

Запитання до лекції

1. Розкрийте суть принципу поліморфізму.
2. Що таке перевизначення методу? Які бувають типи перевизначених методів?
3. Як здійснюється перевизначення конструктора?
4. Що таке рекурсія. Наведіть приклад організації рекурсії.

ЛЕКЦІЯ 14.
ІЄРАРХІЯ КЛАСІВ. НАСЛІДУВАННЯ

14.1. Знайомтво з наслідуванням
14.2. Вкладені та внутрішні класи
14.3, Наслідування

14.1. Знайомство з наслідуванням

Процес, в результаті якого один об'єкт (екземпляр класу) отримує властивості іншого, називається наслідуванням. Це дуже важливий принцип об’єктно-орієнтованого програмування, оскільки успадкування забезпечує принцип ієрархічної класифікації. Наприклад, ноутбук – частина класифікації персональних комп’ютерів, які, своєю чергою, є частиною класифікації електронних обчислювальних машин. Без ієрархії кожен об’єкт мав би визначати всі свої характеристики. Проте, завдяки наслідуванню, об’єкт може визначати лише ті характеристики, які роблять його унікальним. Ноутбук може наслідувати загальні атрибути від свого батьківського класу – ПК, проте унікальність його полягає в наявності батареї, тач-паду, можливості мобільного використання тощо. Решта основних компонентів, таких як пам'ять, материнська плата, клавіатура не є унікальним та можуть бути унаслідувані від батьківського класу, а відтак їх чергове перевизначення при створенні об’єкту «ноутбук», який є дочірнім класом «персонального комп’ютера», не має сенсу.

Наслідування зв’язане також з уже відомою нам інкапсуляцією. Якщо окремий клас інкапсулює окремі властивості, то будь-який його підклас (дочірній клас) буде мати ті ж властивості, а також будь-які інші додаткові, визначені безпосередньо в дочірньому класі. Новий підклас наслідує атрибути всіх своїх батьківських класів, а тому не містить непередбачуваних взаємодій з рештою коду системи.

14.2. Вкладені та внутрішні класи

В мові програмування Java дозволено визначати один клас в межах іншого. Такі класи мають назву **вкладених**. Область дії вкладених класів обмежується областю дії зовнішнього класу. До прикладу, якщо клас B визначений в класі A, то клас B не може існувати незалежно від класу A. Вкладений клас має доступ до членів

того класу, в який його вкладено. Проте зовнішній клас не має доступу до членів вкладеного класу.

Існує два типи вкладених класів: ***статичні*** та ***нестатичні***. Статичним називається вкладений клас, який оголошено з модифікатором **`static`**. Оскільки такі класи вважатимуть статичними, то звернення до нестатичних членів свого зовнішнього класу має забезпечуватись через екземпляр класу (об'єкт). Це означає, що вкладений статичний клас не може посилатись безпосередньо на нестатичні члени свого зовнішнього класу. Зважаючи на можливу плутанину та складність такої реалізації, статичні вкладені класи застосовуються нечасто.

Більш важливим типом вкладеного класу є **внутрішній** клас. Внутрішній клас – це нестатичний вкладений клас. Такі класи мають доступ до всіх змінних та методів свого зовнішнього класу.

Для наочності розглянемо нижче приклад використання внутрішнього класу. В класі `Outer` оголошено одну змінну екземпляра класу `outer_x` та визначено один метод `test()`, а також оголошено внутрішній клас `Inner`.

```
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

//оголошуємо внутрішній клас
    class Inner {
        void display() {
            System.out.println(outer_x);
        }
    }
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Результат роботи програми:
100

В програмі з наведеного прикладу внутрішній клас Inner оголошено (визначено) в області дії класу Outer. Зважаючи на це, будь яка частина коду класу Inner може звертатись до змінної outer_x. Метод внутрішнього класу display() виводить у консоль значення змінної outer_x. В методі main() з класу InnerClassDemo створюється екземпляр класу Outer та викликається метод test(). А безпосередньо в цьому методі створюється екземпляр класу Inner, який викликає метод display().

Слід зважати на те, що екземпляр класу Inner може бути створений тільки всередині класу Outer. В протилежному випадку компілятор Java видасть помилку.

Як вже було згадано, внутрішній клас має доступ до усіх членів зовнішнього класу, проте НЕ НАВПАКИ. Члени внутрішнього класу доступні лише в області дії цього класу та не можуть бути використані зовнішнім класом. Розглянемо приклад:

```
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    class Inner {
        int y = 10; //локальна змінна класу Inner
        void display() {
            System.out.println(outer_x);
        }
    }

    void showy () {
        System.out.println(y); // помилка
    }
}
```

З наведеного прикладу видно, що змінна y оголошена як змінна екземпляра класу Inner, тому вона недоступна за межами цього класу та не може використовуватись в методі showy().

У попередніх прикладах розглядався випадок, коли внутрішній клас оголошено як член в області дії зовнішнього класу. Проте внутрішні класи можна оголошувати і в області дії будь якого блока коду. Наприклад, внутрішній клас можна оголосити в блоці коду певного методу, або навіть в тілі циклу for:

```
class Outer {
    int outer_x = 100;

    void test() {
        for (int i = 0; i < 5; i++) {
            class Inner {
                void display() {
                    System.out.println(outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Результат роботи програми:

```
100
100
100
100
100
```

На завершення додамо, що спочатку розробники Java (версія 1.0) не допускали створення вкладених класів, така можливість з'явилася лише з появою версії Java 1.1.

14.3. Наслідування

Ключове місце у створенні ієрархічної структури класів займає наслідування – це один з основних принципів об’єктно-орієнтованого програмування.

Використовуючи наслідування, можна створити клас, який може унаслідуватись іншими, більш спеціалізованими класами. Кожен з них буде додавати свої особливі характеристики. В термінології об’єктно-орієнтованого програмування клас, від якого унаслідують називають ***суперкласо***м, або батьківським класом, а унаслідуваний клас – ***підкласом***, або дочірнім класом. Відповідно до зазначеного твердження, підклас – це спеціалізована версія суперкласу. Підклас наслідує усі члени, що оголошені в суперкласі, додаючи до них власні елементи.

Для того, щоб наслідувати клас, достатньо ввести назву батьківського класу після імені дочірнього класу, використавши ключове слово **extends**. Для наочності розглянемо короткий приклад, де підклас B наслідує суперклас A.

```
class A {
    int i, j;

    void showij () {
        System.out.println(i+" "+j);
    }
}

class B extends A {
    int k;

    void showk() {
        System.out.println(k);
    }

    void sum() {
        System.out.println("Сума: " + (i+j+k));
    }
}

class SimpleInheritance {

    public static void main(String[] args) {
        A superOb = new A();
```

```
            B subOb = new B();

            superOb.i = 10;
            superOb.j = 20;
            superOb.showij();

            subOb.i = 7;
            subOb.j = 8;
            subOb.k = 9;
            subOb.showij();
            subOb.showk();
            subOb.sum();
        }
    }
```

Результат роботи програми:

```
10 20
7 8
9
Сума: 24
```

З наведеного прикладу видно, що екземпляр суперкласу A має доступ до змінних та методів свого класу та може встановлювати їм значення і викликати відповідні методи. Що стосується екземпляра дочірнього класу B, то йому, окрім власної змінної `k`, також доступні змінні батьківського класу `i` та `j`. Крім того, екземпляр дочірнього класу може викликати не лише методи свого класу `showk()` та `sum()`, але й метод батьківського класу `showij()`. Крім того, в методі `sum()`, який розміщено в класі B, можливе безпосереднє посилання на змінні `i` та `j`, оголошені в класі A. Таким чином, можна узагальнити, що для будь-якого дочірнього класу доступні усі відкриті (крім **`private`**) змінні та методи батьківського класу, від якого він унаслідуваний. В цьому можна переконатись, проаналізувавши результат роботи програми з наведеного прикладу.

Те, що один клас може виступати суперкласом для іншого, не виключає можливості його самостійного використання без прив’язки до підкласу. Крім того, підклас також може виступати суперкласом для іншого класу, забезпечуючи при цьому трирівневу ієрархічну структуру наслідування, до прикладу:

```
class A {
```

```
}
class B extends A {
}
class C extends B {
}
```

Для кожного нового підкласу потрібно вказувати лише один суперклас. В Java не підтримується наслідування декількох суперкласів в одному підкласі. Як представлено в попередньому прикладі, можна лише створити ієрархію наслідування. В такому разі останньому в ієрархії класу C будуть доступні усі відкриті змінні та методи як класу B, так і класу A. Для наочності дещо видозмінимо клас `SimpleInheritance`, в якому для екземпляра класу C буде викликано усі змінні та методи класів B і A, а також розглянемо результат їх роботи:

```
class SimpleInheritance {
    public static void main(String[] args) {
        C subC = new C();
        subC.i = 1;
        subC.j = 2;
        subC.k = 3;
        subC.showij();
        subC.showk();
        subC.sum();
    }
}
```

Результат роботи програми:

```
1 2
3
Сума: 6
```

Не дивлячись на те, що підклас включає в себе усі члени суперкласу, він не може мати доступ до закритих членів (оголошених як **`private`**). Для прикладу, розглянемо наступну ієрархію класів:

```
1 class A {
2   int i;
```

```
3   private int j;
4
5   void setij(int x, int y) {
6       i = x;
7       j = y;
8   }
9 }
```

```
1 class B extends A {
2   int total;
3
4   void sum() {
5       total = i + j; // помилка
6   }
7 }
```

```
1 class SimpleInheritance {
2
3   public static void main(String[] args) {
4       B subOb = new B();
5       subOb.setij(10, 12);
6       subOb.sum();
7       System.out.println(subOb.total);
8   }
9 }
```

Запустити цю програму не вдасться, тому що використання закритої змінної `j` з класу A в методі `sum()`, який розміщено в класі B, призводить до порушення правил доступу (інкапсуляції). Оскільки змінна `j` оголошена в класі A як **`private`,** вона доступна лише іншим членам класу A. Підкласи можуть мати до неї доступ лише через спеціально написані методи.

Розглянемо більш практичний приклад, який допоможе краще продемонструвати можливості наслідування.

```
class Box {
    double width;
    double height;
    double depth;

    Box (Box ob){
```

```
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    Box (double w, double h, double d){
        width = w;
        height = h;
        depth = d;
    }
    Box(){
        width = -1;
        height = -1;
        depth = -1;
    }
    Box (double len){
        width = height = depth = len;
    }

    double volume () {
        return width * height * depth;
    }
}

public class BoxWeight extends Box{
    double weight;

    BoxWeight(double w, double h, double d, double
m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

public class BoxWeightDemo {

    public static void main(String[] args) {
        BoxWeight mybox = new BoxWeight(10, 20,
15, 34.3);
```

```
        System.out.println("Обєм складає:
"+mybox.volume());
        System.out.println("Вага складає:
"+mybox.weight);
    }
}
```

Результат роботи програми:

```
Обєм складає: 3000.0
Вага складає: 34.3
```

З наведеного прикладу стає зрозуміло, що клас BoxWeight, унаслідуваний від класу Box, містить лише одну змінну екземпляра класу **double** weight. Проте, незважаючи на це, конструктор класу дозволяє провести ініціалізацію змінних width, height, depth під час створення екземпляра класу в main()- методі. Це говорить про те, що клас BoxWeight унаслідував ці змінні від свого суперкласу. Крім того, для екземпляра класу BoxWeight також доступний метод volume(), який визначено в класі Box, а це , власне, і є **наслідування**. Узагальнюючи, можна констатувати, що клас BoxWeight наслідує усі характеристики класу Box, доповнюючи їх компонентом weight.

Головна перевага наслідування полягає в тому, що суперклас, який визначає загальні характеристики, можливо буде використовувати для розроблення низки більш спеціалізованих підкласів.

Йдемо далі. Розглянемо складніший випадок. Об'єкту (екземпляру) суперкласу може бути присвоєне посилання на об'єкт (екземпляр) будь якого його підкласу. Опираючись на попередній приклад, розглянемо зазначений випадок:

```
public class BoxWeightDemo {

    public static void main(String[] args) {
        BoxWeight weightbox = new BoxWeight(10,
20, 15, 34.3);
        Box plainbox = new Box(1, 2, 3);
        double vol;
        vol = weightbox.volume();
        System.out.println("Обєм складає: "+vol);
        System.out.println("Вага складає:
"+weightbox.weight);

        plainbox = weightbox;
```

```
        vol = plainbox.volume();
        System.out.println("Обєм складає: "+vol);
        // в наступній стрічці помилка, оскільки
об’єкт plainbox не визначає змінну weight (в класі
BoxWeight такої змінної нема)
        System.out.println("Вага складає:
    "+plainbox.weight);
    }
}
```

Результат роботи програми, якщо прокоментувати останню стрічку:

```
Об’єм складає: 3000.0
Вага складає: 34.3
Об’єм складає: 3000.0
```

На початку наведеного прикладу об’єкт weightbox містить посилання на члени класу BoxWeigh, а об’єкт plainbox – на члени класу Box. Але оскільки клас Box є суперкласом для BoxWeigh, то його екземпляру plainbox можна присвоїти посилання на об’єкт weightbox. Це означає, що якщо екземпляру суперкласу присвоїти посилання на екземпляр підкласу, то забезпечується його доступ до елементів (значень) підкласу, проте лише тих, які визначені в суперкласі. Саме тому у об’єкта plainbox немає доступу до змінної weight. І це пояснюється тим, що суперкласу не відомо, що саме міститься в його дочірньому підкласі. Екземпляр класу Box не має доступу до змінної weight, так як ця змінна не оголошена в ньому. Саме тому остання стрічка коду в наведеному прикладі буде видавати помилку.

Ще один цікавий факт з наведеного прикладу. При створенні екземпляра класу Box з іменем plainbox в конструкторі ініціалізовані три змінні. Проте значення цих змінних не приймались до уваги під час виконання програмою методу volume(), що наочно видно з результатів роботи програми. А все через те, що перед викликом методу volume() до об’єкта plainbox, самому об’єкту присвоєно посилання на інший об’єкт – weightbox. Саме тому розрахунок об’єму «коробки» для обох екземплярів класів буде рівним.

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с.

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с.

3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java

4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java

Запитання до лекції

1. Що таке наслідування? Наведіть приклади наслідування.
2. Дайте визначення поняттям підклас та суперклас.
3. Які клаис називають вкладеними. Перелічіть та опишіть тппи вкладених класів.

Навчальне видання

ПРИДАТКО Олександр Володимирович
ХЛЕВНОЙ Олександр Вікторович
БУРАК Назарій Євгенович

ОСНОВИ ПРОГРАМУВАННЯ (МОВОЮ JAVA)

Курс лекцій

Літературний редактор – Галина Падик
Комп'ютерна верстка – Олександр Хлевной
Друк на різографі – Маріанна Климус

Підписано до друку 16.11.2019 р.
Формат 60×84/16. Гарнітура Times New Roman.
Друк на різографі. Папір офсетний. Наклад: 100.
Ум. друк. арк. 11,5.

Друк ЛДУ БЖД
79007, Україна, м. Львів, вул. Клепарівська, 35
тел./факс: (032) 233-32-40, 233-24-79
ubgd@i.ua