Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики

М.Ф. Копитко, К.С. Іванків

Основи програмування мовою Java

Тексти лекцій

Серія: тексти лекцій №1/02

Львів

Видавничий центр ЛНУ імені Івана Франка

2002

Копитко М.Ф., Іванків К.С. Основи програмування мовою Java: Тексти лекцій. − Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002.− 83 с.

У текстах лекцій подано загальний погляд на мову 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 студентів, аспірантів та співробітників університету, програмістів персональних комп´ютерів, які бажають перейти на нові технології програмування та створювати сучасні програмні продукти.

Рекомендовано до друку науково-методичною радою факультету прикладної математики та інформатики

Протокол №_1__ від _15.02.2002р.

Рецензент: Завідувач кафедри інформаційних систем Львівського національного університету імені Івана Франка, професор, докт. фіз.-мат. наук Г.А.Шинкаренко

Відповідальний за випуск - професор Савула Я.Г.

© Копитко М.Ф., Іванків К.С.

ЗМІСТ

І. ЗНАЙОМСТВО З МОВОЮ *JAVA*

Що таке *Java*? Створюючи програми на більшості мов програмування, треба визначити, в якій операційній системі і на якому процесорі вони працюватимуть. Тільки визначивши це, можна долучити до програми виклик функцій з бібліотеки, призначеної для відповідної операційної системи. Наприклад: якщо розробляють програму для *Windows*, то можна використати бібліотеку *Microsoft Foundation Classes*; для роботи на платформі *Macintosh* − функції з *Mac OS Toolbox*. Після компіляції вихідних текстів отримуємо код, готовий до виконання на певному процесорі. Система *Windows* переважно працює на базі процесорів фірми *Intel*, комп'ютери *Macintosh* використовують процесори *Motorola 680x0* або *PowerPC*. Створюючи програми на *Java*, можна не замислюватися над тим, в якій операційній системі вони працюватимуть. *Java* має власний набір машинно-незалежних бібліотек, які називають пакетами. Щодо процесорів ситуація аналогічна. Компілятор *Java* не генерує безпосередньо інструкції процесору. Він створює проміжний код − байткод для віртуальної машини *Java* (*Java Virtual Machine* − *JVM*). Виходячи з того, що ядро віртуальної машини *Java* реалізовано практично для всіх типів комп'ютерів, вважатимемо файли байткодів незалежними від платформи.

1.1. Створення та еволюція

Започатковано мову *Java* у проекті фірми *Sun Microsystems* під назвою *Green* (1991). Головними розробниками першої робочої версії були Джеймс Гослінг (James Gosling), Патрік Ноутон (Patric Naughton), Кріс Ворс (Chris Warth), Ед Френк (Ed Frank) і Майкл Шерідан (Mike Sheridan). До 1995 року мову називали *Оак*, однак не пройшовши перевірку на допустимість торгової марки, було перейменовано на *Java*. Ідеєю створення нової мови програмування були не потреби *Internet*, а необхідність створення програмного забезпечення, яке не залежить від платформи (тобто архітектури) і використовується в побутових електронних приладах. Під час відпрацювання деталей *Java* виник вагомий чинник, який відіграв важливу роль щодо цієї мови. Таким чинником була всесвітня

інформаційна служба *World Wide Web* (*WWW*), розквіт якої припадає на 1994−1995 р.

Ідеї створення ефективних і незалежних (працюючих на різноманітних процесорах під керівництвом різних операційних систем) програм такі ж давні, як і саме програмування, і завжди витіснялися нагальнішими проблемами. Якщо зважити на те, що більшість програмістів належить до одного з трьох кланів (*Intel*, *Macintosh*, *UNIX*), які безупинно змагаються між собою, то зрозумілим стає те, що нагальної потреби в переміщенні коду довший час не виникало. Проте з виникненням *Internet* і *WWW* проблема переміщення програм стала втричі гострішою. Різко змінилися акценти: від створення коду для вбудованих контролерів побутової техніки до програмування для *Internet*. Це і спричинило великий успіх мови *Java*.

1.2. Від *C/C++* до *Java*

Після виникнення в 1979 р. мови *C++*, до якої було додано порівняно з мовою *C* об'єктно-орієнтовані засоби і збережено усі сильні сторони *C,* складається враження, що програмісти знайшли нарешті досконалу мову. Проте навіть могутня і популярна мова програмування має свої недоліки. Це важкі щодо розуміння і використання аспекти *C++*, пов'язані з керуванням пам'яттю і покажчиками. У *C++* також легко зберігається процедурний стиль програмування. Згідно з думкою П.Нортона, наступним кроком у логічному розвитку мов програмування є *Java*. Ця мова опирається на найпопулярнішу мову *C++* (розробники *Java* зробили це свідомо), проте цілковито відкидає поняття процедурного програмування і змушує підкорятися принципам об'єктно-орієнтованого програмування (ООП). У мові *Java* відсутні покажчики, а керування пам'яттю відбувається автоматично.

Зважаючи на подібність мов *Java* та *C++*, дехто вважає, що *Java* − це "*Internet*-версія *C++*". Насправді це не так. Незважаючи на значний вплив *C++* на створення *Java*, щодо них не існує ніякої сумісності (ні зверху вниз, ні знизу вверх). Крім того, цю мову не розроблено з метою заміни *C++*. Мову *C++* застосовують для вирішення одних проблем, мову *Java* − для інших. Найважливіша з них − створення програм, які не залежать від платформ виконання.

1.3. Девізи *Java*

Розробники *Java* створювали мову з метою втілення таких базових принципів:

- ◊ простота;
- ◊ безпека;
- ◊ перенесення;
- ◊ об'єктно-орієнтована направленість;
- ◊ стійкість щодо помилок;
- ◊ багатопоточність;
- ◊ незалежність від архітектури;
- ◊ інтерпретація;
- ◊ висока продуктивність;
- ◊ розподіленість;
- ◊ динамічність.

Простота, інтерпретація, перенесення, незалежність від архітектури. До простих мов програмування належать ті, які працюють з інтерпретатором (наприклад, *Бейсік*). Перші персональні комп'ютери поставлялися з інтерпретатором *Бейсіка*. Сьогодні їхнє місце займають *HTML* і мови сценаріїв для *Web*. Вивчати і використовувати мови програмування, які компілюються, набагато складніше, ніж ті, які інтерпретуються. Тобто є мови програмування для професіоналів. Наприклад, *C++*, де використання покажчиків і керування пам'яттю є складними не тільки для початківців, але й для досвідчених програмістів. Одна стрічка програми, яка звертається до недозволеного місця в пам'яті, може спричинити до збоїв не тільки програми, але й комп'ютера загалом.

Java − це мова, програми якою компілюються та інтерпретуються, і водночас вона має просту структуру мови високого рівня. Написана програма компілюється в проміжну форму − *байткод*. Пізніше ця програма виконується, тобто інтерпретується виконавчим середовищем *Java*. Байткод дуже відрізняється від машинного коду, який є послідовністю нулів та одиничок. Байткод − це набір інструкцій, які подібні до команд *Асемблера*. Машинний код комп'ютер виконує безпосередньо, а байткод потрібно інтерпретувати. Тому машинний код можна використати тільки на конкретній платформі, для якої його

скомпільовано. Байткод можна виконувати на довільній платформі, на якій є виконавче середовище *Java*. Саме ця можливість і робить програми на *Java* незалежними від архітектури. Так як байткоди є проміжною формою програми, то його інтерпретація вимагає незначних витрат.

Байткод створено для машини, яка реально не існує. Цю машину називають *віртуальною Java-машиною* (*JVM*), вона існує тільки в пам'яті комп'ютера. Створення компілятором *Java* байткоду для неіснуючої машини − це тільки половина процесу, який забезпечує незалежність від архітектури. Другу частину виконує інтерпретатор *Java*, який виконує роль посередника між віртуальною *Java*-машиною та реальним комп'ютером.

Архітектура мови для розподіленого мережевого середовища. Головною вимогою щодо мови для роботи в розподіленому просторі комп'ютерів (наприклад, в *Internet*) − це можливість працювати на різнорідних і розподілених платформах.

Мова *Java* є пристосованою до перенесення завдяки підтримці стандартів IEEE для структур даних, наприклад, цілих чисел, чисел з плаваючою комою і рядків.

До мови *Java* зачислено безпосередньо підтримку таких розповсюджених протоколів як *FTP*, *HTTP*, що забезпечує сумісність під час роботи в мережі.

Java забезпечує розподілену роботу за допомогою механізму виклику віддалених методів (*RMI*), тобто дає змогу використовувати об'єкти, розташовані на локальних і віддалених машинах.

Багатопоточність. У багатопоточних операційних системах для кожного застосування (процесу) надається окрема захищена область пам'яті, в якій зберігаються коди програми і дані. А час одного процесора квантується між цими процесами. З метою запуску процесу або переключення з одного на інший на рівні операційної системи необхідно виконати значний об'єм роботи. Тому для розробників прикладних програм спеціально створили "полегшену" версію системного процесу − потік. Найбільшою проблемою, пов'язаною з процесами і потоками, є їхнє функціонування під керівництвом конкретної операційної системи. Спеціалісти компанії *Sun* зробили потоки частиною мови програ-

мування. Тому багатопоточне застосування, написане мовою *Java*, працюватиме і в операційних середовищах *Windows*, *Unix*, *MacOs*.

Висока продуктивність. Інтерпретатор *Java* може виконувати байткоди зі швидкістю, яка наближається до швидкості виконання коду, відкомпільованого до машинного формату, що досягається завдяки використовування інтерпретатором багатьох потоків виконання. Наприклад, доки комп'ютер чекає на введення даних, фонові потоки можуть зайнятися очищенням пам'яті.

Стійкість до помилок. *Java* − це мова строгого використання типів, що зумовлює зменшення числа помилок під час написання програми.

У мові Java відбувається автоматична перевірка виконання граничних умов під час роботи з масивами і стрічками, які в *Java* є класами.

У *Java* немає арифметики покажчиків, а керування пам'яттю здійснюється автоматично. Програмний код, написаний мовою *Java* не може зіслатися на пам'ять поза простором програми або зробити помилку внаслідок вивільнення пам'яті і тим самим вичерпати всю пам'ять.

Зазначимо, що у *Java* організовано процес автоматичного збирання сміття, тобто об'єктів, на які більше ніхто не вказує.

Безпека. Функції забезпечення безпеки дуже важливі для розподілених мереж з безліччю вірусів, "троянських коней" і т. п. Для реалізації цієї мети розробники мови *Java* створили механізм, який отримав назву пісочниці (*sandbox*):

- перевірку на рівні *JVM*;
- захист на рівні мови;
- інтерфейс *Java Sucurity* (цифрового підпису).

1.4. Структура мови Java

Аплети і застосування. Якщо б мову *Java* використовували для створення машинно незалежних застосувань, то і цього було б достатньо для її успіху в програмістів. Однак у 1993 р. компанія *Sun* звернула увагу на зростання популярності *Internet* і почала доробляти мову *Java* так, щоб написані на ній програми можна було запускати з *Web*-бровзерів. Відтоді самостійні *Java*-програми називають *застосуваннями* (applications), а програми, які виконуються під керівництвом інших програм (переважно *Web*-

бровзерів), − *аплетами* (applets). Аплети необхідні у випадку, коли для створення потрібної *Web*-сторінки не вистачає можливостей HTML, мови сценаріїв, а також у випадку, коли необхідно забезпечити зворотній зв'язок з клієнтом.

Простір імен. Під час написання складних програм інколи важко забезпечити унікальність імен змінних і класів. У мові *Java* використовують систему декількох рівнів вкладеності імен:

0 - простір імен пакета;
1 - простір імен одиниці компіляції (файл класу);
2 - простір імен типу (клас у класі);
3 - простір імен методу;
4 - простір імен локального блоку;
5 - простір імен вкладеного локального блоку.

За підтримку і перетворення просторів імен відповідає компілятор *Java*. Імена, пов'язані з кожним рівнем, відокремлюють від імен інших рівнів крапкою. Наприклад, Java.awt.BorderLayout.

Файли програми. Файл з вихідним текстом програми мовою *Java* є звичайним текстовим файлом з розширенням .java. Після компіляції (за допомогою javac) вихідних текстів отримується по одному файлу для кожного класу, оголошеного в тексті програми. Імена файлів збігаються з іменами класів (з урахуванням регістра) і мають розширення .class.

Пакети. У багатьох мовах програмування набір зв'язаних класів або функцій називають бібліотекою. *Java* надає поняттю бібліотеки певний відтінок, використовуючи для описання набору зв'язаних класів термін *пакет.* Наприклад, базові функції *Java* розташовані в пакеті java.lang.

Оператори імпорту. З метою використання класів з існуючих пакетів необхідно до тексту програми (першими) додати оператори

```
import java.awt.*;
```

Це означає, що всі класи пакета java.awt можна використовувати при написанні програмного коду безпосередньо без посилання на пакет.

Оголошення класів. Усі класи в мові *Java* є похідними від системного класу *Object*. У *Java* допускається тільки одинарне наслідування, тобто в ієрархії перед класом є тільки один базовий клас.

Оголошення інтерфейсу. Інтерфейсом у мові *Java* називають абстрактний клас. Його введено для реалізації наслідування від декількох класів.

1.5. Мова *Java* − це не вдосконалена HTML

Серед початківців розповсюджена хибна думка, що оскільки мову *Java* використовують для створення *Web*-сторінок, її можна вважати деяким вдосконаленням HTML (Hypertext Markup Language − мова розмітки гіпертексту). Насправді це не так. HTML є засобом логічної організації інформації і створення гіпертекстових зв'язків з відповідними даними. Вона дає змогу читати документи не тільки зверху вниз, а й у будь-якому іншому порядку, проте ніколи не була мовою програмування. Єдиний зв'язок між HTML і *Java* − це наявний в HTML дескриптор APPLET, за допомогою якого викликається для виконання аплет *Java*.

1.6. Середовище розробки програм мовою *Java*

Компанія *JavaSoft*, створена розробниками мови *Java*, пропонує безкоштовно набір засобів для програмістів мовою *Java JDK* (*Java Development Kit*) за адресою http:// java.sun.com/products/jdk/. *JDK* містить все необхідне для створення програм: базові функції мови, інтерфейс прикладного програмування (API) з наборами пакетів й основні інструменти. Більшість версій *JDK* містять сім інструментів розробки на *Java*:

1) компілятор (javac);
2) генератор документації (javadoc);
3) генератор файлів заголовків і заглушок мови *C++* для *Java* (javah);
4) інтерпретатор (java);
5) програму перегляду аплетів (appletviewer);
6) реасемблер файлів класів;
7) відлагоджувач програм (jdb).

Сьогодні версії *JDK* існують для більшості операційних систем. Починаючи з версії *JDK 1.2* прийнята нова домовленість щодо імен. Загальну технологію називають *Java 2*, а засоби розробника *Java 2 − SDK* (*Software Development Kit*).

Процес встановлення *SDK* налічує три етапи:

1) отримання SDK (переважно з *Internet*);
2) встановлення SDK;
3) перевірка конфігурації.

При перевірці конфігурації необхідно визначити правильне встановлення змінних оточення PATH, яка вказує на каталог з інструментами SDK, та CLASSPATH, яка задає шлях до каталогів класів *Java* (як готових, так і власних).

1.7. Комерційні інтегровані середовища розробки програм мовою *Java*

Професійні програмісти, яким оплачують за кожну хвилину роботи, повинні працювати дуже продуктивно. Для підвищення продуктивності їхньої праці використовують IDE (*Integrated Development Environment*) − візуальне середовище розробки. Кожна з таких відомих компаній як *Microsoft, Symantec*, *Borland*, *Sun* пропонує свої IDE. Перелік компаній, які мають засоби розробки *Java*, можна знайти на сервері Yahoo за адресою http:// www.yahoo.com/Business_and_Economy/Companies/Software/Programming_Tools/Laguages/Java/. При наявності великої кількості IDE відразу ж виникає запитання, яке середовище розробки найкраще? У 1997 році, згідно з опитуванням, яке здійснило об'єднання developer.com, складено такий список IDE:

1. Visual Cafe for Java від Symantec;
2. JBuilder від Borland;
3. VisualAge for Java від IBM;
4. CodeWarrior Professional від Metrowerks;
5. Visual J++ від Microsoft.

Інша компанія *Market Decisions, Inc.*, яка виконувала незалежні маркетингові дослідження, повідомила, що майже половина розробників *Java* використовують *Visual J++* від *Microsoft*. Детальну інформацію щодо продуктів *Java* компанії *Microsoft* можна знайти за адресою http:// www.microsoft.com/visualj/. Зауважимо також, що компанія *Sun Microsystems* подала судовий позов на *Microsoft* з приводу використання логотипу *Java* на тій підставі, що *Visual J++* не підтримує стандарт *Java*.

Конкурент компанії *Microsoft* на ринку продуктів з *Java* компанія *Symantec* своїм важливим козирем вважає те, що її Visual

Cafe цілковито відповідає стандартові *Java*. Компанія *Symantec* визначає *Visual J++* як дещо підправлене та адаптоване середовище *C++* від *Microsoft*.

Фірма *Borland* також намагається не відставати від конкурентів і пропонує своє середовище JBuilder. Детально познайомитися з JBuilder можна на Web-сервері за адресою http:// www.borland.com/jbuilder/.

1.8. Перша проста програма мовою *Java*

З метою створення програми на *Java* необхідно інсталювати JDK, використовуючи *Internet*-адресу:
http://java.sun.com/products/jdk/. Далі виконати такі дії:

1) написати текст програми;
2) відкомпілювати усі класи за допомогою компілятора javac;
3) виконати програму за допомогою інтерпретатора java.

1.8.1. Написання тексту програми

Якщо не використовувати інтегроване середовище розробки, то для написання тексту програм мовою *Java* можна використовувати звичайний текстовий редактор. Наприклад, в ОС *Windows* – це *NotePad, WordPad.*

Для більшості мов програмування ім'я файла, який містить вихідний текст програми, може бути довільним. Для мови *Java* це не так. У *Java* вихідний код офіційно називається модулем компіляції (compilation unit). Він є текстовим файлом, який містить одне або більше визначень класів. Компілятор вимагає, щоб вихідний файл мав розширення .java, а його ім'я збігалося з іменем класу (з урахуванням регістру), в якому є метод *main()*.

Компанія *Sun* пропонує дотримуватися декількох домовленостей з приводу імен під час написання *Java*-програм:

1. В іменах класів можна використовувати як великі, так і малі букви. Перша буква має бути великою. Наприклад, класам бажано надавати імена *NativeHello* і *HelloWorld*, а не *nativeHello* і *helloWorld*.
2. В іменах методів також можна використовувати символи обох регістрів, однак перша буква має бути малою.

Наприклад, методу можна присвоїти ім'я *sayHello()*, а не *SayHello()* або *sayhello()*.

3. Для найменування властивостей використовують ті ж домовленості, що і для методів. Наприклад, властивість можна назвати *thePoint*, а не *ThePoint* чи *thepoint*.
4. Імена констант переважно пишуть великими буквами. Наприклад, *PI*, а не *pi*.
5. Імена методів доступу до властивостей розпочинають з *set* і *get*.
6. Якщо властивість має тип *boolean*, то краще в ролі префікса в методі писати *is*, *has*.

Під час написання програм небажано використовувати відкриті властивості класу прямо, а тільки через методи доступу.

Аналіз коду програми. Розглянемо текст загальноприйнятої першої програми, яка виводить текстовий рядок на екран.

```
1    /* Проста програма мовою Java */
2    public class Example {
3       public static void main(String args [ ]){
4          System.out.println("Перша програма мовою Java");
5          }
6    }
```

Нумерацію рядків наведено для зручності пояснення тексту програми, а не через необхідність. Незважаючи на те, що програма дуже коротка, вона складається з декількох ключових особливостей, визначальних для всіх програм мовою *Java*. Вона розпочинається з коментарію: текст, обрамлений /*...*/, у першому рядку. Цей тип коментарю називають багаторядковим. У мові *Java* є однорядкові коментарі, які розпочинаються з //.

Звернемо увагу на те, що в мові *Java* усі змінні і методи (у тім числі *main*) не можуть бути за межами класу. Тому другий рядок є оголошенням класу *Example.*

Усі *Java*-програми (крім аплетів) розпочинають свою роботу з виклику методу *main()* (рядок 3). Метод *main ()* необхідно оголосити як *public*, тому що його викликають кодом, визначеним за межами класу. Ключове слово *static* дає змогу викликати метод *main()* без обов'язкового створення екземпляра класу (об'єкта типу *Example)*. Це вимушений крок, оскільки *main()* викликає інтер-

претатор *java* до створення будь-яких об'єктів. Ключове слово *void* повідомляє, що метод не повертає ніякого значення.

Необхідно пам'ятати, що в програмах мовою *Java* розрізняється регістр букв. Наприклад, метод *Main* буде відкомпільовано, проте інтерпретатор *java* його не знайде.

Складні програми можуть мати десятки класів, однак тільки один може (і повинен) мати метод *main()*. Виняток становлять аплети – їхній запуск здійснює *Web*-броузер за допомогою інших засобів.

У методі *main()* один параметр *String args[ ]* (*String* з великої букви тому, що це клас). У цей масив записуються параметри командного рядка, якщо їх задають під час запуску програми. В програмі з елементів масиву *args[0], args[1]* тощо можна зчитувати і використовувати параметри, задані у командному рядку. Кількість переданих до програми параметрів можна визначити за допомогою методу *args.length()*.

Четвертий рядок програми містить виклик методу *System.out.println()*. Інженери компанії *Sun* здійснили чималу підготовчу роботу і написали значну кількість кодів, які можна використовувати в своїх програмах. Ці коди розміщено в пакетах, імена яких починаються з *java.*, *sun.* і *javax*. Клас *System* розташовано в пакеті *java.lang*, який автоматично імпортується в усі програми. Об'єкт *out* має тип *PrintStream* і метод *println()*. *Out* – це вихідний потік, який під'єднується до консолі. Оскільки сучасні операційні системи мають графічний віконний інтерфейс, консоль введення-виведення використовують, переважно, для простих демонстраційних програм. Оператор *System.out.println()* завершується крапкою з комою. Запам'ятаємо, що всі оператори в *Java* завершуються " ; ".

Усі блоки програм мовою *Java* обрамлюють фігурними дужками "{}". У наведеній програмі є тіло класу і тіло методу, тобто два вкладені блоки.

1.8.2. Компіляція програми

Після написання тексту програми його бажано розмістити в своєму робочому каталозі. Для компілятора це не має значення, проте зручно зберігати вихідний код мовою *Java* і файли класів в

одному каталозі. Відкриваємо вікно командної оболонки. Переконайтесь, що поточним є ваш робочий каталог. Після цього необхідно задати команду:

javac Example.java

Ім'я класу необхідно задавати з урахуванням регістра, незважаючи на те, що, наприклад, у *Windows* імена файлів не залежать від регістра.

1.8.3. Запуск програми

Компілятор створює файл *Example.class*, який містить байткод програми, тобто інструкції з виконання інтерпретатором java. У командній стрічці необхідно задати (поточним має бути робочий каталог з файлом класу)

java Example.class

Розширення *.class* можна не задавати. Якщо файл буде знайдено, то інтерпретатор виконає код, який є в заданому класі, тобто виведе в командний рядок: "Перша програма мовою Java".

1.8.4. Проблеми

Під час написання і запуску *Java*-програми часто виникають такі проблеми:

1. Командна оболонка не може знайти файл компілятора javac. Необхідно перевірити, чи змінна PATH має значення підкаталога BIN каталога SDK.
2. Змінна CLASSPATH не вказує каталогів з класами.

1.9. Перший аплет

Мовою *Java* можна писати ще один тип програм: аплети (applets). Це невеликі програми, які розташовані на сервері, поєднаному з *Internet*. Аплети передаються мережею, автоматично встановлюються і запускаються як частина *Web*-документа.

Аплети принципово відрізняються від застосувань за структурою і деякими ключовими областями. В застосуванні може використовуватися або інтерфейс командного рядка (як в описаній вище першій програмі *Example*), або графічний інтерфейс користувача (GUI). В аплетах можна використовувати тільки GUI.

Створення програм з використанням GUI є набагато складнішим, ніж тих, які використовують командний рядок.

Java є об'єктно-орієнтованою мовою. Інженери компанії *Sun* створили завершений аплет, який можна використовувати в ролі заготовки. Все, що потрібно для написання програми-аплета, − це створити клас, який успадковує клас ***Java.applet.Applet***, і перевизначити методи, які не задовольняють наших вимог. Нижче наведено код найпростішого аплета:

```
import java.applet.Applet;
import java.awt.Graphics;
   public class HelloApplet extends Applet
      {   public void paint(Graphics g)
            { g.drawString ("Перший аплет", 20, 20); }
       }
```

Оператори ***import*** дають змогу імпортувати класи, і в тексті програми можна писати просто ***Applet*** і ***Graphics*** замість повного імені j***ava.applet.Applet*** і ***java.awt.Graphics***.

Написавши простий оператор

HelloApplet extends Applet,

ми отримуємо готовий аплет, який успадковує всі змінні і методи базового класу ***Applet***, створеного компанією *Sun*.

При створенні аплета для відображення інформації необхідно використовувати методи малювання, а не виведення в потік. Доступ до графічного простору, наданого броузером, здійснюється за допомогою об'єкта ***Graphics***. Його необхідно передати в якості параметра методу ***paint()***. Метод ***paint()*** викликається щоразу, коли вікно аплета повинно знову з'явитися на екрані.

Аплет компілюється так само, як і проста програма. Проте виконання аплета можна здійснити через:

1) *Java*-сумісний *Web*-бровзер (який підтримує тег APPLET);
2) Викликом ***appletviewer*** з SDK.

Для запуску аплета на виконання необхідно написати невеликий фрагмент коду HTML:

```
< HTML>
<BODY>
<APPLET code ="HelloApplet.class" width="200" height="200">
</APPLET>
```

</BODY>
</HTML>

Якщо цей код HTML розмістити у файлі RunApp.html, то аплет можна запустити командою

appletviewer RunApp.html

з командного рядка, або шляхом відкривання цієї сторінки в *Java*-сумісному *Web*-бровзері.

ІІ. ОСНОВИ МОВИ *JAVA*

2.1. Типи даних, змінні, масиви

Java – мова, яка чітко дотримується типів, що значною мірою гарантує безпеку програм і стійкість до помилок. Що це означає? По-перше, кожна змінна має свій тип, і кожний тип чітко визначено. По-друге, усі присвоєння, які виконують в явному вигляді або через пересиляння параметрів у методи, перевіряються на співпадання типів. У *Java* не передбачено автоматичне приведення типів або перетворення конфліктуючих типів, як в інших мовах. Компілятор *Java* перевіряє усі вирази і параметри щодо сумісності типів.

2.1.1. Прості типи

До простих типів у *Java* належать чотири групи:

1. Цілочисельні: ***byte, short, int*** і ***long***, які призначені для цілих зі знаком;
2. Числа з плаваючою комою: ***float*** і ***double*** – числа з дробовою частиною;
3. Символи: ***char*** − всі символи, включаючи букви і цифри;
4. Логічні: ***boolean*** – "істина" і "фальш".

Ці типи використовують у програмах, об’єднують в масиви тощо.

Чому введено термін *прості типи*? Тому, що вони не є об’єктами. Незважаючи на те, що в усьому іншому *Java* – цілковито об’єктно-орієнтована мова, прості типи є винятком. Все пояснюється ефективністю. Перетворення простих типів в об’єкти спричинило б до суттєвого зниження продуктивності.

Для простих типів мови *Java* явно задається діапазон значень і чітко визначені правила виконання операцій над ними. У мові *C++* діапазон цілочисельних значень може залежати від середовища виконання програми. У мові *Java*, виходячи з потреб мобільності, усі типи даних мають строго визначений діапазон. Це дає змогу писати програми з впевненістю, що вони будуть працювати на комп’ютерах з будь-якою архітектурою. Строге задання розмірів цілого приводить до деякого зниження швидкості вико-

нання програм у деяких середовищах, однак цього не уникнути, коли головною метою є мобільність.

Цілочисельні типи. У *Java* визначено чотири цілочисельні типи: *byte, short, int* і *long*. Усі вони визначають знакові, тобто додатні і від'ємні значення. *Java* не підтримує беззнакові цілі. У табл. 2.1 наведено ширину і діапазони цілочисельних типів.

Таблиця 2.1. **Ширина в бітах і діапазони значень цілочисельних типів**

Тип	Ширина	Діапазон
long	64	від - 9223372036854775808 до 9223372036854775807
int	32	від – 2147483648 до 2147483647
short	16	від – 32768 до 32767
byte	8	від – 128 до 127

Ширину цілочисельного типу, не дивлячись на те, що її вимірюють в бітах, треба розглядати не як об'єм пам'яті, яку він займає, а як поведінку, визначену для змінних і виразів заданого типу. Виконуване середовище *Java* використовує свою кількість байтів, яка краще відповідає апаратній платформі, забезпечуючи у цьому випадку поведінку, визначену через заданий програмістом тип. Скоріше всього виконуване середовище *Java* відводить на деяких платформах 32 біти для типів *byte* і *short*.

Змінні типу *byte* особливо корисні при роботі з потоком даних або файлом, а також з неформатованими даними.

Short – цей тип рідко використовують, тому що розміщення старшого і молодшого байтів не завжди збігається з реальним підходом, який використовують переважно на 16-тирозрядних комп'ютерах.

Int – найвживаніший цілочисельний тип. Його необхідно використовувати при створенні лічильника, індексації масивів, виконанні обчислень над цілими числами. Іноді здається, що використання *byte* чи *short* економить пам'ять, однак найвірогідніше, що конкретне середовище виконання все одно перетворить ці типи в *int*. Необхідно пам'ятати, що тип в мові *Java* визначає поведінку, а не розмір пам'яті. Єдиним винятком є масив, коли *byte* гарантує використання 8-ми бітів на елемент масиву, *short* – 16-ти бітів, а *int* – 32-х бітів.

Змінні типу *long* необхідні в тих випадках, коли тип *int* є замалим для збереження бажаного результату.

Дійсні типи. У *Java* існує два дійсних типи *float* і *double* для представлення чисел з одинарною і подвійною точністю відповідно. Основні характеристики цих типів наведено в табл. 2.2.

Т а б л и ц я 2.2. **Ширина в бітах і діапазони значень дійсних типів**

Тип	Ширина	Діапазон
double	64	від –1.7e-308 до 1.7e+308
float	32	від –3.4e-038 до 3.4e+038

Змінні типу *float* використовують для представлення чисел з дробовою частиною, проте з незначною точністю. На деяких процесорах операції зі змінними типу *float* виконуються швидше, ніж з типами *double* і вимагають менше пам'яті. Проте коли значення стають дуже великими або дуже малими втрачається точність обчислень.

Тип *double* використовують для представлення дійсних чисел з подвійною точністю і він вимагає 64 біти пам'яті.

На деяких сучасних процесорах, оптимізованих для високошвидкісних математичних обчислень, операції з подвійною точністю виконуються швидше ніж, з одинарною. Наприклад, усі математичні функції такі як *sin(), cos(), sqrt()* повертають значення типу *double*. Якщо необхідно виконати багато ітерацій або оперувати з великими числами, краще вибирати *double*.

Символи. Виходячи з того, що мову *Java* створено для всесвітнього використання, для кодування символів використовують Unicode (повну інформацію можна отримати за адресою http://www.unicode.org). Unicode визначає повний міжнародний символьний набір, який дає змогу представляти усі символи мов усіх народів світу. Тому необхідно для символу 16 бітів. Отже, *char* у мові *Java* є 16-тибітовим типом. Значення типу *char* є цілі числа з діапазону від 0 до 65536. Стандартний набір символів, відомий як ASCII, займає інтервал від 0 до 127, а розширений 8-мибітовий символьний набір (ISO-Latin-1) - від 0 до 255. Значення змінних *char* можна додавати та інкрементувати.

Логічний тип. У мові *Java* є тип *boolean*, який використовується для задання двох значень: *true* ("істина") і *false* ("фальш"). На відміну від мови *C++*, де будь-яке число відмінне від нуля є істина, в мові *Java* цей тип не є числом.

2.1.2. Літерали

Цілі літерали. Будь-яке ціле число є цілим літералом. У *Java* можна використовувати цілі числа у десятковій, вісімковій і шістнадцятковій системі числення. Вісімкові літерали розпочинаються з нуля, тому 0 не використовують на початку десяткових чисел. Наприклад, значення 09 викличе помилку компілятора, тому що це запис цілочисельного вісімкового літералу і "9" не є цифрою вісімкової системи числення (від 0 до 7). Шістнадцятковий літерал розпочинається з символів ∅x або ∅X, за якими знаходяться декілька шістнадцяткових цифр. Можна використовувати верхній і нижній регістри для цифр A-F.

Цілі літерали створюють значення типу *int*. Незважаючи на те, що мова *Java* чітко типізована, цілі літерали можна присвоювати усім цілочисельним типам (для *byte* і *short* літеральні значення не повинні виходити за діапазони значень). Для задання літерала типу *long* необхідно вказати про це компілятору, додаванням наприкінці букви L або *l*.

Літерали з плаваючою комою. Дійсні числа, які є літералами з плаваючою комою, можуть бути задані у вигляді стандартної або наукової нотації. Стандартна нотація − це число, крапка, дробова частина (2.3, 3.1415926, 0.667). Наукова нотація − це стандартна нотація плюс суфікс, який задає степінь числа 10, на який має помножитися дійсне число (6.022E23, 31415E-05, 2e+100).

Дійсні літерали за замовчуванням перетворюються в *Java* до подвійної точності. Для задання літерала типу *float* необхідно додати в кінці літералу F або f. Можна також явно вказати, що це літерал з подвійною точністю, задавши наприкінці букву D або d.

Логічні літерали. Існує два значення *true* і *false*, які мають тип *boolean*, і є логічними літералами. Ці значення не приводяться ні до 1, ні до 0. У мові *Java* їх можна присвоїти тільки змінним, оголошеним як *boolean*, або використати у виразах для логічних операторів.

Символьні літерали. Літеральний символ подається в середині одинарних лапок: 'a', 'z', 'D' і т. д. Для символів, які не можна ввести з клавіатури, задаються escape-послідовності. У табл.2.3 наведено деякі приклади символьних літералів.

Т а б л и ц я 2.3. Задання символьних літералів

Описання або escape-послідовність	Послідов-ність	Результат
Будь-який символ, напрклад, у	'y'	y
Повернення на один символ (BS)	'\b'	повернення (backspace)
Горизонтальна табуляція (HT)	'\t'	табуляція (tab)
Перевід стрічки (LF)	'\n'	переведення рядка (новий рядок)
Перевід сторінки (FF)	'\f'	переведення сторінки (нова сторінка)
Повернення каретки (CR)	'\r'	повернення каретки
Подвійна лапка	\"	"
Одинарна лапка	\'	'
Обернена риска	\\	\
Вісімковий набір бітів	\ddd	вісімковий символ ddd
Шістнадцятковий набір бітів	\xdd	шістнадцятковий символ dd
Символ Unicode	\udddd	символ з кодом dddd в Unicode

Рядкові літерали. Рядкові літерали задають шляхом обрамлення подвійними лапками послідовності символів: "Добрий день", "дві\nстрічки", "\"Фраза у подвійних лапках\"".

Важливо пам’ятати, що рядкові літерали повинні починатися і закінчуватися в одному рядку, оскільки в мові *Java* немає escape-послідовності продовження рядка. Зауважимо, що рядки в *Java* є об'єктами класу *String*, а не масивами символів як *C++*.

2.1.3. Змінні

Змінна - це базова одиниця для збереження інформації в програмі мовою Java. Вона визначається комбінацією типу, ідентифікатора і необов’язкової ініціалізації. Змінна має область видимості і час існування.

Оголошення змінної. У мові *Java* всі змінні повинні бути оголошені ще до їхнього використання. Основна форма оголошення змінної має вигляд:

тип ідентифікатор [=значення][,ідентифікатор[=значення]...];

Під типом розуміють один із простих типів *Java*, ім'я класу або інтерфейсу. Деякі приклади оголошення змінних у *Java*:

```
int a,b,c;
int d=2, e, f=5;
byte z=22;
double pi=3.1415926;
char x;
```

Динамічна ініціалізація. *Java* допускає динамічну ініціалізацію змінних не тільки константами, але й виразами.

```
// Демонстрація динамічної ініціалізації змінних
class DynInit {
  public static void main(String args[ ]){
     double a=3.0,b=4.0; // звичайна ініціалізація
//оголошення і динамічна ініціалізація
     double c=Math.sgrt (a*a+b*b);
     System.out.println ("Довжина гіпотенузи"+c);
   }
}
```

Вираз для ініціалізації змінної може використовувати будь-який елемент, доступний в момент ініціалізації: виклик методу, використання змінних або літералів. У нашому прикладі використано метод *sgrt* з класу *Math* у виразі для ініціалізації змінної *c*.

Область доступності і час існування змінних. У *Java* можна оголошувати змінні не тільки на початку методу, (наприклад, *main()*), але й всередині будь-якого блоку. Блок − це частина програмного коду, яка починається з відкриваючої фігурної дужки { і закінчується закриваючою фігурною дужкою }. Усередині блоку змінна може бути оголошена в будь-якому місці, але до її першого використання.

Залежно від того, де оголошено змінну, задається область доступності (видимості) до неї. В *Java* область доступності змінної визначається класом, методом і блоком. Область доступності, яку визначають класом, розглядатимемо далі. Тепер розглянемо області, які визначають всередині методу. Область доступності, яку визначають методом, розпочинається з фігурної дужки. Якщо метод має параметри, то вони також доступні в усьому методі. Якщо змінну оголошено всередині блоку, вона є недоступною

ззовні цього блоку. Отже, ця змінна є захищена від несанкціонованого доступу і зміни. Правила визначення області доступності забезпечують фундамент для інкапсуляції.

Області доступності можуть бути вкладеними. Це означає, що об'єкти, оголошені в зовнішній області, будуть доступні для коду внутрішнього блоку. Але об'єкти, оголошені у внутрішньому блоці, не будуть доступними ззовні.

```
// Демонстрація області доступності змінної блоку
  class Scope {
  public static void main(String args[ ]){
  int x; // змінна доступна всюди в main
  x=10;
  if(x==10) { //початок нового блоку
  int y=20; //змінна y доступна тільки в цьому блоці
  System.out.println ("x i y" + x + " " + y);
  x=y+2;
            } // завершення блоку
  // y=100; Помилка! Змінна y вже недоступна
  System.out.println("x є" + x);
  }  // завершення методу
  }// завершення класу
```

Важливо також пам’ятати, що змінні створюються в момент входу в їхню область доступності і знищуються під час виходу з неї. Отже, змінні не зберігають своїх значень при повторному звертанні до методу і т. п.

У мові *Java* не можна оголошувати з одним іменем змінну в укладених блоках (у мові *C++* це можливо). Наприклад, така програма не буде правильно компілюватися:

```
  class ScopeErr {
  public static void main(String args[ ]){
  int bar=1;
        {
  int bar=2; //Помилка! bar вже оголошена вище
        }
      }
  }
```

2.1.4. Перетворення і приведення типів

У процесі програмування присвоєння значень одного типу змінній іншого типу є звиклим явищем. Якщо два типи сумісні, в *Java* автоматично виконається таке перетворення. Проте не всі типи сумісні і тому не всі перетворення типів дозволені в неявному виді.

Автоматичне перетворення типів. У *Java* можливе автоматичне перетворення типів за умови виконання двох умов:

1) два типи сумісні;
2) тип призначення більший від вихідного типу.

При виконанні цих двох умов виконується перетворення з розширенням (*widening conversion*). Наприклад, змінної типу *int* достатньо, щоб прийняти всі значення типу *byte*.

При перетворенні з розширенням числові типи сумісні одні з одними, проте несумісні з *char* і *boolean*. Крім того, *char* і *boolean* несумісні між собою.

Приведення несумісних типів. Перетворення типу *int* у *byte* не виконається автоматично. Цей вид перетворення іноді називають перетворенням зі звуженням (*narrowing conversion*), тому що значення скорочується, щоб підійти під тип призначення. Для перетворення між двома несумісними типами необхідно виконати операцію приведення. Приведення (*cast*) − це перетворення типів у явному вигляді. Воно матиме загальний вигляд:

(тип-призначення) значення,

де *тип-призначення* задає тип, до якого необхідно перетворити значення.

Під час приведення типу *int* до типу *byte*, якщо значення цілого більше діапазону представлення *byte*, воно буде вкорочене за модулем діапазону типу *byte* (тобто буде взято залишок від ділення цілого на діапазон типу *byte*).

Внаслідок присвоєння дійсного числа цілому виконується інший тип перетворення: відсікання (*trancation*), тобто відкидається дробова частина. Якщо цілочисельна частина перевищуватиме можливе значення для цілого типу, то вона буде вкорочена за модулем діапазону типу призначення.

```
// Демонстрація приведення типів
class Conversion {
```

```
public static void main(String args[ ]){
byte b;
int i=257;
double d=323.142;
b=(byte)i;
System.out.println("i в b :" + i +" "+ b);
i=(int)d;
System.out.println("d в i :" + d +" "+ i);
b=(byte)d;
System.out.println("d в b :" + d +" "+ b);
    }
  }
```

Результат роботи цієї програми матиме вигляд:

i в *b* : 257 1

d в *i*: 323.142 323

d в *b* : 323.142 67

Аналізуючи ці результати необхідно пам'ятати, що діапазон для типу ***byte*** є 256.

Автоматичне перетворення типів у виразах. У мові *Java* визначено декілька правил перетворення типів у виразах:

- змінні типу ***byte*** і ***short*** завжди автоматично перетворюються в *int;*
- якщо один з операндів має тип ***long***, увесь вираз перетворюється в *long;*
- якщо один операнд має тип ***float***, то увесь вираз перетворюється у *float;*
- якщо один операнд має тип ***double***, то результат буде типу ***double***.

Незважаючи на вигідність автоматичного перетворення типів у виразах, воно може спричинити помилки компілятора. Наприклад, коректний на перший погляд код може бути джерелом проблеми:

```
byte b=50;
b=b*2; //Помилка! Тип int не присвоюється byte.
```

Виявляється, що значення 100 цілком допустиме для ***byte*** не може бути збережене в ***b***. Внаслідок обчислень операнди були автоматично перетворені в тип ***int***, результат також типу ***int***. Без

виконання операції приведення його не можна присвоїти типу *byte*. Це є справедливим навіть у тому випадку, коли значення належить діапазону призначення.

У випадках, коли виникають сумніви, краще задати явне приведення типів. Коректний код має вигляд:

```
byte b=50;
b=(byte)b*2;
```

2.1.5. Масиви

Масив (*array*) – це група однотипних елементів, які "відгукуються" на одне спільне ім'я. Масиви можна створювати будь-якого типу і довільного розміру. Доступ до заданого елемента масиву здійснюється за допомогою індексів.

Організація масивів у мові *Java* інша, ніж у *C++*.

Одновимірні масиви. Формат оголошення одновимірного масиву має вигляд:

тип ім'я_змінної[];

Таке оголошення засвідчує, що *ім'я_змінної* – це змінна масиву, проте вона насправді ще не існує і має значення *null*. Для того, щоб зв'язати *ім'я_змінної* з реальним фізичним масивом, необхідно виділити пам'ять за допомогою оператора *new*

```
ім'я_змінної=new тип[розмір];
```

Елементи масиву в *Java* автоматично ініціалізуються нулями. Усі індекси масиву розпочинаються з нуля. Можна об'єднати оголошення змінної масиву з виділенням пам'яті:

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

Масиви можна ініціалізувати аналогічно як і прості змінні. У цьому випадку розмір масиву визначається автоматично кількістю елементів ініціалізації. Ініціалізатор масиву – це список виразів, розділених комою і розміщених у фігурних дужках.

```
// Демонстрація ініціалізації масиву
class AutoArray{
    public static void main(String args[ ]){
        int month_days[ ]={31,28,31,30,31,30,31,31,30,31,30,31}
        }
   }
```

Компілятор *Java* стежить за індексами масиву, щоб не вийшли за межі масиву. Виконавча система *Java* обов'язково перевірить усі індекси масивів. У випадку виходу за межі видасть помилку виконання. Це також відрізняє мову *Java* від мови *C++*.

Багатовимірні масиви. У мові *Java* багатовимірні масиви (як і в *C++*) є насправді масивами масивів. Щоб оголосити багатовимірний масив, необхідно задати додатковий індекс за допомогою ще одного набору квадратних дужок:

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

Другий індекс визначає стовпець, перший рядок. Виокремлюючи пам'ять для багатовимірних масивів, необхідно визначити пам'ять тільки для першого виміру. Виділення пам'яті для решти вимірів можна здійснити окремо. У цьому випадку кількість елементів у кожному рядку може бути різною. Наступний програмний код створює двовимірний масив для трикутної матриці.

```
// Демонстрація двовимірного масиву
class TwoDAgain {
        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];
                  }
    }
```

Багатовимірні масиви ініціалізуються даними між двома парами фігурних дужок. Напиклад:

```
class Matrix{
        public static void main(String args [ ]){
                double m[ ][ ]= {
                        {0,  1,   2,   3}
                        {4,  5,   6,   7}
                        {8,  9,   10, 11}
                        {12, 13, 14, 15}
                                        };
            };
    };
```

Альтернативний синтаксис оголошення масивів. Існує альтернативний формат оголошення масиву

тип[] ім'я_змінної;

Такі оголошення є еквівалентними:

```
int a1[ ]=new int[3];
 int[ ] a2=new int[3];
char twod1[ ][ ]=new char[3][4];
char [ ][ ] twod2=new char[3][4];
```

Зауваження. Оголошення масивів оператором *new* вказує на те, що масиви в мові *Java* – це об'єкти. Число елементів, які можна розмістити в масиві, зберігається в екземплярі змінної *length*. Наприклад, *a1.length* дорівнює 3.

2.2. Операції

Операції в мові *Java* можна розділити на такі групи:

- арифметичні;
- порозрядні;
- логічні;
- операції відношення.

Далі у тексті наведемо зведені таблиці цих операцій. Детальнішу інформацію можна подивитися в [9].

2.2.1. Арифметичні операції

Арифметичні операції використовуються при програмуванні математичних виразів. Їхній перелік наведено в табл. 2.4.

2.2.2. Побітові операції

У мові *Java* визначено декілька побітових операцій (табл. 2.5), які застосовують до цілочисельних типів (*long, int, short, char* і *byte*). Такі операції виконуються над конкретними бітами своїх операндів.

Усі цілочисельні типи представлено степенями числа 2. Наприклад, $42 = 2 + 8 + 32 = 2^1 + 2^3 + 2^5 = 00101010$. Від'ємні числа в мові *Java* представляють як доповнення до 2. Тобто від'ємні числа спочатку піддають інверсії (1 заміняються на 0, 0 – на 1), а потім

додають 1 (*мінус сорок два* записують як 11010101, *плюс один* дає 11010110). Найстарший розряд визначає знак цілого.

Т а б л и ц я 2.4. **Арифметичні операції**

Операція	Результат
+	Додавання
-	Віднімання (а також унарний мінус - від'ємне значення)
*	Множення
/	Ділення
%	Ділення за модулем (залишок від ділення, цілі і дійсні)
++	Інкремент (збільшення на одиницю операнда, є постфіксна і префіксна форми)
+=	Присвоєння результату додавання (a=a+2 еквівалентно a+=2)
-=	Присвоєння результату віднімання
*=	Присвоєння результату множення
/=	Присвоєння результату ділення
%=	Присвоєння результату ділення за модулем
--	Декремент (зменшення на одиницю операнда, є постфіксна і префіксна форми)

Т а б л и ц я 2.5. **Побітові операції**

Операція	Результат
~	Порозрядне унарне заперечення (NOT)
&	Порозрядне "І" (AND)
|	Порозрядне АБО (OR)
^	Порозрядне заперечення АБО (OR)
>>	Зсув вправо
>>>	Зсув вправо з заповненням нулем
<<	Зсув вліво
&=	Порозрядне присвоєння результату І
|=	Порозрядне присвоєння результату АБО
^=	Порозрядне присвоєння результату заперечуючого АБО
>>=	Порозрядне присвоєння результату зсуву вправо
>>>=	Порозрядне присвоєння результату зсуву вправо з заповненням нулем
<<=	Порозрядне присвоєння результату зсуву вліво

2.2.3. Операції відношення

У мові *Java* за допомогою перевірки на рівність, нерівність тощо (табл. 2.6) можна порівнювати значення будь-якого типу: цілі, числа з плаваючою комою, символи, логічні типи.

Т а б л и ц я 2.6. **Операції відношення**

Операція	Результат
==	Рівність
!=	Нерівність
>	Більше ніж
<	Менше ніж
>=	Більше або рівне
<=	Менше або рівне

Зауваження 1. Для об'єктів оператор "==" перевіряє, чи вказують покажчики на один об'єкт, а не рівність двох об'єктів. Для перевірки на рівність (співпадання) об'єктів є метод *equals()*.

Зауваження 2. У мові *C++* будь-яка відмінна від нуля величина має значення *true, 0 - false*. Тому в *C++* правильним буде код:

```
int done;
...
if (!done) { };
```

Мовою *Java* необхідно писати у таких випадках тільки з явною перевіркою на рівність нулю:

```
int done;
...
if (done!=0) {};
```

2.2.4. Логічні операції

Логічні операції виконують тільки з операндами типу *boolean*. Їхній перелік наведено у табл. 2.7. Логічні операції виконують над значеннями типу *boolean* аналогічно, як вони діють над бітами цілих типів.

Перевагу використання логічних обчислень за скороченою схемою демонструє фрагмент програми:

if(denom!=0 && num/denom>10).

Т а б л и ц я 2.7. **Логічні операції**

Операція	Результат
&	Логічне І
|	Логічне АБО
^	Логічне заперечуюче АБО
||	Обчислення АБО за короткою схемою
&&	Обчислення І за короткою схемою
!	Логічне упарне НІ
&=	Присвоєння результату логічного І
|=	Присвоєння результату логічного АБО
^=	Присвоєння результату логічного заперечуючого АБО
==	Рівність
!=	Нерівність
?:	Тернарний оператор if-then-else

2.2.5. Операція присвоєння

Операцію присвоєння виконують у мові *Java* аналогічно як і в усіх інших мовах програмування. Вона має такий вигляд:

змінна = вираз;

Тип змінної повинен бути сумісним з типом виразу. Ця операція дає змогу створювати ланцюжок присвоєнь: *x=y=z=100;*

2.2.6. Умовна операція «?:»

Загальна форма (тернарної, тобто операції, яка має три операнди):

вираз1? вираз 2: вираз 3

У цій формі *вираз 1* повинен повертати тип *boolean*. Якщо *вираз 1* істинний, то обчислюють *вираз 2*, інакше *вираз 3*. *Вираз 2* і *вираз 3* повинні повертати значення одного типу. Використання цієї операції демонструє наступний код:

```
// Отримання абсолютного значення числа і
clas Ternary {
public static void main(String args[ ]){
int i,k;
i=-10;
k=i<0?-i:i;
  }
}
```

2.2.7. Пріоритети операцій

У мові *Java* погоджено такий порядок виконання операцій у виразах:

1.	( )	[]	.	
2.	++	--	~	!
3.	*	/	%	
4.	+	-		
5.	>>	>>>	<<	
6.	>	>=	<	<=
7.	==	!=		
8.	&			
9.	^			
10.	|			
11.	&&			
12.	||			
13.	?:			
14.	=	операція=		

Бажано використовувати круглі дужки з метою спрощення програми, тому що їх наявність не впливає на швидкість виконання програми. Наприклад, не викликає сумніву, який з цих двох виразів зручніше прочитати:

a|4+c>>b&7||b>a%3 чи *(a|(((4+c)>>b)&7))||(b>(a%3))*

2.3. Оператори

Оператор − це певною мірою довільна стрічка програмного коду, яка закінчується крапкою з комою. Оператором може бути вираз, виклик методу, оголошення. Об'єднані фігурними дужками в блок оператори утворюють складений оператор.

Послідовне виконання коду можна змінювати, використовуючи оператори керування. У мові *Java* маємо три категорії операторів керування: вибору, ітерації та переходу.

2.3.1. Оператори вибору

Виконання тієї чи іншої частини коду програми залежно від виконання певної умови організовується за допомогою оператора *if*. Його загальний вигляд:

```
if (умова) оператор 1;
else оператор 2;
```

Оператор 1, оператор 2 – це не тільки один оператор, але й складений оператор. Варіанта ***else*** може не існувати. Оператори ***if*** можуть бути вкладеними. Вони можуть утворювати ланцюжок операторів:

```
if (умова)
    оператор;
  else if (умова)
      оператор;
        else if (умова)
                оператор;
        …
                else
                  оператор;
```

Якщо одна з умов стає ***true***, то виконується оператор, який є наступним за цим ***if***. Решта операторів пропускають.

Подібним до оператора ***if*** є оператор ***switch***, який дає змогу здійснювати розгалуження програмного коду в декількох напрямах. Він має вигляд:

```
switch (вираз)
{
    case значення 1:
       //послідовність операторів
    break;
    case значення 2:
      //послідовність операторів
    break;
     ...
    case значення N:
      //послідовність операторів
    break;
  default:
      //послідовність операторів
   break;
}
```

Вираз може повертати значення будь-якого простого типу. Кожне значення, задане в операторах *case*, повинне мати тип, сумісний з типом *виразу*. Значення в *case* повинні бути літералами, а не змінними. Оператори *case* з однаковими значеннями недопустимі.

Оператор *switch* працює так: після обчислення значення виразу в заголовку оператора порівнюють його результат послідовно зі значеннями усіх констант в усіх варіантах *case*. Виконується послідовність операторів, які йдуть за варіантом *case*, що відповідає значенню виразу. Якщо ні одна з констант не збігається зі значенням виразу, то виконується послідовність операторів, заданих за замовчуванням (після *default*). Оператор *default* може бути відсутнім.

Оператор *break* є ознакою закінчення послідовності операторів і використовується для виходу з оператора, який безпосередньо його містить. У нашому випадку він передає керування на перший рядок за фігурною закриваючою дужкою оператора *switch*. Оператор *break* є необов'язковим. Якщо його пропустити, то виконаються усі наступні оператори *case*, доки не зустрінеться оператор *break* або не буде завершення оператора *switch*.

Оператори *switch* можна використовувати в складі операторів, які належать до іншого оператора *switch*. Так як *switch* визначає свій внутрішній блок {}, між операторами *case* внутрішнього і зовнішнього *switch* не виникає конфліктів.

Три важливі особливості оператора *switch*:

1) на відміну від оператора *if* оператор *switch* може здійснювати перевірку тільки на рівність;
2) не допускається в одному операторі *switch* двох однакових значень в *case;*
3) оператор *switch* є ефективнішим, ніж набір (ланцюг) операторів *if*.

2.3.2. Ітераційні оператори (циклу)

Ітераційні оператори (у мові *Java* − *for, while, do-while*) дають змогу створювати структури, які називаються циклами (*loop*). Цикли, або ітераційні оператори, дають змогу повторювати виконання операторів або груп операторів. Число повторень у

деяких випадках фіксоване, а в інших визначається під час обчислень внаслідок перевірки умов виходу з циклу.

Оператор *while* є базовим оператором циклу в мові *Java*. Він повторює виконання оператора або складеного оператора до того часу, доки керуючий вираз (умова) є істинним. Синтаксис циклу *while* такий :

```
while (умовний вираз) {
        // тіло циклу
 }
```

Якщо в циклі виконується один оператор, фігурні дужки не обов'язкові. Якщо значення умовного виразу є *false*, то тіло циклу не виконуватиметься жодного разу. Тіло циклу *while* може бути пустим. Такий цикл використовують в наступному коді програми.

```
//Демонстрація циклу
class NoBody {
        public static void main{String args[ ]){
                int i,j;
                i=100;
                j=200;
        // знаходження середнього між i і j
                while (++i < --j);
                        System.out.println ("середнє рівне
                " + i);
                }
}
```

Зрозуміло, що в циклі *while* перевірка умови здійснюється перед виконанням тіла циклу. Можлива ситуація, коли тіло циклу не виконується жодного разу. Інколи необхідно виконати тіло навіть тоді, коли умова не виконується. У цих випадках використовують оператор циклу *do-while.* Синтаксис оператора:

```
do {
    // тіло циклу
  } while (умовний вираз);
```

У циклі *do-while* перевірка умови здійснюється після виконання тіла циклу. Тому у випадку використання оператора *do-while* тіло циклу виконується хоча б один раз.

Ще однією формою оператора циклу в *Java* є ***for.*** Конструкція цього оператора виглядає так:

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

Цикл ***for*** працює так. Перед початком роботи виконуються дії, вказані в частині ініціалізації. Найчастіше − це ініціалізація змінної керування циклом, яка виконує роль лічильника, що керує роботою циклу. Ініціалізаційний вираз виконується тільки один раз. Далі перевіряється умова, яка є булевим виразом. Переважно тут порівнюється змінна керування циклом з її кінцевим значенням. Якщо умова є істинна, цикл виконується, якщо – ***false***, то тіло циклу не виконується. Після цього виконується частина циклу, яку називають ітерацією. Переважно – це вираз, де змінна циклу збільшується або зменшується на одиницю. Потім виконується властиво цикл, протягом якого внаслідок кожного проходження спочатку обчислюється умовний вираз, потім виконується тіло циклу, а після цього – ітераційний вираз. Цей процес повторюється доти, доки умовний вираз не стане ***false***. Цикли ***for*** можуть бути вкладеними.

Часто змінну циклу використовують тільки в цьому циклі і більше ніде. У цьому випадку можна оголосити цю змінну в ініціалізаційній частині оператора ***for***. Наприклад,

```
// Оголошення змінної керування циклом всередині for
class ForTick {
        public static void main(String args[ ]){
            for (int n=10; n>0; n--)
              System.out.println("Обернений рахунок-"+n);
                }
       }
```

Інколи необхідно, щоб в ініціалізаційній та ітераційній частинах циклу ***for*** виконувалось не по одному оператору. В цьому випадку ці оператори треба розділити комою. Наприклад,

```
// Використання ком в операторі циклу
class Comma {
        public static void main(String args[ ]){
                int a,b;
                for (a=1, b=4; a<b; a++, b--){
```

```
                    System.out.println("a="+a);
                    System.out.println("b="+b);
                    }
            }
    }
```

Існує декілька модифікацій циклу *for* тому, що кожна з трьох або всі три частини циклу – ініціалізаційна, перевірочна та ітераційна – не обов'язково використовуються для вказаних цілей і взагалі можуть бути відсутніми, проте їхнє розділення крапкою з комою (;) опускати не можна. Наприклад, організація безмежного циклу виглядатиме так:

```
for ( ; ; ){…};
```

2.3.3. Оператори переходу

У мові *Java* передбачено три оператори переходу: *break, continue, return*. Вони призначені для передавання керування іншій частині програми. Є ще один засіб змінити послідовність виконання операторів у програмі – це обробка виняткових ситуацій. Оператор *break* використовують у мові *Java* у трьох випадках:

- вихід з оператора *switch;*
- вихід з циклу;
- він є "цивілізованим" оператором *goto*.

За допомогою оператора *break* можна вийти з циклу примусово, проігнорувавши умовний вираз і всі оператори, які йдуть за *break*. Керування передається наступному за тілом циклу оператору. Оператор *break* можна використати у будь-якому циклі, навіть у нескінченному. Якщо *break* використовується у вкладених циклах, то вихід відбувається тільки з внутрішнього (одного) циклу. У циклі може бути декілька операторів *break*.

Оператори *break*, які належать оператору *switch*, що є в тілі циклу, не впливають на хід виконання циклу.

Зауваження. Оператор *break* не є нормальним виходом з циклу. Для цього існує умовний вираз. Його використання в циклах бажане тільки в особливих випадках.

Часто буває необхідно вийти з глибоко вкладеного циклу. Для реалізації такої ситуації в *Java* визначено розширену форму оператора *break* :

break мітка;

Тут *мітка* – це ім'я, яке ідентифікує блок програми. Під час виконання цієї форми *break* керування передається блоку з міткою, який також повинен мати у своєму складі оператор *break*. Блок, на який передається керування не обов'язково повинен містити блок, з якого здійснюється вихід.

Для присвоєння блокові імені необхідно на його початку поставити мітку (label) – це будь-який ідентифікатор, дозволений в *Java*, після якого стоїть двокрапка (":"). При посиланні на мітку блоку з оператора *break*, керування буде передано в кінець блоку. Найчастіше *break* з міткою використовується для виходу з укладених циклів.

Оператор *continue* можна використовувати у будь-якій з трьох форм циклів. Його виконання спричиняє таку зміну логіки програми, що решта операторів тіла циклу пропускаються. Для циклів *while* або *for* відразу ж за оператором *continue* розпочинається новий крок, а для циклу *do-while* перевіряється умова на виході. Проте для усіх циклів частина коду від *continue* до кінцевого оператора циклу пропускається. Як і *break* оператор *continue* може мати мітку, яка визначає, на який із вкладених циклів здійснюється перехід. Наприклад,

```
// Використання оператора continue з міткою
class ContinueLabel  {
    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();
        }
    }
```

Ця програма виводить трикутну таблицю множення чисел від 0 до 9.

Для явного повернення з методу використовують оператор *return*.

ІІІ. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ *JAVA*

Об'єктно-орієнтоване програмування (ОПП) настільки інтегроване в *Java*, що написання навіть найпростіших програм вимагає знань базових принципів ООП. Як відомо, всі програми складаються з двох елементів: коду і даних. Вони можуть взаємодіяти з зовнішнім світом за принципом "що відбувається" або "на що впливає". Існує два стилі побудови програм. Перший − процесно-орієнтована модель. Цей підхід представляє програму як низку послідовно виконуваних операцій (такі мови програмування, як *Pascal*, *Fortran*, *C* успішно використовують цю модель). Однак внаслідок збільшення розмірів і складності програми такий підхід має значні недоліки. З метою розв'язання цих проблем розроблено другий підхід, який називають об'єктно-орієнтованим програмуванням. Програми, побудовані згідно принципу ООП, можна вважати такими, в яких дані керують доступом до коду. Перші об'єктно-орієнтовані мови виникли ще у 1967р. У 1983 р. Б'ярн Страуструп розробив першу версію *C* з класами. Спеціалісти − розробники *Java* також були програмістами на *C++*.

3.1. Теоретичні аспекти ОПП

Усі об'єктно-орієнтовані мови програмування забезпечують реалізацію базових концепцій: інкапсуляції, поліморфізму, наслідування.

3.1.1. Абстракція (класи та об'єкти)

Центральним елементом об'єктно-орієнтованого програмування є абстракція. Завдяки можливості абстрагуватися людство справляється з проявами складних речей в оточуючому світі. Наприклад, ніхто не думає про автомобіль як про набір з тисячі деталей, коли треба переїхати з одного місця в інше. Автомобіль сприймається як об'єкт в цілому з певними характеристиками: швидкість руху, необхідна кількість пального і т.п.

Існує могутній механізм створення абстракції, який полягає в використанні ієрархічної класифікації. Такий підхід дозволяє

проаналізувати семантику складних систем, розбивши їх на простіші фрагменти.

Ієрархічні абстракції складних систем можна застосувати і до комп'ютерних програм. Дані з традиційної процесно-орієнтованої програми за допомогою абстракції можна перетворити на об'єкти. А послідовність обробки цих даних може стати набором повідомлень, які передаються між об'єктами. Таким чином, кожний із об'єктів описує власну унікальну поведінку, відповідаючи на повідомлення виконати певну дію. В цьому і полягає суть ООП.

3.1.2. Інкапсуляція та передавання повідомлень

Інкапсуляція (*encapsulation* − в перекладі "герметизація) - це механізм, який зв'язує код і дані, захищаючи у цьому випадку їх від зовнішнього пошкодження і некоректного використання. Це захисна оболонка, яка забороняє довільний доступ іншому кодові, зовнішньому щодо даного. Доступ до даних і коду суворо контролюється інтерфейсом. Кожний знає як доступитися до даних, не задумуючись над деталями реалізації коду.

У мові *Java* (як і в інших об'єктно-орієнтованих мовах) основою інкапсуляції є клас. Клас визначає структуру суті, створеної даними і кодом, і поведінку цієї суті, яка, використовуючи визначення класу, може втілюватися у багатьох об'єктах. Кожний об'єкт класу повторює структуру і поведінку, визначену класом, так ніби його відлили за допомогою форми у вигляді класу. Тому об'єкти називаються екземплярами класу. Клас є логічною конструкцією, а об'єкт − його фізичним втіленням. При створенні класу визначаються дані і код. Зокрема, дані називають змінними класу, а код, який виконується над цими даними, − методами класу. Поведінку та інтерфейс класу визначають методи, які опрацьовують дані екземпляру класу. Кожний метод або змінну визначають як внутрішні або загальнодоступні. Якщо доступ до власних членів класу збоку інших частин програми можна отримати тільки через загальнодоступні методи, то можна вважати, що принцип інкапсуляції цілковито реалізовано.

3.1.3. Наслідування

Усі об'єкти реального світу можна впорядкувати в певний ієрархічний ряд (зверзу донизу). Наприклад, спанієль належить до класу собак, які є ссавцями, які належать до більш узагальненого поняття тварин. Кожний рівень ієрархічного ряду має свої специфічні характеристики. Якщо не застосовувати принцип наслідування, то для кожного об'єкта з найнижчого рівня треба було б визначати усі характеристики заново.

Цей аспект ООП дає змогу створювати нові класи, які успадковують функціональні можливості уже існуючих.

3.1.4. Поліморфізм

Концепція поліморфізму полягає в тому, що за допомогою одного інтерфейсу реалізують декілька методів. Наприклад, під час створенняі різних об'єктів працює метод *create*, хоча для кожного з них його реалізовано по-різному. Вибір конкретної дії (тобто методу) стосовно кожної ситуації перекладається на компілятор або інтерпретатор. Програмістові необхідно запам'ятати, як застосувати загальний інтерфейс: *open, close* тощо.

3.1.5. Спільна дія поліморфізму, інкапсуляції та наслідування

Найповніше ілюструє силу об'єктно-орієнтованого підходу приклад з реального життя − об'єкт "автомобіль". Усі водії використовують наслідування для управління різними типами засобів руху: автобусом, вантажним автомобілем і та іншими засобами пересування. Постійно водії зустрічаються з інкапсульованими засобами в автомобілі. Педалі гальма, газу, кермо − це частини складного механізму, які пропонують простий інтерфейс, зховуючи при цьому конкретну реалізацію дій. Нарешті, поліморфізм, демонструє здатність виробників автомобілів пропонувати все нові варіанти одного і того ж, за змістом, транспортного засобу. Адже в усіх автомобілях необхідно натиснути на педаль гальма, щоб зупинитись, повернути кермо для зміни напряму руху і т.п.

Тобто трансформація окремих деталей в об'єкт, який названо "автомобілем", досягається шляхом застосування інкапсуляції, наслідування і поліморфізму. Те ж саме справедливо і для програм. Застосовуючи об'єктно-орієнтований підхід, різні частини складної

програми можна зібрати докупи з метою створення зв'язаного, стійкого до помилок і працюючого цілого.

3.2. Ознайомлення з класами

Класи є ядром мови *Java*. Що б ми не захотіли реалізувати в *Java*-програмі, її неохідно оформити у вигляді класу. Найважливіше, що необхідно знати про клас – це те, що він визначає новий тип даних. Один раз описаний клас можна надалі використовувати з метою створення об'єктів цього класу. Отже, клас є шаблоном (*template*) об'єкта, а об'єкт – екземпляром (*instance*) класу.

3.2.1. Визначення класу

Визначення класу полягає в описі його вигляду і природи. Це відбувається за допомогою визначення даних, які в ньому містяться, і програмного коду, який керує цими даними. Загальний вигляд оголошення класу записують так:

```
class ім'я_класу{
        тип об'єктна_змінна1;
        тип об'єктна_змінна 2;
        //…
        тип об'єктна_змінна N;
        тип ім'я_методу1(список параметрів){
                // тіло методу
                }
        тип ім'я_методу2(список параметрів){
                // тіло методу
                }
        //…
        тип ім'я_методуN(список параметрів){
                // тіло методу
                }
}
```

Методи і змінні, визначені всередині класу, називають членами (*members*) цього класу.

Дані або змінні, визначені у блоці *class,* можуть належати конкретному екземплярові класу (об'єктові), внаслідок чого їх називають змінними екземпляра, або бути глобальними змінними (*instance variables*), спільними для всіх екземплярів конкретного

класу і носити назву змінних класу. З метою визначення змінної класу перед її оголошенням необхідно додати модифікатор ***static***. Статичні змінні зберігаються в одному місці оперативної пам'яті і доступні завжди протягом виконання програми з усіх екземплярів класу. Якщо ще в оголошенні змінної додати модифікатор ***final,*** то значення змінної не можна змінювати в підкласах. Фактично − це оголошення константи. За загальноприйнятими правилами імена таких змінних пишуть великими літерами.

Програмний код міститься в методах. Головні частини методу − це ім'я, параметри, тип значення, яке повертається, і тіло. Методи в мові *Java* створюються тільки як частини класу. Це її основна відмінність від інших мов програмування, в яких можна окремо написати і виконати функцію або процедуру. Список параметрів задає типи та імена для інформації, яку необхідно передати в метод. Ім'я методу і список параметрів винятково ідентифікують метод. Для виходу з методу використовують оператор ***return,*** за яким іде вираз, значення якого необхідно повернути. Метод може повертати значення довільного типу або не повертати нічого. У таких випадках на місці типу в оголошенні методу необхідно написати слово ***void***.

Як приклад простого класу наведемо клас ***Box*** (коробка)

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

Цей клас визначає тип даних ***Box***. Його можна використовувати для визначення об'єктів. Важливо пам'ятати, що описуючи клас, ми задаємо лише шаблон. Реальний об'єкт типу ***Box*** при цьому не створюється. Для того, щоб створити об'єкт типу ***Box***, необхідно записати оператор виду

```
Box mybox = new Box(); // створює об'єкт з іменем mybox
```

Для доступу до змінних класу ***Box*** необхідно використати оператор "крапка" (.). Розглянемо приклад програми, яка використовує клас Box:

```
// оголошення класу Box
class Box {
        double width;
```

```
        double heigth;
        double depth;
    }
// у цьому класі створюється об'єкт типу Box
  class BoxDemo{
    public static void main(String args[ ]){
        Box mybox=new Box();
        double vol;
    //  присвоєння значень змінним класу
        mybox.width=10;
        mybox.heigth=20;
        mybox.depth=30;
    //  обчислення об'єму коробки
        vol=mybox.width * mybox.heigth * mybox.depth;
        System.out.println ("об'єм коробки ="+vol);
      }
    }
```

Для створення об'єкта типу **Box**, використовують оператор:

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

У цьому операторі об'єднано два кроки: оголошення змінної *mybox* і створення реального об'єкта в пам'яті на стадії виконання. Його можна розбити на два:

```
Box mybox; // Оголошує посилання на об'єкт типу Box,
                // яка має значення null;
mybox=new Box();//Розміщує об'єкт типу Box в пам'яті і
                    //  присвоює mybox адресу,
                    // куди розмістився об'єкт Box.
```

Посилання на об'єкти – це покажчики на комірки в пам'яті, яку ще називають "купою". У мові *Java* не можна присвоїти цьому покажчикові довільне значення або оперувати з ним як з цілим (на відміну від *C/C++*).

Файл з програмою необхідно назвати *BoxDemo.java*, тому що метод *main()* міститься в цьому класі. Класи **Box** і **BoxDemo** можна розмістити також в окремих файлах *Box.java, BoxDemo.java*.

Детальніший погляд на оператор new. Оператор **new** динамічно (під час виконання) розміщує об'єкт в оперативній пам'яті загального призначення (купі).

Його загальний вигляд

змінна_класу= new ім'я_класу();

Ім'я класу, після якого стоять круглі дужки – це виклик конструктора класу (*constructor*). Конструктор описує дії, які виконуються при створенні об'єкта цього класу. Конструктор задається всередині визначення класу. Якщо конструктор явно не заданий (як і було в класі *Box*), *Java* автоматично викликає заданий за замовчуванням конструктор.

Оператор *new* виокремлює пам'ять для об'єкта під час виконання програми. Якщо пам'яті для наступного об'єкта не вистачає, виникає виняткова ситуація, яку необхідно опрацювати. Цим мова *Java* відрізняється від мови *C++*, де у випадку невдачі оператор *new* повертає *null*.

Присвоєння значень змінним-посиланням на об'єкти. Після виконання фрагмента програми

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

b1 і *b2* посилаються на один і той же об'єкт в пам'яті. Внаслідок присвоєння *b2=b1* не буде зроблено копії об'єкта *b1*. Тобто внаслідок присвоєння значення однієї змінної-посилання на об'єкт іншій змінній- посиланню створюється копія посилання, а не копія цього об'єкта. Для копіювання об'єктів існують інші засоби.

3.2.2. Методи класу

Класи сформовано переважно з двох частин: змінних і методів. Кожний з методів за своїми функціями можна зачислити до однієї з семи категорій:

1. Конструктори. Викликаються для створення екземплярів об'єктів деякого класу.
2. Деструктори. Викликаються, коли робота з екземпляром об'єкта закінчена.
3. Копіювальники. Використовуються для копіювання екземпляра об'єкта в інший.
4. Set-методи. Викликаються з метою присвоєння значення змінній класу.
5. Get-методи. Зчитування значення змінної класу.

6. Методи введення-виведення. Викликаються для виконання взаємодії з зовнішніми пристроями.
7. Методи, специфічні для області визначення застосування.

Чимало експертів з програмування вважає, що тіло методу повинно містити до 20-ти стрічок або менше, тобто приблизно одну сторінку екранного тексту. В цьому випадку метод можна прочитати і зрозуміти з першого разу.

Методи в класі Box. Розширимо визначення вже згадуваного класу *Box*, додавши два методи:

```
class Box {
        double width;
        double height;
        double depth;
        // метод обчислення об'єму
        double volume( ){
                return width*height*depth;
                        } // завершення методу volume
        // метод задання розмірів коробки
        void setDim(double w, double h, double d){
                width=w;
                height=h;
                depth=d;
                        } // завершення методу setDim
} // завершення визначення класу
```

Декілька зауважень до тексту програми:

1) у методах класу посилання на змінні класу відбувається безпосередньо без посилання на об'єкт;
2) у методі *SetDim* використані параметри (*параметр* – це визначена в методі змінна, якій внаслідок виклику методу присвоюється значення, яке називають *аргументом).*
3) коректно розроблена програма – це програма, в якій доступ до змінних класу відбувається через методи класу (у нашому випадку реалізовано лише присвоєння цим змінним значень у методі *SetDim()*).

Передавання параметрів у мові Java. Передавання аргументів у методи здійснюється двома способами. Прості типи в мові *Java* (як і в більшості мов) передаються за значеннями (*call-by-*

value), тобто у формальний параметр копіюється значення аргументу. Все, що відбувається з параметром, не впливає на аргумент.

Об'єкти передаються за посиланням (*call-by-reference*). Тобто параметрові присвоюється значення посилання на об'єкт. Обидва посилання − і аргумент, і параметр − вказують на один і той же об'єкт у пам'яті. Тому всі зміни, які відбулися в методі, впливають на об'єкт, переданий як аргумент.

Використання об'єктів у ролі параметрів. В усіх попередніх прикладах у ролі параметрів методів передавалися прості типи. Проте в метод класу в ролі параметра можна передати об'єкт будь-якого типу. Розглянемо програму:

```
clas Test {
        int a, b;
        Test (int i, int j){ // конструктор класу
                a=i;
                b=j;
        }
// метод, який повертає true, якщо об'єкт 0 рівний тому,
// для якого викликається цей метод
        boolean eguals (Test 0){
                if (0.a==a && 0.b==b) return true;
                        else return false;
                }
} // завершення класу Test
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));
       }
}// завершення класу Passob
```

Результатом виконання цієї програми буде:

```
ob1==ob2: true
ob1==ob3: false.
```

Цей приклад засвідчує, що параметром може бути не тільки будь-який вбудований тип, але й клас, для якого цей метод визначається.

Особливо зручно робити це в конструкторах для створення точної копії вже існуючого об'єкта. Наприклад, у класі *Box* можна додати ще один варіант конструктора:

```
// створення дубліката об'єкта
Box (Box ob) {
        widht=ob.width;
        height =ob.height;
        depth=ob.depth;
        }
```

Повернення об'єктів методами. Метод може повертати не тільки прості типи даних, але і значення, описане як клас. Розглянемо програму

```
// Приклад повернення об'єкта з методу
class Test { int a;
        Test(int i ){   a=i; }
        // Цей метод повертає об'єкт типу Test,
        // в якому a збільшене на 10
        Test incrByten( ) {
                Test temp = new Test(a+10);
                return temp; }
        }// завершення класу Test
    class Retob {
        public static void main (String args[ ]){
                Test ob1=new Test(2),  ob2;
                ob2=0b1.incrByten( );
                ob2=0b2.incrByten( );
                System.out.println ("ob1.a= "+ob1.a);
                System.out.println ("ob2.a= "+ob2.a);
                System.out.println ("ob2.a= "+ob2.a);
        }
    }
```

Результатом виконання цієї програми будуть значення:

```
ob1.a=2
ob2.a=12
ob2.a=22.
```

Отже об'єкт *ob2* створюється не викликом безпосередньо конструктора класу, а методу *incrByten*, який повертає об'єкт. Хоча оператор *new* виокремив пам'ять у методі, який закінчив свою роботу, об'єкт буде існувати до того часу, доки в програмі буде хоча б одне посилання на нього.

Рекурсія. У мові *Java* підтримується механізм рекурсії (*recursion*), за допомогою якого метод може звертатися до самого себе. Такий метод називають рекурсивним.

Класичним прикладом рекурсії є обчислення факторіала:

```
// Приклад рекурсивного методу
class Factorial {
      int fact(int n){
            int result;
            if (n==1) return 1;
            // метод викликає сам себе
            result = fact(n-1)*n; return result;
            } // завершення методу fact
} // завершення класу Factorial
class Recursion {
      public static void main(String args[ ]) {
            Factorial f = new Factorial( );
            System.out.println ("Факторіал 4=" +f.fact(4));
            }
      }
```

При рекурсивному викликові методу в стек поміщаються нові локальні змінні і параметри і код методу починає виконуватися з початку із новими змінними. При рекурсивному виклику не створюється нова копія методу, новими стають тільки аргументи. При обчисленні факторіалу звертання відбувається доти, поки не буде повернена одиниця, яка в оберненому порядку помножиться на значення *n* (для кожного виклику своє).

При написанні рекурсивних методів треба використовувати хоча б один оператор *if*, щоб метод завершив роботу.

Рекурсивні версії багатьох програм можуть працювати довше, ніж їх ітераційні еквіваленти, через затрати на додаткові виклики функції. Надзвичайно велика кількість викликів методу може стати причиною переповнення стеку, в якому зберігаються

параметри і локальні змінні методу. Але є думка, що мислити в термінах рекурсії простіше, ніж в термінах ітерацій.

3.2.3. Конструктори

При створенні чергового екземпляра класу проводити ініціалізацію усіх його змінних є клопітно, можна щось випадково упустити. Для того в класах існують спеціальні методи, які називають конструкторами. Їхнє призначення полягає в ініціалізації внутрішнього стану об'єкта так, щоб внаслідок виконання оператора *new()* ми отримували цілковито завершений щодо користування об'єкт.

Конструктор (*constructor*) здійснює ініціалізацію об'єкта при його створенні. Ім'я конструктора завжди збігається з іменем класу, а за синтаксисом він нагадує метод. При написанні конструктора явно не вказується тип значення, яке він повертає, зокрема і *void*. Насправді, він повертає об'єкт типу, який визначено класом, конструктором якого він є.

Якщо конструктор класу не описано явно, *Java* створює для цього класу конструктор за замовчуванням. Конструктор за замовчуванням присвоює змінним класу нульові значення. Для простих класів цього буває достатньо. Якщо при визначенні класу є явно описаний конструктор, то при створенні екземпляра класу конструктор за замовчуванням не викликається: не можна викликати конструктор класу без параметрів, попередньо не написавши його.

У конструктор можна передавати параметри. У визначення класу *Box* (п. 3.2.1) можна додати конструктор:

```
// конструктор класу Box
Box(double w, double h, double d){
        width = w;
        height=h;
        depth=d;  }
// демонстрація виклику конструктора з параметрами
class BoxDemo{
        public static void main(string args[ ]){
                Box mybox = new Box(10, 20, 15);
                double vol = mybox.volume( ); } }
```

3.2.4. Перевантаження методів

Одним із способів реалізації поліморфізму є перевантаження методів (*metod overloading*). У мові *Java* (як і в інших мовах) можна в одному класі визначати декілька методів з одним іменем, однак різною кількістю параметрів. Ці методи називають перевантаженими (*overloaded*).

При перевантаженні методу, щоб визначити, яку версію методу треба виконати, використовується кількість і (або) тип аргументів. Типи значень, які повертає метод, можуть бути різними і вони не використовуються як ознака, за якою розрізняють дві версії методу. Коли компілятор *Java* зустрічає виклик перевантаженого методу, виконується та версія, параметри якого відповідають заданим під час виклику аргументам.

```
// Демонстрація перевантаження методів test()
class OverloadDemo {
        // метод без параметрів
        void test() {
                System.out.println("Без параметрів");
                        }
        // метод з одним цілочисельним параметром
        void test(int a)  {
                System.out.println ("Ціле число a="+a);
                        }
        // метод з двома цілочисельними параметрами
        void test(int a, int b)  {
            System.out.println ("Цілі числа a і b"+a+","+b);
                        }
        // метод з параметрам типу double
        double test(double a)  {
                System.out.println ("double a="+a);
                return a*a;
                        }
        } // завершення визначення класу OverloadDemo
class Overload {
        public static void main(String args[ ] ){
                OverloadDemo  ob=new OverloadDemo( );
                double result;
```

```
        // виклик усіх версій методу test
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result=ob.test(123.2);
        System.out.println ("Результат="+result);
        }
}
```

Якщо типи параметрів не збігаються з параметрами хоча б одного з методів, то відбувається автоматичне перетворення типів. Заберемо в попередньому прикладі у визначенні класу *OverloadDemo* версію методу *test(int a)*. Тоді при звертанні *ob.test(10)* відбудеться перетворення 10 в *double* і виклик методу *test(double a)*. При наявності необхідного методу перетворення типів не відбувається і звертання здійснюється до відповідного методу без перетворення типів.

Без реалізації перевантажених методів кожний метод повинен був би мати унікальне ім'я. У мовах програмування, які не допускали перевантаження методів (наприклад, *Fortran, C*), для обчислення усіх математичних функцій визначено свої назви для аргументів цілого типу, дійсного, з подвійною точністю. Функція *abs()* - повертала модуль цілого числа, *labs()* – цілого числа подвійної довжини, *fabs()* – дійсного числа і т. п.

Цінність перевантаження методів полягає у можливості присвоїти одне ім'я спорідненим методом. Ім'я у цьому випадку вказує на спільну дію, яка не залежить від типів аргументів. Вибір конкретної версії методу здійснює компілятор.

Унаслідок перевантаження методів кожна його версія може виконувати будь-які дії. Немає правила, яке б вимагало, щоб перевантажені методи мали виконувати подібні функції. Проте з погляду стилю ООП, під час перевантаження методів вже зрозуміло, що це реалізація поліморфізму і виконуються споріднені операції.

3.2.5. Перевантаження конструкторів

У мові *Java* перевантаження конструкторів – це норма, що й розглянемо на прикладі класу *Box* (п. 3.2.1*)*. Для цього класу було

визначено конструктор з трьома параметрами (розмірами коробки). Це означає, що кожне створення нового об'єкта типу *Box* вимагає задати певні розміри. Як бути у випадку, коли вони невідомі, або несуттєві при створенні нового об'єкта. Розв'язати ці очевидні ситуації при створенні об'єкта типу *Box* можна за допомогою перевантаження конструктора. У цьому випадку реалізація класу *Box* буде такою:

```
class Box {
        double width;
        double height;
        double depth;
        // конструктор, коли всі розміри задано
        Box (double w, double h, double d){
                width =w;
                height=h;
                depth=d;
                        }
        // конструктор, коли всі розміри не задано
        Box( ) {
                width =-1;
                height=-1;
                depth=-1;
                }
        // конструктор, який створює куб
        Box(double len) {
                width = height = depth = len;
                                }
        // метод обчислення об'єму коробки
        double volume( ) {
                return width*height* depth;
                }
} // завершення визначення класу Box
class OverloadCons {
        public static void main(String args [ ]){
        // створення об'єктів Box
        // за допомогою різних конструкторів
                Box mybox1=new Box(10,20,15);
                Box mybox2=new Box( );
```

```
        Box mycube=new Box(7);
        double vol;
        vol = mybox1.volume( );
        }
}
```

3.2.6. Основи керування доступом

Інкапсуляція – це не тільки зв'язування даних з кодом, який ними маніпулює, але й керування доступом (*accesscontrol*). Завдяки інкапсуляції можна дозволяти або забороняти доступ до елементів класу.

Спосіб доступу до елементів класу можна задати за допомогою специфікатора доступу (*access control*). У мові *Java* підтримується декілька специфікаторів доступу. Деякі з них призначені переважно для забезпечення спадковості або роботи з пакетами (набір групи класів). Ці механізми керування доступом розглянемо далі.

У мові *Java* є такі специфікатори доступу: ***public, private і protected.*** Існує поняття доступу, задане за замовчуванням. Член класу, визначений як ***public,*** є доступним з довільного місця програми, а член класу, визначений як ***private,*** –тільки для членів даного класу. Якщо специфікатор не задано, елемент класу вважається доступним в рамках його пакета, але не ззовні пакета.

Наведемо приклад класу, доступ до змінних якого здійснюється тільки через методи. Для оголошення змінної загальнодоступною мають бути дуже вагомі причини. Дана програма реалізує цілочисельний стек:

```
class Stack {
    private int stck [ ]=new int [10];
    private int tos;
    // Ініціалізація вершини стека
    Stack( ){
        tos=-1;
        }
    // метод, який додає елемент у стек
    void push(int item) {
        if (tos==9)
```

```
            System.out.println ("Стек повний");
        else
            stck[++tos]=item;
            }
// метод, який добуває елемент зі стека
int pop( ) {
  if (tos<0) {
     System.out.println ("Вихід за нижню границю");
     return 0; }
        else
            return stck[tos--];
      }
}
```

3.2.7. Поняття статичних даних

Бувають випадки, коли необхідно визначити член класу (змінну або метод), призначені для використання без створених конкретних об'єктів класу. Для створення такого члена класу необхідно перед його оголошенням поставити ключове слово ***static***. До таких членів класу можна звертатися ще до створення хоча б одного екземпляра (об'єкта) класу. Статичними можуть бути як змінні, так і методи.

Екземпляри змінних, оголошених як ***static***, є, за змістом, глобальними. При створенні об'єктів даного класу змінні не копіюються. Усі екземпляри класу користуються однією і тією ж статичною змінною.

На методи, оголошені як ***static***, накладено такі обмеження:

- вони викликають тільки статичні методи;
- вони використовують тільки статичні змінні;
- вони не можуть посилатися на ***this*** і ***super***.

Виклик статичного методу класу здійснюють так:

```
ім'я_класу.ім'я_методу( );
```

Якщо для ініціалізації статичних змінних необхідно виконати певні обчислення, можна оголосити блок як ***static,*** який виконається тільки один раз − під час першого завантаження класу.

```
// Демонстрація статичних змінних, методів і блоків
class UseStatic {
```

```
static int a=3;
static int b;
static void meth(int x){
        System.out.println ("x"+x);
        System.out.println ("a"="+a);
        System.out.println ("b"="+b);
                }
static {
    System.out.println ("Статичний блок ініціалізації");
    b=a*4; }
public static void main (String args[ ]){
                        meth(42); }
}
```

За допомогою статичних членів класу в мові *Java* реалізовано аналоги глобальних функцій і глобальних змінних.

3.2.8. Збирання сміття. Метод **finalize( )**

Зважаючи на те, що об'єкти розміщуються в пам'яті динамічно за допомогою оператора ***new***, виникає питання: як вони знищуються і як звільняється у цьому випадку пам'ять. У мові *C++* це здійснюють оператором ***delete***. У мові *Java* звільнення пам'яті відбувається автоматично. Цю технологію називають збиранням сміття (*garbage collection*). Періодично виконавча система *Java* переглядає, чи є посилання на об'єкт. Якщо немає, то об'єкт знищують і пам'ять вивільняється. Однак збирання сміття може і не здійснюватися, якщо програма навіть не наближається до критичного використання ресурсів. У таких випадках пам'ять повертається в операційну систему з завершенням роботи програми. При різних запусках однієї і тієї ж програми мовою *Java* збирання сміття відбувається неідентично.

Трапляються випадки, коли перед знищенням об'єкта необхідно виконати певні дії: закрити файл і т. п. З цією метою в *Java* розроблений спеціальний механізм, який називають фіналізацією (*finalization*) і реалізують у методі ***finalize( )***:

```
protected void finalize( ){
    //дії, які необхідно виконати перед знищенням об'єкта
    }
```

Збирання сміття в *Java* відбувається періодично або не відбувається зовсім, тому не можна передбачити, коли спрацює метод *finalize()*. Для нормального режиму роботи програми покладатися на метод *finalize()* не можна. Насправді його використовують з метою стеження за процесом збирання сміття (за умови, що воно відбувається).

Для запуску позачергового процесу збирання сміття використовують виклик спеціального методу класу *System:*

```
System.gc();
```

Процес збирання сміття в мові *Java* впливає також на швидкість виділення пам'яті для нових об'єктів. Віртуальна машина *Java* (*JVM*) за часом розміщення об'єкта в пам'яті "купі" майже досягає рівня виділення пам'яті стека в інших мовах програмування.

3.3. Ієрархія класів

3.3.1. Основи наслідування

Однією з найбільших переваг ООП є багаторазове використання програмного коду. Тобто використання вже існуючих класів. На відміну від процедурного програмування програмний код не копіюють і змінюють, а використовують без змін. Є два способи реалізації цієї ідеї. Перший полягає у створенні всередині нового класу об'єктів існуючих класів. Такий підхід називають композицією. Новий клас є набором об'єктів і фактично використовує функціональність існуючого коду, а не його структуру. Другий підхід полягає в використанні структури існуючого класу, до якого додають новий код, незмінюючи існуючого. Насправді це спадковість (наслідування, *inheritance*), одна зі складових ОПП. Завдяки спадковості будується ієрархія класів. Клас, який наслідують, називають базовим класом (*superclass*). Підклас – це спеціалізована версія базового класу. Базовий клас і підклас абсолютно самостійні класи. Підклас може бути для іншого класу базовим класом.

У мові *Java* оголошення класу, який є спадкоємцем базового класу, матиме вигляд

```
class ім'я_підкласу extends ім'я_суперкласу  {
        // тіло класу
    }
```

Тобто у визначення класу необхідно внести дані щодо імені базового класу з ключовим словом ***extends***. Навіть тоді, коли внаслідок створення нового класу явно не вказують клас наслідування, автоматично відбувається наслідування від кореневого класу *Java* з іменем ***Object*** (так і було в усіх попередньо наведених прикладах). Наступна програма демонструє приклад створення одного класу на базі іншого:

```
class A {
        int i,j;
        void showij( ){
                System.out.println ("i ma j:" +i +", "+j);
                }
} // завершення класу A
//  створення підкласу
class B extends A {
        int k;
        void showk( ){
                System.out.println ("k:" +k);}
        void sum( ){
                System.out.println ("i + j + k:" +(i + j + k));  }
} // завершення класу B
class SimpleInheritance {
        public static void main(String args[ ]){
                A superOb=new A( );
                B subOb=new B( );
                // використання базового класу безпосередньо
                superOb.i = 10;
                superOb.j = 20;
                superOb.showij();
                // підклас має доступ до public членів базового класу
                subOb.i=7;
                subOb.j=8;
                subOb.k=9;
                subOb.showij( );
                subOb.showk( );
                subOb.sum( );
                }
 }
```

Зауважимо, що у наведеному вище прикладі усі члени класу *A* оголошено відкритими (без специфікаторів доступу) у рамках цього пакета. Якщо створювати підклас класу *A* в іншому пакеті, то він не матиме доступу ні до його змінних, ні до методу ***showij( )***. Доступ із підкласів завжди є до ***public*** членів базового класу, а також до захищених (***protected***) членів базового класу. Член класу, оголошений як ***private***, доступний тільки в самому класі.

Якщо розглянути клас *B*, який є підкласом *A*, і клас *C*, який є підкласом *B* і так далі, то утвориться багаторівнева ієрархія класів, яка є нормальним явищем для усіх ОПП мов програмування. Кожний підклас наслідує всі властивості своїх базових класів.

У мові *Java* не підтримується багаточисельне наслідування. Тобто один підклас матиме тільки один базовий клас. У цьому мова *Java* відрізняється від *C++*. Ні один клас не може бути базовим класом для самого себе.

Прикладом розширення класу *Box* може бути врахування ваги коробки: ***weight*** (вага). Новий підклас матиме довжину, ширину, висоту і вагу.

```
// Клас, який є підкласом Box
class BoxWeight extends Box {
        double weight;
        // конструктор BoxWeight
        BoxWeight(double w, double h, double d, double m){
                width=w;
                height=h;
                depth=d;
                weight=m;
                }
} // завершення класу BoxWeight
class DemoBoxWeight {
   public static void main(String args[ ]){
      BoxWeight mybox= new BoxWeight(10,20,15,34.3);
      double vol;
      vol=mybox1.volume( );
      System.out.println ("Об'єм:" +vol );
      }
}
```

Як видно з прикладу, підклас *BoxWeight* наслідував усі характеристики класу *Box* (метод *volume(),* змінні *width, height, depth*) і додав вагу. У класі *BoxWeight* не було необхідності повторювати все, що було у класі *Box*.

Змінна, яка є посиланням на базовий клас, може посилатися на об'єкт підкласу, утворений від цього базового класу. Проте у цьому випадку вона матиме доступ тільки до елементів, визначених у базовому класі. Це цілком логічно, адже базовому класу нічого невідомо про те, що додано в підкласах.

Ключове слово super. Клас *BoxWeight,* розроблений у наведеному прикладі, явно не ефективний, особливо його конструктор. По-перше, він повторює конструктор базового класу, а, по-друге, має доступ до елементів базового класу. Така реалізація порушує принцип ОПП, а саме інкапсуляцію. Для нормального виходу з цієї ситуації у мові *Java* передбачено ключове слово *super*. Його використовують у двох випадках: 1) з метою виклику конструктора базового класу; 2) з метою доступу до членів базового класу, перевизначених такими ж членами підкласу.

Підклас може викликати конструктор базового класу за формою:

```
super (список_параметрів);
```

До списку параметрів належать усі параметри, необхідні конструкторові базового класу. Оператор *super()* буде першим оператором, який виконується в конструкторі підкласу. Якщо навіть його там немає, то викликають заданий за замовчуванням або конструктор без параметрів базового класу. Це цілком логічно, оскільки все, що необхідно задати в базовому класі, задають перед ініціалізацією підкласу.

Дещо краща версія класу *BoxWeight* може бути такою:

```
// використання оператора super( )
class BoxWeight extends Box {
        double weight;
        BoxWeight( double w, double h, double d, double m){
                // виклик конструктора базового класу
                super(w, h, d);
                weight= m;
        }
}
```

За такої реалізації змінні базового класу будуть проініціалізованими навіть за умови, що вони оголошені як *private*.

Якщо в базовому класі конструктори перевантажені, виклик оператора *super()* можна здійснити у будь-якій формі, визначеній у базовому класі. Особливу увагу звернемо на конструктор створення дубліката об'єкта:

```
BoxWeight (BoxWeight ob){
        super(ob);
        weight =ob.weight;
}
```

Тут оператор *super* викликаний з параметром *BoxWeight*, а не *Box*. Однак буде викликаний конструктор базового класу *Box(Box ob)*. Як уже зазначено, змінна типу базового класу може посилатися на об'єкт, який походить від цього класу. Проте у цьому випадку класові *Box* відомі тільки про його власні змінні.

Під час виклику *super()* виконується конструктор базового класу, який в ієрархії перебуває безпосередньо на один рівень вище від класу, який його викликає.

Для звертання до елементів базового класу (методів і екземплярів змінних) використовують форму:

```
super.елемент;
```

Форма *super* найприйнятніша у випадках, коли для членів базового класу використано ті ж імена, що й для членів підкласу. Наприклад:

```
class A { int i;  }
class B extends A {
        int i; // ця змінна i перекриває i в A
        B(int a, int b){
                super.i=a; // i в класі A
                i=b; // i в класі B
                }
        void show( ){
                System.out.println ("i в базовому класі:"+super.i);
                System.out.println ("i в підкласі:"+i);
                }
}// завершення класу B
```

```
class UseSuper {
        public static void main(String args[ ]){
                B subOb=new B(1, 2);
                subOb.show ();
                        }
                }
```

За допомогою *super* можна також викликати методи, перекриті методами підкласу.

Використання ключового слова final під час наслідування. Ключове слово *final* має декілька значень, однак основне з них можна виділити словами: "Цього не можна змінювати". Використовувати *final* можна для змінних, методів і класів.

У п. 3.2.1 зазначено, що додавання до оголошення змінної класу слова *final* вказує компілятору, що вона є незмінною, тобто константою. Переважно це константи простих типів. При використанні слова *final* перед посиланням на об'єкти фактично стає незмінним тільки посилання, а не сам об'єкт. У цьому випадку посилання вже не зможе вказувати на інший об'єкт, а сам об'єкт можна змінювати. Це обмеження належить і до масивів, які в мові Java є об'єктами. Змінні класу, оголошені як *final,* необхідно визначати або під час їхнього оголошення, або в кожному з існуючих у класі конструкторі. Так можна гарантувати визначення змінних *final* перед їхнім використанням.

У мові *Java* дозволено оголошувати зі словом *final* параметри методів. Тоді їх не можна змінювати в методі.

Якщо перед оголошенням методу додати слово *final,* то такий метод стає незмінним. Це використовують тоді, коли є небажаним перекриття методу в підкласах. Наприклад,

```
class A {
  final void meth( ) {
    System.out.println("Цей метод не можна перекривати");
                }
        }
```

Будь-яка спроба перекрити метод, оголошений як *final,* дає помилку під час компіляції.

З іншого боку, методи, оголошені як *final*, дають змогу покращити реалізацію програми. Компілятор організовує звертання

до них як до виклику вбудованого коду (фактично заміняє виклик методу на копію реального коду, який є в цьому методі), тому що вони не будуть перекриватися в підкласі. Отже, ліквідовуються накладні витрати, спричинені викликом звичайного методу.

У мові *Java* (як і в інших мовах) використовують методику пізнього зв'язування (*late binding*), тобто рішення, який саме метод буде викликано, приймається під час виконання програми. Проте *final*-методи не можуть бути перекриті, тому питання, пов'язані з їхнім викликом, вирішується на стадії компіляції. Цю методику називають раннім зв'язуванням (*early binding*).

З метою заборони створення підкласів деякого класу використовують також ключове слово *final*, яке ставлять на початку визначення класу. Оголошення класу як *final* неявно робить усі його методи також *final* (без можливості перекриття).

Очевидно, що не можна оголосити клас *final* і *abstract* одночасно, тому що клас. оголошений як *abstract* вимагає підкласів для його цілковитої реалізації.

3.3.2. Поліморфізм

Перекривання методів. Якщо в ієрархії класів ім'я, типи і кількість параметрів методу підкласу збігаються з іменем, типом і кількістю параметрів методу базового класу, то вважають, що метод підкласу перекриває (*override*) метод базового класу. Звертання до такого перекритого методу завжди викликає метод підкласу. З метою виклику перекритого методу базового класу необхідно використати ключове слово *super.ім'я_методу*.

Зауважимо, що перекривання методів відбувається тільки тоді, коли цілковито збігаються ім'я, типи і кількість параметрів. Інакше це зводиться до перевантаження методів. Тобто для підкласів викликають або його метод, або метод базового класу. Залежно від того, які параметри задано під час виклику.

Динамічна диспетчеризація методів. Перекривання методів є базою одного з найважливіших принципів мови *Java* – динамічної диспетчеризації методів (*dynamic method dicpatch*). Це механізм, який дає змогу викликати перекриті методи під час виконання програми, а не під час компіляції. Фактично – це забезпечення поліморфізму на етапі виконання програми.

Згадаємо важливий принцип: змінна, яка вказує (посилається) на об'єкт базового класу, може також вказувати на об'єкт підкласу. Цей принцип використовують у мові *Java* для прийняття рішення під час виконання програми. Якщо перекритий метод викликається за допомогою посилання на базовий клас, *Java* визначає, який метод викликати залежно від типу об'єкта, на який вказує це посилання. Іншими словами, визначаючим параметром під час вибору версії перекритого методу є тип об'єкта, на який у змінній є посилання, а не тип самої змінної. Отже, якщо в базовому класі визначено метод, який надалі перекрито методом у підкласі, то внаслідок посилання змінної базового класу на об'єкти різного типу виконуватимуться різні версії цього методу. Розглянемо це на прикладі:

```
class A {
    void callme( ){ System.out.println ("метод callme класу A"); }
}
class B extends A {
   void callme( ){ System.out.println ("метод callme класу B");  }
}
class C extends A {
  void callme( ){  System.out.println ("метод callme класу C"); }
}
class Dispatch {
        public static void main(String args[ ]){
                A a=new A( );
                B b=new B( );
                C c=new C( );
                A r; // посиляння на тип A
                r=a;
                r.callme( ); // виклик методу для об'єкта класу A
                r=b;
                r.callme( ); // виклик для об'єкта класу B
                r=c;
                r.callme( ); // виклик для об'єкта класу C
                }
}
```

У цьому прикладі змінну *r* визначено як змінну класу *A* – базового класу для класів *B,C*. Як бачимо, викликається версія

методу *callme()*, визначена типом об'єкта. Якби все визначалося типом посилання *r*, то завжди викликався б метод *callme()* класу *A*.

Перекриття методів дає змогу визначити в базовому класі усі спільні для необхідних класів методи, а в підкласах визначити специфічні шляхи їхньої реалізації. Це ще один прояв поліморфізму – один інтерфейс, декілька методів.

Завдяки комбінації наслідування і перекриття методів базовий клас визначає загальний вигляд методів, які надалі використовують усі його підкласи.

3.3.3. Абстрактні класи

Визначаючи загальний клас, який пізніше є базовим класом для інших підкласів, не можна до кінця визначити усі деталі його методів, а взагалі і не потрібно. У загальному класі вказується тільки природа методів, а конкретні деталі задаються безпосередньо у підкласах. Тобто створення базового класу з методами, які не виконують ніяких дій, насправді є створенням абстрактного класу. Методи, які в базовому класі фактично є "заглушками", необхідно перекрити у підкласах. Для позначення таких методів використовують ключове слово *abstract:*

abstract тип ім'я_методу(список_параметрів);

Абстрактні методи (*abstract method*) не мають коду (тіла). Їхня реалізація передається на розгляд підкласу.

Якщо клас містить хоча б один абстрактний метод, його необхідно оголосити як абстрактний за допомогою слова *abstract* перед словом *class*. Для абстрактного класу не можна створювати об'єктів. Не можна оголошувати абстрактними конструктори та статичні методи.

Якщо підклас абстрактного базового класу не реалізує всіх абстрактних методів базового класу, то його також необхідно оголосити абстрактним. В абстрактних класах поряд з абстрактними методами існують і звичайні, цілковито реалізовані методи.

Абстрактні класи використовують для створення посилань з типом цих класів для реалізації динамічного поліморфізму. Ці посилання пізніше використовують з метою прив'язки до об'єктів підкласів. Наведемо приклад програми з абстрактним класом:

```
// Використання абстрактних методів і класів
abstract class Figure {
        double dim1, dim2;
        Figure (double a; double b){ // конструктор класу
                dim1= a;
                dim2= b;
                }
        abstract double area( ); // абстрактний метод
}
class Rectangle extends Figure { //прямокутник
        Rectangle (double a; double b){
                super (a, b);
                }
        double area( ) {
                return dim1 * dim2;
                }
}
class Triangle extends Figure { //трикутник
        Triangle (double a; double b){
                super (a, b);
                }
        double area( ) {
                return dim1 * dim2/2;
                }
}
class Abstract Areas {
     public static void main(String args[ ]){
        Rectangle r=new Rectangle(9,5);
        Triangle t=new Triangle(10,8);
        Figure figref ; // оголошення посилання
                        // без створення об'єкта
       figref = r;
      System.out.println("Площа прямокутника:"+figref.area( ));
       figref = t;
       System.out.println("Площа трикутника:"+figref.area( ));
       }
}
```

3.3.4. Інтерфейси

Як зазначено у п. 3.3.3, використання абстрактного класу дає змогу частину його методів не реалізовувати у програмному коді, а лише означити поведінку. Цілковито абстрагувати інтерфейс класу від його реалізації можна за допомогою ключового слова *interfase*. Іншими словами, інтерфейс дає змогу визначити, що повинен робити клас, а не те, як він це має робити.

За синтаксисом інтерфейси подібні до класів, однак вони не мають екземплярів змінних, а оголошені методи не мають коду їхньої реалізації. На практиці це означає, що можна створювати інтерфейси, не знаючи, як вони будуть реалізовані.

Якщо інтерфейс визначений, його можна реалізувати в багатьох класах. І навпаки, один клас може реалізувати декілька інтерфейсів. Отже, у мові *Java* реалізовано множинне наслідування. При реалізації інтерфейсу в класі повинні бути реалізовані всі методи, визначені в інтерфейсі. Проте клас вільний у виборі способу реалізації інтерфейсу.

Призначення інтерфейсу – це забезпечення динамічного вибору під час виконання програми. Якщо об'єкт одного класу викликає метод другого класу, то присутність обидвох цих класів необхідна на етапі компіляції з метою перевірки сумісності параметрів. За умови значної ієрархії класів це зумовлює проблеми для вирішення яких призначені інтерфейси. Вони переривають зв'язок між визначенням методів і їхньою реалізацією. Ієрархія інтерфейсів відрізняється від ієрархії класів. Класи, які не зв'язані між собою ієрархічними зв'язками, можуть реалізувати один і той же інтерфейс.

Визначення інтерфейсу. Загальний вигляд визначення інтерфейсу:

```
доступ interfase ім'я {
        тип ім'я методу 1 (список_параметрів);
        тип ім'я методу 2 (список_параметрів);
        тип змінна_константа 1=значення;
        тип змінна_константа 2=значення;
        //…
        тип ім'я методу N (список_параметрів);
        тип змінна_константа N=значення; }
```

На місці *доступ* вказується *public* (у цьому випадку ім'я інтерфейсу має збігатися з іменем файла, в якому його визначено), або нічого. Якщо не вказано нічого, то інтерфейс доступний тільки в пакеті, де його визначено. Методи інтерфейсу – це, насправді, абстрактні методи. Змінні, оголошені в інтерфейсі, є фактично *final і static*. Їх необхідно проініціалізувати. Інтерфейс, оголошений як *public,* має усі змінні і методи *public*.

Реалізація інтерфейсу. Реалізацію інтерфейсу здійснюють в одному або декількох класах. Для цього у визначення класу долучають оператор *implements* і створюють методи, оголошені в інтерфейсі. Загальний вигляд класу, який реалізує інтерфейс, можна подати так:

```
доступ class ім'я_класу [extends суперклас]
                implements інтерфейс [, інтерфейс] {
                //тіло класу
        }
```

Доступ – це *public* (або пусто). Один клас може реалізовувати декілька інтерфейсів. Тоді їхні імена йтимуть за ключовим словом *implements* одне за одним через кому.

Якщо клас реалізує два інтерфейси, в яких визначено метод з одним і тим же іменем і списком параметрів, то цей реалізований метод можна використовувати з посиланнями обох інтерфейсів.

Методи, оголошені в інтерфейсі, у класі необхідно оголосити як *public*. Список параметрів методу в класі точно має збігатися зі списком параметрів, заданих у визначенні інтерфейсу. У класах, які реалізують інтерфейси, можуть бути свої додаткові члени: змінні і методи.

Якщо до визначення класу зачислено інтерфейс, однак не всі методи цього інтерфейсу реалізовано у цьому класі, то цей клас необхідно оголосити абстрактним класом.

Розширення інтерфейсів. До кожного інтерфейсу можна додати оголошення нових методів або об'єднати декілька інтерфейсів в один, використовуючи наслідування. Як і в класах, один інтерфейс може наслідувати інший інтерфейс за допомогою ключового слова *extends*. Наприклад,

```
interfase B extends A [,C] {
                //тіло інтерфейсу
    }
```

У класі, який цілковито реалізує інтерфейс, необхідно реалізовувати усі методи, оголошені у повному ланцюгу інтерфейсів, які наслідують один одного.

Змінні інтерфейсу. Враховуючи те, що змінні інтерфейсу автоматично є статичними (*static*) і незмінними (*final*), інтерфейси можна використовувати для імпортування спільних констант у декілька класів. З цією метою необхідно описати та обов'язково проініціалізувати змінні в інтерфейсі. Внаслідок включення цього інтерфейсу в операторі *implements* усі імена змінних потраплять в область доступності класу як константи. Якщо ж цей інтерфейс не містить оголошення методів, то їх не потрібно реалізовувати.

Використання інтерфейсів. Аналогічно, як і посилання на клас, можна оголошувати змінні-посилання на інтерфейси. Така змінна може зберігати покажчик на будь-який екземпляр класу, який реалізує цей інтерфейс. Під час звертання до методу за посередництвом такої змінної-посилання вибір здійснюватиметься, виходячи з реального екземпляру, на який посилається ця змінна. Це одна з ключових властивостей інтерфейсу. Метод вибирається динамічно під час виконання програми. Це дає змогу компілювати класи навіть після програми, яка викликає їхні методи. Під час компіляції програми враховується звертання до інтерфейсу, а не до методу класу, який реалізує інтерфейс. Такий пошук методів під час виконання програми вимагає додаткового часу, тому зловживання інтерфейсами, коли важлива швидкодія, є небажаним.

3.3.5. Внутрішні класи

Визначення класу, цілковито розташованого в іншому класі, дає внутрішній клас (*inner class*). Очевидне, на перший погляд, використання внутрішніх класів для реалізації механізму інкапсуляції (захищення коду) не є головною метою їхнього призначення. Для цього в об'єктно-орієнтованих мовах (зокрема в *Java*) достатньо надати класові доступ за замовчуванням ("дружній") і він буде видимим тільки у пакеті, в якому його визначено. Головна роль внутрішнього класу − це можливість виконати перетворення вверх до базового класу або інтерфейсу (покажчик на об'єкт замінить покажчик на інтерфейс).

На відміну від звичайних класів (які можуть бути відкритими або мати доступ за замовчуванням), внутрішні класи можна оголошувати з ключовим словом ***private*** або ***protected.*** Це дає змогу програмістові цілком закрити деталі реалізації, а компілятору оптимізувати код для виконання.

Внутрішні класи можна визначати всередині методу або складеного оператора (блоку). Розглянемо приклад. Нехай у своєму файлі (*Destination.java*) визначено інтерфейс:

```
public interface Destination {
        String readLabel();
}
```

У файлі *Parcel4.java* розміщено програмний код:

```
// визначення класу в тілі методу
public class Parcel4 {
        public Destination dest(String s) { // початок методу
                // початок внутрішнього класу
                class PDestination implements Destination {
                        private String label;
                        private PDestination(String whereTo) {
                                label = whereTo;
                                }
                        // завершення конструктора внутрішнього
                        класу
                        public String readLabel()
                                // реалізація методу інтерфейсу
                                { return label; }
                        } // завершення внутрішнього класу
                        PDestination
                return new PDestination(s);
        } // завершення методу dest
        public static void main(String[ ] args) {
                Parcel4 p = new Parcel4();
                Destination d = p.dest("Танзанія");
                }
} // завершення класу Parcel4
```

У наведеному коді метод ***dest*** повертає посилання на інтерфейс ***Destination,*** однак чітко тип об’єкта визначити немож-

ливо (клас *PDestination* ззовні методу невідомий). Незважаючи на те, що клас *PDestination* оголошено у методі, в якому також створюється об'єкт цього класу, цей об'єкт буде доступним і після виходу з методу (через посилання на інтерфейс).

Внутрішні класи можуть бути безіменними. Розглянемо приклад створення об'єкта такого класу. Нехай у файлі *Contents.java* визначено інтерфейс:

```
public interface Contents {
        int value();
}
```

Розглянемо програмний код, розташований у файлі *Parcel6.java*:

```
// Метод повертає внутрішній клас без імені
public class Parcel6 {
        public Contents cont() {
                return new Contents() {
                                private int i = 11;
                                public int value() { return i; }
                        }; // Крапка з комою засвідчує
                           // про закінчення оператора
                } //завершення методу cont
        public static void main(String[ ] args) {
                Parcel6 p = new Parcel6();
                Contents c = p.cont();
                }
} // завершення класу
```

У методі *cont()* в операторі повернення значення (*return new Contents())* іде визначення класу, об'єкт якого повертається. Такий дивний, на перший погляд, запис означає, що створюється об'єкт неіменованого класу, який наслідує інтерфейс *Contents.* Фактично − це є скороченою формою запису наступного коду:

```
class MyContents() implements Contents {
                private int i = 11;
                public int value() { return i; }
        }
return new MyContents();
```

Об'єкт внутрішнього класу можна створити тільки разом з об'єктом зовнішнього класу, в якому його визначено. У попередніх

прикладах обов'язкова присутність операторів: *Parcel4 p = new Parcel4(); Destination d = p.dest("Танзанія"); Parcel6 p = new Parcel6(); Contents c = p.cont();,* у яких використано посилання *p* на об'єкт зовнішнього класу. Якщо об'єкт внутрішнього класу створюється не одним з методів (як в попередніх прикладах), а конструктором, то оператор *new* записують так:

Parcel11.Contents c=p.new Contents();

У цьому випадку *p* − посилання на об'єкт класу *Parcel11,* зовнішнього щодо класу *Contents.*

Внутрішній клас може звертатися до всіх змінних (зокрема *private)* і методів класу, в якому його визначено, так як до своїх власних. Фактично він містить покажчик (прихований, доступний через *this*) на об'єкт зовнішнього класу.

Якщо зв'язок між об'єктом внутрішнього і зовнішнього класів не обов'язковий, то можна оголосити внутрішній клас з ключовим словом *static*.

3.3.6. Відображення у програмах відношень між об'єктами

При побудові ієрархії класів між ними існують різні типи відносин,. що необхідно пам'ятати, розрізняти і відповідно відображати у програмі. Стандартними вважають такі відношення:

- є (*is a*) – папуга є птахом;
- має (*has a*) – папуга має крила;
- використовує (*uses a*) – папуга використовує сідало;
- створює (*creates a*) – папуга створює (несе) яйця.

Програмування відношення "є" між класами. Відношення "є" встановлюється між двома класами і вказує на те, що один з класів є особливим (спеціалізованим) типом іншого класу. Головним способом реалізації відношення "є" є створення підкласу. Наприклад,

public class Parrot extends Bird {…}

Характеристики (відкриті) класу *Bird* (птах) будуть і характеристиками класу *Parrot* (папуга).

Програмування відношення типу "має".

Відношення типу "має" інколи називають відношенням типу "є частиною" або "використовується для реалізації". Якщо класи *A* і *B*, з'єднані відношенням "має" і клас *B* використовується

декількома іншими класами, то його необхідно реалізувати у власному файлі класу, а в опис класу *A* розмістити покажчик на екземпляр класу *B*. Отже, якщо в проекті необхідно відобразити, що і птахи, і жуки, і кажани мають крила, то клас *B* треба оголосити так:

```
public class Bird {
        private Wing leftWing;
        private Wing rightWing;
…
}
```

Однак, якщо клас *A* цілковито інкапсулює методи і властивості класу *B*, тоді останній необхідно реалізувати як внутрішній клас класу *A*. У випадку, коли програмне застосування стосується тільки птахів і ніяким іншим об'єктам не потрібно доступу до класу *Wing* (крило), його реалізують у вигляді:

```
public class Bird {
        class Wing { ... }
        private Wing leftWing;
        private Wing rightWing;
        …  }
```

Програмування відношення "використовує". Відношення "використовує" реалізують шляхом передачі об'єкта одного класу в методи іншого класу. Наприклад, щоб дати змогу папузі відпочити, можна записати такі оператори:

```
Perch aPerch = new Perch();
theParrot.land (aPerch);
```

Щоб приземлитися (виклик методу *land*), папуга повинен знайти сідало (описане класом *Perch*). Об'єкт класу *Parrot* використовує об'єкт класу *Perch*.

Відношення типу "використовує" існує і в тому випадку, коли метод одного класу використовує функціональні можливості іншого класу.

Наприклад:

```
public class Parrot {   ...
        public boolean isSick(Veterinarian theVet)
                {   return theVet.check(this); }
        }
```

Відношення типу "створює". Найчастіше відношення встановлюються між двома класами (наприклад, "є"), або між двома екземплярами об'єктів ("використовує"). У цьому розумінні "створює" особливе, тому що встановлюється зв'язок між окремим екземпляром об'єкта і класом. Наприклад, дані, що папуга готується вивести потомство, можна записати так:

```
Egg layEgg(){
        Egg theEgg = new Egg();
        return theEgg; }
```

Тут клас *Parrot* (папуга) викликає клас *Egg* (яйце) для створення нового екземпляра класу (об'єкта *Egg*). Будь-який об'єкт, який тепер звернеться до даного екземпляра об'єкта *Parrot*, отримає доступ і до нового екземпляра об'єкта *Egg*.

3.3.7. Клас Object

Кожна об'єктно-орієнтована мова має вершину ієрархії − один базовий клас для всіх решта класів. У мові *Java* його назва *Object*. Це означає, що змінна, яка є посиланням на тип *Object*, може посилатися на об'єкт будь-якого класу, у тім числі масиви, які в мові *Java* є класами.

Методи, визначені в класі *Object,* наведено в табл. 3.1.

Т а б л и ц я 3.1. **Методи класу *Object***

Метод	Призначення
Object clone()	Створює копію об'єкта
boolean equals(Object ob)	Порівнює один об'єкт з іншим
void finalize()	Викликається перед знищенням об'єкта
Class getClass()	Визначає клас об'єкта
int hashCode()	Повертає хеш-код, зв'язаний з об'єктом
void notify()	Відновлює виконання потоку
void notifyAll()	Відновлення виконання всіх потоків
String toString()	Повертає рядок з описом об'єкту
void wait() void wait(long millisek) void wait(long millisek, int nanosek)	Очікує виконання іншого потоку

Методи *getClass(), notify(), notifyAll() і wait()* оголошені в класі *Object* як *final*. Інші методи можуть бути перекриті в підкласах. У нових класах бажано перекривати методи *equals()* і

toString() для врахування їх особливостей. Метод *toString()* викликається автоматично при виведенні інформації про об'єкт методом *println()*.

3.3.8. Пакети: сховище класів

У мові *Java* пакети (*packages*) відіграють таку ж роль, як бібліотеки в інших мовах програмування. Пакети – це контейнери, які використовуються для ізольованого зберігання споріднених класів з метою уникнути перетину імен класів. Пакети розташовуються ієрархічно і явно імпортуються в визначення нових класів. Розробники *Java* написали власний набір машинно-незалежних бібліотек класів, які зберігаються в пакетах і дають змогу не задумуватися на тим, в якій операційній системі виконуватиметься програма мовою *Java.*

Файл вихідних пакетів у мові *Java* може налічувати будь-які (або всі) перераховані нижче складові:

- один оператор пакета (не обов'язково);
- довільну кількість операторів імпорту (не обов'язково);
- одне оголошення загальнодоступного класу (обов'язково);
- довільну кількість *private*-класів у цьому пакеті (не обов'язково).

У розглянутих прикладах програм не було необхідності у використанні пакетів. Завжди існував тільки загальнодоступний клас, імена всіх класів належали одному і тому ж просторові імен ("пакетові за замовчуванням"). Через це кожному класові присвоювали унікальне ім'я, щоб уникнути конфлікту імен.

Створення пакета. За необхідності зазначити, що всі компоненти програми (записані в окремих файлах .java, .class) функціонально зв'язані між собою, бажано зачислити їх до одного пакета. З метою створення пакета необхідно в кожному файлі з вихідними текстами програми поставити першим оператор:

package ім'я_пакета;

Усі класи, оголошені в цих файлах, будуть зачислені до цього пакета. Якщо оператор *package* відсутній у файлі, то імена класів цього файла, за замовчуванням, належать до пакета без імені. Існує

домовленість, що для запису імен пакетів використовують лише малі літери.

Для зберігання пакетів у мові Java використовують систему імен каталогів. Наприклад, якщо створити пакет:

```
package mypackage;
```

то всі файли з розширенням .class, які належать до mypackage, повинні перебувати в директорії з іменем mypackage. Важливо, щоб ім'я директорії точно збігалося з іменем пакета.

Можна створювати ієрархію пакетів, задаючи оператор:

```
package pkg1 [.pkg2 [.pkg3]];
```

Вкладеність пакетів у мові *Java* обмежується тільки можливостями файлової системи. Ієрархія пакетів повинна відображатися в структурі директорій. Пакет, оголошений як

```
package java.awt.image;
```

необхідно зберігати в одній з директорій, які визначають інтерпретатором *Java* так: до директорій, перерахованих у CLASSPATH, додають java\awt\image (для *Windows*).

Під час вибору імені пакета треба бути уважним. Не можна змінити ім'я пакета, не змінивши назву директорії, в якій зберігаються його файли.

Імпортування пакетів. *У*сі стандартні класи *Java* зберігаються у пакетах, які мають відповідні імена. Звертання до класу з вкладених пакетів вимагає задати повний список імен пакетів, розділених крапками. Для ліквідації цієї незручності в мові *Java* існує оператор ***import***. Він дає змогу імпортувати класи і навіть цілі пакети. До імпортованого класу можна звертатися тільки за іменем.

У вихідному файлі оператори ***import*** ставлять безпосередньо за оператором ***package*** (якщо він є) і перед першим визначенням класу. Він має вигляд:

```
import pkg1 [.pkg2].(ім'я_класу |*);
```

Оператор ***import*** має закінчуватися іменем класу або символом "зірочка" ("*"), який вказує на те, що імпортувати необхідно весь пакет. Наприклад,

```
import java.util.Date;
import java.io.*;
```

На відміну від оператора ***#include*** мови *C++*, оператор ***import*** не пов'язує з програмою тих класів пакета, які в ній не

використовуються. Символ "*" впливає тільки на час компіляції, а на час виконання і розмір класу абсолютно не впливає.

Якщо у двох цілковито імпортованих пакетах є два класи з однаковими іменами, то компілятор видає помилку тільки за першого звертання до імені класу. У таких випадках необхідно задати ім'я класу з вкладенням у пакет. Наприклад, створення об'єкта типу *Vector* запишеться так:

```
java.util.Vector v = new java.util.Vector();
```

Мова *Java* має пакет, який автоматично імпортується компілятором у всі програми. Це пакет ***java.lang***, який містить 4 інтерфейси, 27 класів, 20 класів помилок і 24 класи виняткових ситуацій. Серед класів пакета ***java.lang*** є такі важливі класи як *Object, String, System*, класи-оболонки для простих типів (детальніше можна подивитися в документації).

Обмеження доступу. І класи, і пакети є засобами інкапсуляції. Пакети відіграють роль сховища для класів і вкладених пакетів. Класи – це сховище даних (змінних) і методів. Клас є найменшою одиницею абстракції в мові *Java*. Завдяки пакетам у мові *Java* з'явився ще один вимір в системі керування доступом.

У сфері взаємодії класів і пакетів виокремлюють чотири категорії доступності елементів класу:

- підкласи у тому ж пакеті;
- не підкласи у тому ж пакеті;
- підкласи у різних пакетах;
- класи, які не є підкласами і перебувають в інших пакетах.

Механізм керування доступом можна розглядати спрощено: все, що оголошено як ***public***, доступно всюди; все, що оголошено як ***private*** – за межами класу невидиме.

Якщо елементові класу не присвоєно специфікатор доступу, то від доступний у підкласах та інших класах пакета. Це доступ за замовчуванням ("дружній"). Якщо треба, щоб член класу був доступний за межами пакета, проте тільки у підкласах цього класу, його оголошують як ***protected***. Інформацію щодо видимості елементів класів наведено у табл. 3.2.

Т а б л и ц я 3.2. **Доступ до елементів класу**

Атрибути доступу	private	без специфікатора	protected	public
Той же клас	так	так	так	так
Підклас в тому ж пакеті	ні	так	так	так
Не-підклас в тому ж пакеті	ні	так	так	так
Підклас в іншому пакеті	ні	ні	так	так
Не-підклас в іншому пакеті	ні	ні	ні	так

Створення власних пакетів. Під час створення програмного забезпечення мовою *Java* можна користуватися такими рекомендаціями:

- кожен клас (за винятком внутрішніх) розмістити в окремий власний файл з тим же іменем і розширенням *.java;*
- створити кореневу директорію, в якій міститимуться усі ваші пакети;
- якщо до складу проекту зачислено понад 5−9 класів, бажано розмістити їх у підпакетах.

Щоб уникнути конфлікту імен класів, фірма Sun рекомендує розробникам за основу для побудови імен пакетів використовувати цілковито уточнене ім'я їхнього домену в *Internet*, записане у зворотному порядку. Наприклад, для *franko.lviv.ua,* пакети повинні починатися з назви *ua.lviv.franko.pmi.kpm* і т. д.

Клас Package. Швидкий розвиток і спеціалізація пакетів спричинила виникнення у *Java2* нового класу *Package* (пакет *java.lang*), який дає змогу працювати з інформацією щодо версії пакета. Наступна програма використовує клас *Package* з метою отримання інформації про доступні пакети:

```
class PkgTest {
    public static void main(String[] args) {
        Package pkgs[ ];
        // повертає всі пакети,
        // які в даний момент відомі програмі
        pkgs = Package.getPackages()
        for (int i=0; i < pkgs.length(); i++)
```

```
            System.out.println (
                    pkgs[i].getName ()+  "  " +
                   pkgs[i].getImplementationTitle()+" "
                    +pkgs[i].getImplementationVendor()+  "  "
                   +pkgs[i].getImplementationVersion());
        }
}
```

Ця програма для кожного з пакетів виводить його ім’я, заголовок, ім’я виробника і номер версії.

СПИСОК ЛІТЕРАТУРИ

1. *Аарон И.Волш.* Основы программирования на Java для World Wide Web. − М.: Диалектика, 1996. − 512 с.
2. *Бернард Ван Хейк.* JDBC: Java и базы данных. − М.: ЛОРИ, 1999. − 320 с.
3. *Брюс Э.* Философия Java. Библиотека программиста. − СПб: Питер, 2001. − 880с.
4. *Вейнер П.* Языки программирования Java и JavaScript. − М.: ЛОРИ, 1998. − 242 с.
5. *Дунаев С.* Доступ к базам данных и техника работы в сети. − М.: Диалог-МИФИ, 2000. − 416 с.
6. *Маслов В.В*. Основы программирования на языке Java.- М.: Горячая линия -Телеком, 2000. − 132 с.
7. *Морган М.* Java 2. Руководство разработчика. − М.: Издательский дом "Вильямс", 2000. − 720 с.
8. *Нортон П., Станек У.* Программирование на Java. SAMS Publishin, 1996, " СК Пресс", 1998. − В 2 кн.
9. *Ноутон П., Шилдт Г.* Java 2. − СПб: БХВ-Петербург, 2000. − 1072 с.
10. *Чекмарев А*. Средства визуального проектирования на Java. − СПб.: BHV-Санкт-Петербург, 1998. − 400с.

Навчальне видання

Копитко Марія Федорівна
Іванків Катерина Степанівна

Основи програмування мовою Java

Редактор
Технічний редактор
Коректор

Підп. до друку . .2002. Формат 60x84/16. Папір друк. Друк на різогр.
Умови друк. арк. . Тираж 100 прим. Зам. .
Видавничий центр Львівського національного університету
імені Івана Франка. 79000, м.Львів, вул. Дорошенка, 41.