МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Є. Ю. Беркунський

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Методичні вказівки для студентів напрямку "Комп'ютерні науки"

Рекомендовано Методичною радою НУК

Миколаїв 2007

УДК 004.045: 004.432

Беркунський Є.Ю. Об'єктно-орієнтоване програмування мовою Java: Методичні вказівки для студентів напрямку "Комп'ютерні науки". – Миколаїв: НУК, 2006. – 52 с.

Кафедра інформаційних управляючих систем і технологій

Методичні вказівки містять опис концепцій об'єктно-орієнтованого програмування та їх реалізацій у мові програмування Java. Методичні вказівки можуть бути корисними при виконанні лабораторних робіт, при роботі над завданнями курсових робіт з а також при самостійному вивченні мови програмування Java.

Методичні вказівки призначені для студентів усіх форм навчання напрямку "Комп'ютерні науки".

Рецензент д-р техн. наук, проф. К.В. Кошкін

Згідно з наказом № 110 від 25.04.07 "методичні вказівки публікуються в авторській редакції, і відповідальність за їх редагування несе автор".

Навчальне видання

БЕРКУНСЬКИЙ Євген Юрійович

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Методичні вказівки для студентів напрямку "Комп'ютерні науки"

(українською мовою)

Комп'ютерна верстка *О.М. Черевата*

Коректор *М.О. Паненко*

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ДК № 2506 від 25.05.2006 р.

Підписано до друку 06.11.07. Папір офсетний. Формат 60×84/16. Друк офсетний. Гарнітура "Таймс". Ум. друк. арк.3,1. Обл.-вид. арк. 3,3. Тираж 150 прим. Вид. № 39. Зам. № 355. Ціна договірна.

Видавець і виготівник Національний університет кораблебудування, 54002, м. Миколаїв, вул. Скороходова, 5

Вступ

Об'єктно-орієнтовані мови програмування вперше з'явились у 60-ті роки минулого сторіччя. Проте, в останні 10 років ми є свідками того, як безпрецедентно зростає їх застосування у створенні програмного забезпечення. Спочатку ці мови програмування не користувалися популярністю, проте успіх таких мов, як Java, C++, Object Pascal(Delphi) привернув увагу до об'єктно-орієнтованих технологій. Це не є випадковим. Після довгих років забуття та важкої боротьби з міцно встановленими традиціями, об'єктно-орієнтоване програмування (ООП) в своєму розвитку досягло такого рівня, що люди, нарешті, змогли побачити потенціальні можливості цієї технології. З'явилася незліченна кількість книг і спеціальних випусків академічних (і не тільки) журналів, присвячених цьому предмету. Щоб оцінити цю активність, можна відзначити, що ООП вітається з більшим ентузіазмом, ніж той, котрий ми бачили раніше при проголошенні таких революційних ідей, як "структурне програмування" чи "експертні системи". Раніше вам би довелось переконувати свого керівника дозволити вам використовувати об'єктно-орієнтовану мову. Сьогодні у багатьох компаніях її застосування є обов'язковим. Не буде перебільшенням сказати, що зроблені правильні висновки.

Чому ООП так популярно?

Наведемо деякі (найголовніші) причини величезної популярності ООП в останнє десятиліття:

- сподівання, що ООП може просто і швидко призвести до росту продуктивності і поліпшення надійності програм, допомагаючи тим самим розв'язати кризу в програмному забезпеченні;
- бажання перейти від існуючих мов програмування до нових технологій;
- надихаюча подібність з ідеями, що народилися в інших областях.

Об'єктно-орієнтоване програмування є лише останньою ланкою в довгому ланцюзі рішень, що були запропоновані для розв'язання "кризи програмного забезпечення". Відверто кажучи, криза програмного забезпечення просто означає, що наша уява і ті задачі, що ми хочемо вирішити за допомогою комп'ютерів, майже завжди випереджають наші можливості.

На жаль, є й інша причина особливої популярності таких мов програмування, як Object Pascal і C++ (по контрасту з Smalltalk і Java). Вона полягає в тому, що й адміністрація і розроблювачі сподіваються, що програміст, який розробляє програми мовами C чи Pascal може перейти на C++ чи Object Pascal з тією же легкістю, з якою відбувається додавання декількох літер на титульний лист сертифіката про спеціальність. На жаль, так відбувається не завжди. ООП є новим розумінням того, що власне називається обчисленнями, а також того, як ми можемо структурувати інформацію всередині комп'ютера. Щоб стати професіоналом у техніці ООП *потрібна повна переоцінка звичних методів розробки програм*. В цьому посібнику ми спочатку розглянемо основи "чистої" об'єктно-орієнтованої мови програмування Java, а потім – основні ідеї ООП, та як вони реалізуються в Java.

Чому саме Java?

Для вивчення технології об'єктно-орієнтованого програмування можна обрати декілька можливих сучасних мов програмування: C++, Java, Delphi та ін. Проте автор зупинився на Java з наступних причин. По-перше, звісно, Java є об'єктно-орієнтованою мовою програмування, тобто містить основні концепції технології програмування, що забезпечує багаторазове використання коду. Крім того, в мові Java використовується лаконічний синтаксис, подібний до мови C++, але при цьому в ньому відсутні деякі механізми, що викликають багато труднощів у програмістів-початківців, такі як арифметика вказівників, об'єднання і структури, а також вирази невизначеного типу та ін.

Розділ 1. Основи мови Java

1.1. Як працює Java-програма

Мова Java є об'єктно-орієнтованою, багатопотоковою, динамічною і таке інше, але зовсім не ці властивості перетворюють її в "найкращу мову прикладного програмування". Головне тут те, що Java-програми

виконуються у віртуальній машині, розміщеній усередині комп'ютера, на якому запущена програма.

Java-програма не має ніякого контакту із справжнім, фізичним комп'ютером; усе, про що вона знає – це віртуальна машина. Такий підхід приводить нас до деяких важливих висновків.

По-перше, Java-програми не залежать від комп'ютерної платформи, на якій вони виконуються. Коли ви напишете і скомпілюєте Java-програму, вона буде працювати без змін на будь-якій платформі, де є віртуальна машина. Іншими словами, Java-програма завжди розробляється тільки для єдиної платформи – віртуальної машини.

Переносимість мови чи переносимість програми

Ми маємо повне право сказати, що мова Java машинонезалежна, тобто переносима. Однак це буде лише частина правди. Мова ANSI C, наприклад, теж не залежить від платформи, однак програми, що розроблені нею не є переносимими – їх необхідно щораз компілювати заново на кожній новій платформі. Крім того, мова ANSI C залишає такі речі, як розміри і формати внутрішніх структур даних, на розсуд розроблювачів конкретного операційного середовища – у Java ж усі вони заздалегідь строго визначені і незмінні. І це усього лише одна з переваг!

По-друге, віртуальна машина вирішує, що Java-програмі дозволено, а що ні в якому разі робити не можна. Програми на мовах типу C чи C++ запускаються безпосередньо операційною системою. Тому вони одержують прямий доступ до системних ресурсів комп'ютера, включаючи оперативну пам'ять і файлову систему.

Оскільки Java-програми виконуються віртуальною машиною, її розроблювачі і вирішують, що можна, а чого не можна дозволяти робити програмі. Оточення, у якому працює Java-програма, називається оболонкою часу виконання (*runtime environment*). Віртуальна машина відіграє роль бастіону на шляху між Java-програмою і комп'ютером, на якому та виконується. Java-програма ніколи не зможе одержати прямий доступ до пристроїв уведення-виведення, файлової системи і навіть пам'яті. Замість Java-програми все це робить віртуальна машина.

Java – революційна мова в тім розумінні, що вона не містить властивостей, необхідних тільки з розумінь зворотної сумісності (властивостей, що обов'язкові для застарілих мов програмування). У той же час вона зовсім не змушує вивчати масу нововинайдених програмістських кон-

цепцій. Java являє собою синтез декількох мов програмування – її попередниць.

Коротко говорячи, Java базується на найкращих конструкціях мови C++ з деякими найкращими концепціями, що були взяті з інших мов програмування. З неї вилучені деякі важкі для розуміння і навряд чи комусь по-справжньому потрібні властивості (які приводять до виникнення більшості помилок). Розглянемо основні властивості мови Java.

Розроблювачі мови Java довго розмірковували про те, чого програми, що створені мовою C++, так схильні до помилок. Вони забезпечили мову Java засобами, що дозволяють виключити саму можливість створювати програми, в яких було б приховано найбільш поширені помилки. Для цього в мові Java зроблено наступне:

• Виключена можливість явного виділення та звільнення пам'яті.

Пам'ять в мові Java звільнюється автоматично за допомогою механізму "прибирання сміття". Програміста захищено від помилок, пов'язаних з неправильним використанням пам'яті.

• Введено справжні масиви та заборонена арифметика вказівників.

Тепер програмісти в принципі не можуть стерти дані з пам'яті внаслідок неправильного використання вказівників.

• Виключена можливість переплутати оператор присвоювання з оператором порівняння на рівність.

Тепер неможна навіть скомпілювати вираз if(ntries=3)…(ця помилка – джерело більшості непорозумінь в мовах C та C++).

• Виключено множинне наслідування. Воно замінене новим поняттям – *інтерфейсом*, що запозичене з Objective C.

Інтерфейс дає програмісту майже все, що той може отримати від множинного наслідування, уникаючи при цьому складностей, що виникають при управлінні ієрархіями класів.

1.2. Перші програми на мові Java

Традиційно, при вивченні C-подібних мов, перша програма повинна виводити текст "Hello, World!". Ми також не будемо відступати від цієї традиції, проте, найчастіше мову Java застосовують для написання двох видів програм: "автономних" програм і аплетів (насправді, можна написати і комбіновану програму, яка може бути і аплетом, і автономною програмою, в залежності від способу запуску), тому буде дві програми.

Приклад 1.1

```
/** Перша автономна програма Java яка виводить текст "Hello, World!"*/
public class First {
        /* статичний метод main, що повинен бути присутнім
          * у автономній програмі
          */
        public static void main(String[] args) {
                System.out.println("Hello, World!");
        }
}
```

Як можна бачити з цього прикладу, програма мовою Java – це опи-сання класу деякого об'єкта з іменем First (це ім'я буде іменем програ-ми), що містить в собі обов'язковий метод main – метод, з якого почи-нається виконання програми. У цьому методі можуть бути створені інші об'єкти та управління програмою може бути передано їм. Оскільки ця наша програма дуже проста, то вся робота (виведення повідомлення) виконується прямо тут – через звертання до стандартного об'єкта System.out повідомленням про виклик його метода println (що друкує на екрані потрібний нам текст).

Приклад 1.2

```
/** Перший аплет Java
 *  виводить текст "Hello, World!"
*/
import java.awt.*;
import java.applet.*;

/** Головний клас аплету */
public class FirstApplet extends Applet {

        /* Метод, що викликається для відображення аплету */
        public void paint(Graphics g) {
                g.drawString("Hello, World!",10,25);
        }
}
```

У порівнянні з попереднім прикладом можна побачити, що у аплета ніби немає метода main, з якого починається виконання програми. Про-

те, насправді аплет має метод main, що унаслідується від стандартного класу Applet, який автоматично запускає життєвий цикл аплета.

Питання, як ввести в комп'ютер, скомпілювати і виконати ці та інші програми, розглядаються у Додатку А.

1.3. Огляд структури Java-програми

Усі Java-програми містять у собі чотири основні різновиди будівельних блоків: **класи** та **інтерфейси** (classes and interfaces), **методи** (methods), **змінні** (variables) і **пакети** (packages). Якою б з мов програмування ви не користувалися раніше, ви швидше за усе вже добре знайомі з методами, що є не що інше, ніж функції, або підпрограми, та зі змінними, у яких зберігаються дані. З іншого боку, класи являють собою фундамент об'єктно-орієнтованих властивостей мови Java. Поки що, для простоти, можна сказати, що клас – це деяке ціле, що містить у собі змінні і методи. Нарешті, пакети містять у собі класи і допомагають компілятору знайти ті класи, що потрібні йому для компіляції програми.

Java-програма може містити в собі будь-яку кількість класів, але один з цих класів завжди має особливий статус і безпосередньо взаємодіє з оболонкою часу виконання. Як такий особливий клас, оболонка часу виконання завжди сприймає перший із класів, визначених у тексті програми. Ми будемо називати цей клас первинним класом (primary class). У первинному класі обов'язково повинні бути визначені один чи кілька спеціальних методів.

Коли програма запускається з командного рядка, як ми робили це з прикладом 1.1, системі потрібно тільки один спеціальний метод, що повинний бути присутнім у первинному класі, – метод main, (та бажано, ще один особливий метод – конструктор). У аплеті первинний клас повинний містити вже кілька таких спеціальних методів, один з яких був визначений у нашому прикладі 1.2, це метод paint (Аплет може не мати власного конструктора, проте користуватися тим, який унаслідується).

Тепер ми перейдемо до докладного розгляду кожного з чотирьох основних блоків Java-програми: змінних, методів, класів і пакетів.

1.4. Змінні

Скоріше за все, поняття змінної не вимагає занадто докладних пояснень; зручніше за все уявляти собі змінну як сховище для одиниці даних, що має власне ім'я. Будь-яка змінна в мові Java, як і в більшості інших мов програмування, належить до визначеного типу. Тип змінної визначає, якого роду інформацію можна в ній зберігати. Наприклад, змінні типу **int**

використовуються для збереження цілочислових значень. Приведемо приклад використання змінної цього типу.

Приклад 1.3

```
public class UsesInt {
        public static void main(String[] args) {
                int i = 4;
                System.out.println("Value of i="+i);
        }
}
```

У цьому прикладі ми використовували знак операції присвоювання = для того, щоб присвоїти змінній **i** значення **4**, а потім вивели значення цієї змінної за допомогою методу System.out.println. Тип змінної, котрий ми використовували в даному прикладі, відноситься до однієї з двох великих груп типів, що використовуються у Java, – до примітивних типів (primitive types). Інша велика група типів поєднує в собі посилальні типи (reference types), що містять у собі типи, визначені користувачем, рядки і типи масивів. До примітивних типів відносяться стандартні, вбудовані в мову типи для представлення чисельних значень, одиночних символів і логічних значень. Навпаки, усі посилальні типи є динамічними типами. Головні розбіжності між двома згаданими групами типів перелічені в табл. 1.1.

Таблиця 1.1. **Порівняння примітивних і посилальних типів**

Характеристика	Примітивні типи	Посилальні типи
Чи визначені в самій мові Java?	Так	Ні
Чи мають визначений розмір?	Так	Ні
Чи повинна для змінних цих типів виділятися пам'ять під час роботи програми?	Ні	Так

На практиці найважливішим розходженням між примітивними і посилальними типами є те, про що свідчить останній рядок табл. 1.1, а саме – що пам'ять для змінних посилального типу повинна виділятися під час виконання програми. Використовуючи змінні посилальних типів, ми повинні явно вимагати необхідну кількість пам'яті для кожної змінної перш, ніж ми зможемо зберегти в цієї змінний деяке значення. Причина цього проста: оболонка часу виконання сама по собі не знає, яка кількість пам'яті потрібна для того чи іншого посилального типу. Розглянемо при-

клад, що ілюструє це розходження. При розгляді прикладу майте на увазі, що всі типи масиву відносяться до посилальних типів.

Приклад 1.3

```
public class Variables {
        public static void main(String[] args) {
                int myPrimitive;          // змінна примітивного типу
                int myReference[];        // змінна посилального типу
                myPrimitive = 1;
               /* одразу після описання ми можемо записувати дані
                * у змінну примітивного типу
               */
                myReference = new int[3];
               /* проте, перш ніж зберігати дані в змінній посилального типу,
                * ми повинні виділити пам'ять для неї і лише після цього
                * ми можемо писати у неї дані
               */

                myReference[0] = 0;
                myReference[1] = 1;
                myReference[2] = 2;
                 //
        }
}
```

Примітка. *Як ви можете бачити, посилальні типи дуже схожі на вказівники, що застосовуються в C/C++. Однак є і серйозні відмінності. По-перше, використовуючи посилальні типи, ви не можете одержати доступ до фактичних адрес даних у пам'яті. А по-друге, неможливість одержати доступ до адреси в пам'яті в мові Java означає, що в цій мові цілком відсутня арифметика вказівників.*

1.5. Примітивні типи

Спочатку розглянемо примітивні типи мови Java. З одним з цих типів – типом **int** – ми вже познайомилися вище на конкретному прикладі. Усього в мові Java визначено вісім примітивних типів, що перелічені в табл.1.2. Зауважимо, що розміри пам'яті що виділяються для змінних примітивних типів жорстко обумовлені стандартом мови програмування

Java і повинне зберігатися у будь-яких реалізаціях цієї мови. Цей факт прямо слідує лозунгу мови Java – "Write once – work everywhere" (Написано один раз – працює скрізь).

Таблиця 1.2. **Примітивні типи мови Java**

Тип	Розмір	Діапазон	Приклад
byte	1 байт	від –128 до 127	125
short	2 байти	від –32768 до 32767	–23
int	4 байти	від –2147483648 до 2147483647	2002300
long	8 байт	від –922372036854775808 до 922372036854775807	1243565
float	4 байти	Залежить від розрядності числа	1.2f
double	8 байт	Залежить від розрядності числа	123.4
boolean	1 біт	false, true	true
char	2 байти	Усі символи стандарту Unicode	

1.6. Посилальні типи

Як ви вже знаєте, посилальні типи відрізняються від примітивних тим, що вони не визначені в самій мові Java, і тому кількість пам'яті, що потрібна для змінних цих типів, заздалегідь знати неможливо. У прикладі 1.3 ми вже зустрічалися з одним з посилальних типів – типом масиву. Масиви в мові Java можуть складатися зі змінних будь-якого іншого типу цієї мови, включаючи типи, визначені користувачем (які становлять більшість типів, використовуваних на практиці).

Перш ніж ми перейдемо до докладного розгляду посилальних типів, треба освоїтися з деякими термінами, що відносяться до цієї області. Коли ми виділяємо пам'ять для змінної посилального типу за допомогою оператора **new**, ми тим самим ***реалізуємо*** цей посилальний тип. Таким чином, кожна змінна посилального типу є ***реалізацією*** чи ***екземпляром*** відповідного типу.

Ця термінологія може здатися новою і незвичною, тому варто розглянути процес реалізації докладніше. Проблема полягає в тім, що мова Java не дозволяє нам просто оголосити змінну посилального типу і відразу ж почати записувати в неї значення. Ми повинні спочатку запросити в оболонки часу виконання деякий обсяг пам'яті (що визначається автоматично в залежності від типу змінної, що реалізується), а оболонка, у свою чергу, повинна зробити запис у своїх внутрішніх таблицях, що ми активізували змінну даного посилального типу. Весь цей процес у цілому і називається реалізацією змінної. Після реалізації, коли ми маємо у своє-

му розпорядженні екземпляр змінної даного типу, ми вже можемо використовувати цей екземпляр для збереження даних. Важливо розуміти, що екземпляр змінної і сам посилальний тип, до якого ця змінна відноситься, є якісно різними поняттями – для збереження змінної можна використовувати тільки реалізований екземпляр змінної посилального типу.

1.7. Типи, визначені користувачем

Більшість мов дозволяють програмісту визначати нові типи. Наприклад, у мові C нові типи можна створювати за допомогою оператора **struct**, а в Pascal – за допомогою записів (**record**). Мова Java дозволяє визначати нові типи за допомогою класів (**class**), а також за допомогою інтерфейсів (**interface**).

На найпростішому рівні розгляду класи схожі на структури або записи – вони теж дозволяють зберігати набори змінних різних типів. Однак є і важлива відмінність: класи крім змінних можуть містити в собі також і методи. Нижче приведений приклад оголошення нового типу, названого "MyType". Ключове слово **public**, що стоїть перед визначенням типу, є так званим модифікатором доступу (access modifier) і вказує на те, що доступ до даних членів класу можуть одержати методи, що не входять у даний клас.

Приклад 1.4

```
class MyType {
        public int myDataMember = 4;
        public void myMethodMember() {
                System.out.println("I'm a member!");
                System.out.println("myData=" + myDataMember);
        }
}
```

Зверніть увагу на те, що цей приклад нагадує за структурою власне програми мовою Java, що зустрічались раніше. Ця подібність відображує ту подвійну роль, яку класи грають у мові Java.

У програмах, що наводилися вище, як приклади, класи використовувалися як засіб організації вмісту – даних і алгоритмів кожної програми. Але класи можуть також використовуватися і для визначення нових типів. Змінні типів, визначених через класи, називаються об'єктами, або реалізаціями чи екземплярами відповідних класів. Створення чи реалізація

об'єкта здійснюється за допомогою оператора **new**, а доступ до членів (складових частин) класу – за допомогою оператора "крапка" (**.**).

Приклад 1.5

```
public class RunMe {
        public static void main(String ARGV[]) {
                MyType Mine = new MyType();
                int i = Mine.myDataMember;
                Mine.myMethodMember();
        }
}
```

Приклад 1.5 ілюструє три основних види дій, які можна робити з об'єктом: створення об'єкта, доступ до членів-змінних об'єкта і доступ до членів-методів цього об'єкта. Останній рядок коду в цьому прикладі викликає метод myMethodMember, що виводить на екран наступне:
I'm a member!
myData=4

Оскільки тип myDataType є посилальним типом, ми повинні використовувати оператор new. Цей оператор запитує в системи визначену кількість пам'яті для збереження нашого об'єкта. Крім того, ми можемо визначити, які ще дії повинні виконуватися в момент реалізації класу, визначивши так званий конструктор (constructor). Ось як виглядає конструктор для типу myDataType, єдина функція якого – повідомити про те, що відбувається реалізація класу.

Приклад 1.6 Конструктор, який повідомляє про реалізацію класу.

```
public class MyType {
        int myDataMember = 0;
        public MyType() {
                System.out.println("Object in process!");
        }
}
```

Конструктори можна використовувати також для ініціалізації (присвоєння початкових значень) членів-змінних даного класу. Ось приклад конструктора, що присвоює змінній myDataMember цілочислове значення, передане цьому конструктору через аргумент.

Приклад 1.7 Конструктор, який виконує ініціалізацію змінної.

```
public MyType(int val) {
        System.out.println("setting myDataMember=" + val);
        myDataMember = val;
}
```

Тепер представимо, що обидва приведених вище конструктора визначені в нашому класі myDataType. Ось ще один фрагмент програми, у якому використовуються обидва ці конструктора.

Приклад 1.8 Клас з двома конструкторами.

```
public class RunMe {
        public static void main(String argv[]) {
                MyType instance1 = new MyType();
                instance1. myMethodMember();
                MyType instance2 = new MyType(100);
                instance2. myMethodMember();
        }
}
```

Результат роботи цієї програми буде мати наступний вид:

```
Object in progress!
I'm a member!
myData=4
setting myDataMember=100
I'm a member!
myData=100
```

1.8. Тип String

Дотепер ми розглядали примітивні типи і типи, що визначаються користувачем. Тепер розглянемо один особливий тип, що являє собою гібрид цих двох типів, – тип **String** (тип строкових змінних). В основі своїй тип **String** є типом, визначеним користувачем, тому що він визначається як однойменний клас **String**, що містить у собі методи і змінні. Але в той же час цей тип виявляє деякі властивості примітивного типу, що виражається, зокрема, у тім, як здійснюється присвоєння значень змінним цього типу.

```
String myString = "Hello!";
```

Незважаючи на те, що такий спосіб оголошення й ініціалізації змінних типу **String** є не зовсім законним з погляду синтаксису типів, визначених користувачем, не можна не визнати, що для об'єкта, який так часто зустрічається в програмуванні, як рядок символів, цей спосіб є самим очевидним і зручним. Крім того, для конкатенації (додавання) двох рядків можна використовувати знак операції +.

```
int myint = 4;
String anotherString = myString + "myint is " + myint;
```

Після виконання зазначених дій значення змінної anotherString буде "Hello! myint is 4". Однак оскільки anotherString є в той же самий час і об'єктом, ми можемо викликати методи – члени класу **String**. Так, щоб вирізати перші п'ять символів рядка anotherString, потрібно написати наступний вираз:

```
String helloString=anotherString.substring(5);
```

Як бачите, реалізація змінних типу **String** не вимагає застосування оператора **new**. З погляду практики програмування це дуже зручно, оскільки рядкові змінні використовуються дуже часто. Однак, програмуючи мовою Java, ви завжди повинні пам'ятати про те, що тип **String** є особливим – це єдиний визначений користувачем тип, змінні якого можуть з'являтися і використовуватися без застосування оператора **new**.

1.9. Типи масиву

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

Приклад 1.9 Виділення пам'яті для масивів

```
int myIntArray[];
myIntArray = new int[3];
MyType myObjectArray[];
myObjectArray = new MyType[3];
```

Оператор **new** дає команду оболонці часу виконання виділити необхідну кількість пам'яті під масив. Як видно з цього приклада, не-

обов'язково повідомляти розмір масиву тоді ж, коли ви створюєте змінну-масив. Після того, як ви створили масив оператором **new**, доступ до цього масиву здійснюється точно так само, як у мовах C чи Pascal.

Приклад 1.10 Присвоювання значень елементам масивів

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

Масиви в мові Java мають три важливих переваги перед масивами в інших мовах. По-перше, як ви тільки що бачили, програмісту нетреба (і неможна!) вказувати розмір масиву при його оголошенні. По-друге, будь-який масив у мові Java є змінною – а це значить, що його можна передати як параметр методу і використовувати як значення, що повертається методом. І по-третє, не складає ніякої складності довідатися, який розмір даного масиву в будь-який момент часу. Наприклад, так визначається розмір масиву, що ми оголосили вище.

Приклад 1.10 Отримання довжини масиву

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

1.10. Методи

Метод у мові Java являє собою підпрограму, аналогічну функціям мов C і Pascal. Кожен метод має тип значення, що повертається, і може викликатися з передачею деяких параметрів. Тіло методу може містити оголошення змінних і оператори. На відміну від мови C оголошення змінних можуть розташовуватися в будь-якім місці тіла методу, у тому числі і після деяких операторів.

Нижче ми розглянемо питання, зв'язані зі значеннями, що повертаються методами, і з передачею параметрів методам. Наприкінці розділу ми познайомимося з особливою властивістю мови Java, що називається перевантаженням методів (method overloading), завдяки якій можна давати те саме ім'я декільком методам, що розрізняються між собою списком прийнятих параметрів.

З кожним методом повинний бути співвіднесений тип значення, що повертається ним. Тип **void**, що був приписаний нашому методу main у попередніх прикладах, є спеціальним способом указати системі, що даний метод не повертає ніякого значення. Методи, які повертають тип яких оголошений за допомогою ключового слова **void**, аналогічні процедурам мови Pascal. Методи, у яких значення, що повертається, належить до будь-якого іншого типу, крім **void**, повинні містити у своєму тілі оператор **return**. Значення, що повертається, може належати до кожного з типів, про які ми говорили в розділі "1.4 Змінні", – включаючи як примітивні типи, так і типи, визначені через клас. Нижче наведені приклади методів, що не повертають ніякого значення, і методів, що повертають значення визначеного типу.

Приклад 1.11 Викликання методів

```
public class MethodExamples {
        static void voidMethod() {
                System.out.println("I am a void method");
        }
        static int returnInt() {
                int i = 4;
                System.out.println("returning 4");
                return i;
        }
        static public final void main(String S[]) {
                System.out.println("Hello, methods!");
                System.out.println("Calling a void method");
                voidMethod();
                int ans = returnInt();
                System.out.print("method says –.–");
                System.out.println(ans);
        }
}
```

Як ви, імовірно, помітили, виклик методів у цьому прикладі здійснювався точно так само, як ми викликали би функції чи процедури в іншій, не об'єктно-орієнтованій мові. Це пов'язано з тим, що в нашому прикладі статичні методи викликали інші статичні методи, що належать до того ж класу. Те ж саме вірно і для тих випадків, коли нестатичні (динамічні) методи викликають інші динамічні методи. Однак коли динамічні методи викликають статичні методи і навпаки, а також, коли виникає необхідність викликати метод з іншого класу, синтаксис виклику міняєть-

ся. Докладно ці зміни будуть розглянуті у розділі "Об'єктно-орієнтована розробка програм".

1.11. Передавання параметрів

Як параметри в мові Java можна передавати змінні будь-якого типу, включаючи типи, визначені через класи, і масиви змінних будь-якого типу і розміру. Однак змінні примітивних типів, передані як параметри, поводяться інакше, ніж змінні посилальних типів у тому ж контексті. Спочатку розглянемо передачу змінних примітивних типів.

Усі змінні примітивних типів передаються методам за значенням (by value). Це означає, що в момент виклику методу робиться копія змінної, переданої методу. Якщо метод буде змінювати у своєму тілі значення переданої йому як параметр змінної, то вміст вихідної змінної змінюватися не буде тому що всі дії будуть виконуватися з її копією. Цей спосіб передавання традиційний для С-подібних мов.

Навпроти, значення змінних посилального типу, які передаються як параметри, можна змінити в тілі методу. Розглянемо приклад з масивом цілих чисел.

Приклад 1.12 Викликання методів з параметром-масивом

```
public class ReferenceParameterExample {
        static void changeArray(int referenceVariable[]) {
                referenceVariable[2] = 100;
        }
        public static void main(String argv[]) {
                int anArray[] = new int[3];
                anArray[2] = 10;
                System.out.print("anArray[2]=");
                System.out.println(anArray[2]);
                changeArray(anArray);
                System.out.println(anArray[2]);
        }
}
```

Виведення цієї програми має такий вигляд:

```
anArray[2]=10
100
```

Таким чином, можна зробити висновок, що коли ми передаємо методу як параметр змінну посилального типу, ми явно змінюємо те, на що вказує ця змінна, – у нашому випадку масив цілих чисел.

1.12. Рядкові змінні і передача параметрів

Незважаючи на те, що тип **String** є визначеним користувачем типом, він не поводиться як посилальний тип при передачі параметрів. Змінні типу **String** як параметри методу завжди передаються за значенням, – тобто передавши методу рядкову змінну, ви в тілі методу будете фактично працювати з копією цієї рядкової змінної. Іншими словами, зміна значення рядкової змінної в тілі методу не впливає на значення цієї ж змінної зовні методу.

1.13. Перевантаження методів

Ви напевно зіштовхувалися з необхідністю створювати дві чи кілька функцій, що виконують, по суті, ті самі дії, але мають різні списки параметрів. Мова Java дає в таких ситуаціях більш витончений вихід з положення. У цій мові ви можете присвоїти одне і те саме ім'я декільком методам, що розрізняються списками своїх параметрів. Наприклад, нехай ми маємо метод для порівняння двох цілих чисел (див. приклад 1.13а)

Приклад 1.13а Метод, що порівнює два цілих числа

```
public static String compareNums(int i, int j) {
        if (i == j) {
                return "Числа "+ i + " та " + j +" є рівними";
        }
        if (i > j) {
                return "Число " + i + " більше ніж " + j;
        }
        return "Число " + j + " більше ніж " + i;
}
```

Тепер представте, що нам у програмі знадобилося порівняти не два, а три цілих числа. Звичайно, можна було б визначити для цього новий метод з ім'ям типу compareThreeNums. Але, на щастя, мова Java дозволяє обійтися без множення кількості імен у програмі.

Приклад 1.13б Метод, що порівнює три цілих числа

```
public static String compareNums(int i, int j, int k) {
        String S = compareNums(i,j);
        S = S + "\n";
        S = S + compareNums(i,k);
        return S;
}
```

Складаючи щораз інший список параметрів, ми в такий спосіб можемо визначити будь-яку кількість методів з тим самим ім'ям compareNums. Це стає особливо зручним у тих випадках, коли потрібно зробити ту саму дію над змінними різних типів. Як відомо, Java не дозволяє передавати, приміром, змінні типу **double** методу, параметри якого мають тип **int**. Однак ніщо не заважає нам удатися до перевантаження, визначивши ще один метод з тим же ім'ям і зі списком параметрів типу **double** (чи будь-якого іншого типу).

Приклад 1.13в Метод, що порівнює два числа типу double

```
public static String compareNums(double i, double j) {
        if (i == j) {
                return "Числа "+ i + " та " + j +" є рівними";
        }
        if (i>j) {
                return "Число " + i + " більше ніж " + j;
        }
        return "Число " + j + " більше ніж " + i;
}
```

Переваги перевантаження методів особливо очевидні для тих, кому доводиться цими методами користуватися: замість того щоб пам'ятати кілька імен різних методів, можна обмежитися запам'ятовуванням тільки одного імені, загального для методів з різними параметрами. В обов'язок компілятора входить з'ясування того, який саме метод потрібно викликати в кожнім випадку.

Приклад 1.13г Виклик перевантажених методів

```
public class overloadMethod {
        public static void main(String[ ] args) {
                int a = 3;
                int b = 4;
                int c = 5;
                double d = 3.3;
                double e = 4.4;
                double f = 5.5;
                String S = compareNums(a,b);
                System.out.println(S);
                S = compareNums(a,b,c);
```

```
            System.out.println(S);
            S = compareNums(d,e,f);
            System.out.println(S);
        }
        ...
}
```

Замість символу ... треба підставити описання перевантажених методів з трьох попередніх прикладів.

1.14. Управляючі структури мови Java

У мові Java реалізовано майже всі управляючі структури, що є у мовах C та C++. Коротко перелічимо їх тут.

1. ***Об'явлення, вирази та інструкції.*** Як інструкції, що завершуються символом "крапка с комою", можуть бути використані вирази таких категорій:

a. вирази присвоювання, в яких використовується базовий (=) та складений оператори присвоювання виду *on*=;

b. префіксні та постфіксні форми виразів з операторами інкременту (++) та декременту (--);

c. конструкції виклику методів;

d. вирази створення об'єктів за допомогою оператора **new**.

Об'явлення використовуються для описання (локальних) змінних та ініціалізації їх початковими значеннями.

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

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

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

Спочатку виконується перевірка значення логічного виразу. Якщо результат дорівнює true, виконується Інструкція1, інакше (і при наявності необов'язкового елементу **else**) – Інструкція2. Одна конструк-

ція **if … else** може бути вкладена всередину елементу **else** іншої з метою перевірки послідовності логічних умов. Також така конструкція може бути вкладена замість Інструкції1, але тоді треба пам'ятати, що елемент **else** завжди буде відноситися до найближчого **if**, яке не "закрито" своїм **else**.

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

```
switch (Цілочисловий Вираз) {
        case n: Інструкції
        case m: Інструкції
        ...
        default: Інструкції
}
```

Тіло **switch**, відоме як блок перемикачів, містить набори інструкцій, яким передують мітки, що починаються з службового слова **case**. Кожній мітці **case** ставиться у відповідність ціла константа. Якщо значення цілочислового виразу співпаде зі значенням деякої мітки, управління буде передано першій інструкції, що йде після цієї мітки. Якщо співпадань не знайдено, виконуються інструкції блоку **default**. Якщо ж мітка **default** відсутня, виконання **switch** завершується. При передаванні управління відповідній мітці виконуються всі наступні за нею інструкції, навіть ті, що мають свої власні мітки **case**. Якщо треба вийти з блоку **switch**, треба використати інструкцію **break**. Стандартами оформлення текстів програм, що рекомендовані розробником Java, вимагається наявність **break** в кожному розділі **case**. Якщо відповідний **break** відсутній, бо того вимагає логіка алгоритму, то замість нього треба поставити коментар "/* break не потрібен */" .

5. **while та do … while.** Синтаксис циклічної конструкції **while** виглядає так:

```
while (ЛогічнийВираз)
        Інструкція
```

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

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

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

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

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

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

Секція ініціалізації дозволяє ініціалізувати (можливо, з попереднім об'явленням) змінні циклу і виконується лише один раз. Логічний вираз піддається перевірці, та її результат, якщо він дорівнює **true**, дає підставу для виконання тіла циклу, після чого обчислюються вирази секції змін, і логічний вираз перевіряється у черговий раз. Цикл повторюється доти, поки результатом перевірки не стане значення **false**. Кожна з двох частин конструкції **for** – СекціяІніціалізації та СекціяЗмін – може бути представлена списком виразів, відокремлених символом ","(кома). Вирази у подібних списках обчислюються зліва направо.

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

```
for  ( ;  ; )
        Інструкція
```

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

Цикли **for**, у відповідності до прийнятої угоди, застосовуються тільки в тих випадках, коли необхідно забезпечити "проходження" по значеннях із визначених діапазонів. Якщо вирази ініціалізації та зміни змінних застосовуються не за призначенням і не пов'язані з логічною умовою

циклу, подібні дії можна кваліфікувати не інакше як порушення угоди і ознаку поганого стилю програмування.

6а. У Java SDK 5.0 з'явилась нова форма оператора **for**, що дозволяє писати більш лаконічні цикли для обробки масивів, колекцій і т.д. – **for-each**:

```
for (Тип_елемента  Ім'я_Параметра :
        Ім'я_Колекції_чи_Масиву) Інструкція
```

такий оператор дозволяє виконати Інструкцію для всіх елементів колекції чи масиву.

7. **Мітки.** Інструкції програми можуть бути позначені мітками (labels). Мітка являє собою змістовне ім'я, що дозволяє посилатися на відповідну інструкцію:

```
Мітка: Інструкція
```

Звертатися до мітки дозволено тільки за допомогою команд **break** та **continue** (вони розглядатимуться далі).

8. **break.** Інструкція **break** застосовується для завершення виконання коду будь-якого блоку. Існують дві форми інструкції – безіменна:

```
break;
```

та іменована:

```
break мітка;
```

Безіменна команда **break** перериває виконання коду конструкцій **switch**, **for**, **while** або **do** і може використовуватися лише всередині цих конструкцій. Команда **break** у іменованій формі може перервати виконання будь-якої інструкції, що помічена відповідною міткою.

Команда **break** найчастіше використовується для примусового виходу з тіла циклу. Для виходу із вкладеного циклу чи блоку, достатньо позначити міткою зовнішній блок і вказати її в інструкції **break** як показано в наступному прикладі:

Приклад 1.14 Використання поміченого **break**

```
private float [ ][ ] matrix;
    public boolean workOnFlag(float flag) {
        int y, x;
        boolean found = false;
        search:
        for (y = 0; y < matrix.length; y++) {
            for (x = 0; x < matrix[y].length; x++) {
                if (matrix[y][x] == flag) {
```

```
                        found = true;
                        break search;
                    }
                }
            }
            if (!found) {
                return false;
            }
            // А тут знайдене значення matrix [y] [x]
            // деяким чином обробляється
            return true;
}
```

Відмітимо, що іменована інструкція **break** – це зовсім не те ж саме, що й сумнозвісна команда **goto**. Інструкція **goto** дозволяє "стрибати" по коду без жодних обмежень, переплутуючи порядок обчислень і збиваючи читача з глузду. Команди же **break** і **continue**, що посилаються на мітку, дозволяють лише акуратно залишити відповідний блок і забезпечити його повторення, при цьому потік обчислень залишається цілком очевидним.

9. **continue.** Команда **continue** застосовується лише у контексті циклічних конструкцій і передає управління на кінець тіла циклу. В ситуації з **while** і **do** це призводить до виконання перевірки умови циклу, а при використанні в тілі **for** інструкція **continue** провокує передавання управління секції змін значень змінних циклу.

Як і **break**, команда **continue** дозволяє використання в двох формах – без імені: **continue**; і іменованій: **continue** мітка. Команда **continue** у формі без імені мітки передає управління в кінець поточного циклу, а іменована – в кінець циклу, позначеного відповідною міткою. Мітка повинна відноситися до циклічного виразу.

10. **return.** Інструкція **return** завершує виконання методу і передає управління до коду-ініціатору. Якщо метод не повертає значень, інструкція виглядає просто: **return**. Якщо ж в оголошенні методу вказано тип параметра, що повертається, до складу команди **return** повинен бути включеним вираз, який може бути присвоєний змінній оголошеного типу. Наприклад, якщо метод повертає значення типу **double**, в інструкції **return** дозволяється використовувати вирази, що відносяться до типів **double**, **float** або будь-якого цілих типів. Треба зауважити, що на відміну від мови

Pascal (та деяких інших) де функції можуть повертати значення тільки простих типів, методи у мові Java можуть повертати значення будь-якого типу.

11. **goto.** У мові Java НЕМАЄ інструкції **goto**, що має змогу передавати управління довільному фрагменту коду, хоча у споріднених мовах аналогічні засоби передбачені. Всі засоби, що були розглянуті раніше, дозволяють створювати зрозумілий і надійний код, а також обходитися без допомоги **goto**.

12. Для обробки виключень, тобто ситуацій, що могли б привести до краху програми (наприклад, ділення на нуль, помилка введення-виведення) використовують конструкцію **try…catch…finally…** Обробка виключень у Java спирається в основному, на конструкції C++ (хоча ззовні більше схожа на Object Pascal). У місці, де виникла проблема, ви, можливо, ще не знаєте що з нею робити, проте знаєте, що просто ігнорувати її не можна – треба зупинитись і передати управління блоку обробки.

```
try {
   // Небезпечна діяльність, що може викликати
   // виключення A, B або C
} catch (A a1) {
   // Оброблювач для випадку A
} catch (B b1) {
   // Оброблювач для випадку B
} catch (C c1) {
   // Оброблювач для випадку C
} finally {
   // Дії, що виконуються у будь-якому випадку
}
```

Розділ 2. Основні концепції ООП

2.1. Коротке викладення принципів

Алан Кей, якого вважають одним з тих, хто започаткував об'єктно-орієнтоване програмування, вважає, що наступні положення є фундаментальними характеристиками ООП:

1. Усе є об'єктом.

2. Програма – це група об'єктів, що говорять одне одному, що робити, через повідомлення. Повідомлення – це запит на виконання

дії, доповнений набором аргументів, що можуть знадобитися при виконанні дії. Простіше кажучи, повідомлення – це виклик функції-методу, що належить деякому об'єкту.

3. Кожен об'єкт має незалежну пам'ять, що складається з інших об'єктів.

4. Кожен об'єкт є представником класу, що виражає загальні властивості об'єктів (таких, як цілі числа чи списки). Тут слово "клас" є аналогом слова "тип".

5. У класі задається поведінка (функціональність) об'єкта. Тим самим всі об'єкти, що є екземплярами одного класу, можуть виконувати ті самі дії.

6. Класи організовані в єдину деревоподібну структуру з загальним коренем, називану ієрархією наслідування. Пам'ять і поведінка, зв'язане з екземплярами визначеного класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Підсумовуючи ці принципи Т.Бадд у роботі [2] дещо образно помітив "Замість процесора, що перемелює біти і шарить по кишенях структур даних, ми отримуємо всесвіт добре вихованих об'єктів, що люб'язно просять одне одного про виконання тих чи інших бажань." Або, при використанні ООП ми вважаємо, що *обчислення – це моделювання*.

Звичайно, об'єкти не можуть у всіх випадках реагувати на повідомлення тільки тим, що чемно звертаються до інших із проханням виконати деяку дію. Це приведе до нескінченного циклу запитів, як якби два джентльмени так і не ввійшли в двері, уступаючи один одному дорогу. На деякій стадії, принаймні деякі об'єкти, повинні виконувати якусь роботу перед пересиланням запиту іншим агентам. Ці дії виконуються порізному в різних об'єктно-орієнтованих мовах програмування.

У мовах, де директивний і об'єктно-орієнтований підходи уживаються разом (таких, як C++, Object Pascal), реальні дії виконуються методами, написаними на основній (не об'єктно-орієнтованій) мові. У чисто об'єктно-орієнтованих мовах (таких, як Smalltalk і Java) це виконується за допомогою "примітивних" чи "вбудованих" операцій, що забезпечуються виконавчою системою більш низького рівня.

Після складання технічного завдання, починається етап проектування, або дизайну, майбутньої системи. Об'єктно-орієнтований підхід до проектування заснований на представленні предметної області задачі у виді множини моделей для мовно-незалежної розробки програмної системи на основі її прагматики.

Прагматика визначається метою розробки програмної системи, наприклад обслуговування клієнтів банку, керування роботою аеропорту, обслуговування чемпіонату світу з футболу і т.п. У формулюванні мети беруть участь предмети і поняття реального світу, що мають відношення до створюваної системи (рис. 2.1).

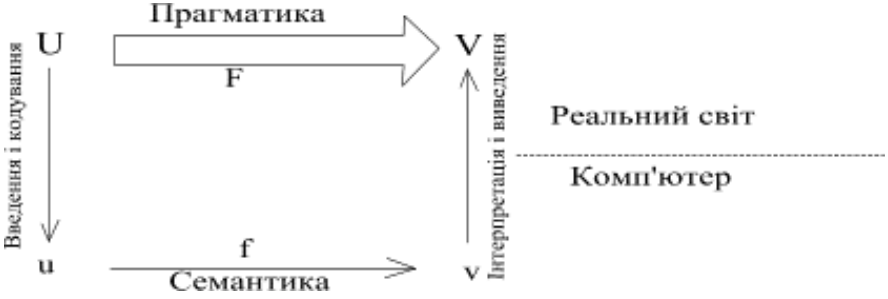

Рис. 2.1. Об'єктно-орієнтоване моделювання

При об'єктно-орієнтованому підході ці предмети і поняття замінюються моделями, тобто визначеними формальними конструкціями, що представляють їх у програмній системі.

Модель містить не всі ознаки і властивості предмета, або поняття, що представляється нею, а тільки ті, котрі істотні для розроблювальної програмної системи. Тим самим модель "бідніше", а, отже, простіше предмета, або поняття що представляється нею. Простота моделі стосовно реального предмета дозволяє зробити її формальною. Завдяки такому характеру моделей при розробці можна чітко виділити всі залежності й операції над ними в створюваній програмній системі. Це спрощує як розробку і вивчення (аналіз) моделей, так і їхню реалізацію на комп'ютері.

Об'єктно-орієнтований підхід допомагає справитися з такими складними проблемами, як:

- зменшення складності програмного забезпечення;
- підвищення надійності програмного забезпечення;
- забезпечення можливості модифікації окремих компонентів програмного забезпечення без зміни інших його компонентів.
- забезпечення можливості повторного використання окремих компонентів програмного забезпечення.

Систематичне застосування об'єктно-орієнтованого підходу дозволяє розробляти добре структуровані, надійні в експлуатації, досить програмні системи, що модифікуються просто. Цим пояснюється інтерес

програмістів до об'єктно-орієнтованого підходу та об'єктно-орієнтованих мов програмування. ООП є одним з напрямків теоретичного і прикладного програмування, що найбільш інтенсивно розвиваються.

2.2. Об'єкти: повідомлення, наслідування і поліморфізм

ООП додає кілька нових важливих ідей до концепції абстрактних типів даних. Головна з них – пересилання повідомлень. Дія ініціюється запитом, зверненим до конкретного об'єкта, а не через виклик функції. У значній мірі це просто зсув наголосу: традиційна точка зору робить основний упор на операції, у той час як ООП на перше місце ставить власне значення. (Викликаєте ви підпрограму push зі стеком і значенням як аргументами, чи ж ви просите об'єкт stack помістити потрібне значення до нього усередину?) Якби це було усе, що є в ООП, ця техніка не розглядалася б як принципове нововведення. Але до пересилання повідомлень додаються могутні механізми перевизначення імен і спільного/багаторазового використання програмного коду.

Неявною в ідеї пересилання повідомлень є думка про те, що інтерпретація повідомлення може мінятися для різних об'єктів. А саме поведінка і реакція, що ініціюються повідомленням, залежать від об'єкта, що одержує повідомлення. Тим самим push може означати одну дію для стека і зовсім іншу для блоку керування механічною рукою. Оскільки імена операцій не зобов'язані бути унікальними, можуть використовуватися прості і явні форми команд. Це приводить до більш легко читаного і зрозумілого коду.

Нарешті, ООП додає механізми наслідування і поліморфізму. Наслідування дозволяє різним типам даних спільно використовувати той самий код, приводячи до зменшення його розміру і підвищенню функціональності. Поліморфізм перекроює цей загальний код так, щоб задовольнити конкретним особливостям окремих типів даних. Упор на незалежність індивідуальних компонентів дозволяє використовувати процес покрокової збірки, при якій окремі блоки програмного забезпечення розробляються, програмуються і налагоджуються до того, як вони поєднуються у велику систему.

2.3. Об'єктно-орієнтоване проектування

Першою задачею проектування програмного продукту є уточнення специфікації. Як відомо початкові описання проекту майже завжди двозначні і незрозумілі в усьому, крім найбільш загальних положень. На цьому етапі ставиться кілька завдань. Одне з них є краще розуміння і від-

чуття того, чим буде кінцевий продукт (принцип "подивися і відчуй" для проектування системи). Потім ця інформація може бути повернута назад замовнику (наприклад, викладачу), щоб побачити, чи є вона відповідною до вихідної концепції. Ймовірно і, можливо, неминуче те, що специфікації для кінцевого продукту будуть змінюватися під час розробки програмної системи, і тому важливо, щоб проект міг легко вмістити в собі нові ідеї, а також щоб потенційно можливі виправлення були виявлені якомога раніше ("Легше великі зміни зробити на початку роботи, ніж маленькі – в кінці"). На цьому ж етапі проводиться обговорення структури майбутньої програмної системи. Зокрема, дії, здійснювані програмною системою, розбиваються на компоненти.

Компонент – це просто абстрактна одиниця, що може виконувати визначену роботу (тобто мати визначені обов'язки). На цьому етапі немає необхідності знати в точності те, як задається компонент та як він буде виконувати свою роботу. Компонент може в остаточному підсумку бути перетворений в окрему функцію, структуру чи клас, чи ж у сукупність інших компонентів (шаблон). На цьому рівні розробки маються дві важливі особливості:

- компонент повинний мати невеликий набір чітко визначених обов'язків;
- компонент повинний взаємодіяти з іншими компонентами настільки слабко, наскільки це можливо.

Розглянемо цей процес докладніше:

Щоб виявити окремі компоненти і визначити їхні обов'язки, команда програмістів пробробляє сценарій системи. Тобто відтворюється запуск додатка, так якби він був вже готовим. Будь-яка дія, що може відбутися, приписується деякому компоненту в якості її обов'язку.

CRC-картки

Як складову частину цього процесу корисно зображувати компоненти за допомогою невеликих індексних карток. На лицевій стороні картки написане ім'я компонента, його обов'язки й імена інших компонентів, з якими він повинен взаємодіяти. Такі картки іноді називаються CRC-картками від слів Component, Responsibility, Collaborator (компонент, обов'язок, співробітники). По мірі того як для компонентів виявляються обов'язки, вони записуються на лицевій стороні CRC-картки.

Перевага CRC-карток у тім, що вони широко доступні, недорогі і з них можна стирати інформацію. Це стимулює експериментування, оскільки альтернативні проекти можуть бути випробувані, вивчені і від-

кинуті з мінімальними витратами. Фізичний поділ карток стимулює інтуїтивне розуміння важливості логічного поділу компонент, що допомагає зробити упор на зв'язності усередині модулів і зачепленні між модулями (які коротенько будуть описані нижче). Невеликий розмір індексної картки служить гарною оцінкою зразкової складності окремого фрагмента – компонент, якому приписується більше задач, чим може помістится на його картці, імовірно, є зайво складним, і повинно бути знайдене більш просте рішення. Може бути, варто переглянути поділ обов'язків чи розбити компонент на два чи більше компоненти.

Виділення компонентів відбувається під час процесу уявного представлення роботи системи. Часто це відбувається як цикл питань "що/хто". По-перше, команда програмістів визначає: що потрібно робити? Це негайно приводить до питання: хто буде виконувати дію? Якщо відбувається деяка дія, повинен бути й агент, якому запропоноване виконувати цю дію. При організації об'єктно-орієнтованої програми кожна дія є обов'язком деякого компонента. Секрет гарного об'єктно-орієнтованого проекту полягає в тому, щоб установити агента для кожної дії.

Готовність до змін

Як було сказано, єдине, що є постійним у житті, – неминучість змін. Те ж саме справедливо для програмного забезпечення. Незалежно від того як ретельно ви намагаєтеся розробити вихідні специфікації і проект програмної системи, майже напевно зміни в бажаннях чи потребах користувача будуть викликати відповідні виправлення в програмі (найчастіше протягом усього життєвого циклу системи). Розроблювачі повинні передбачати це і планувати свої дії відповідним чином:

• Головна мета полягає в тому, що зміни повинні торкатися якнайменше компонент. Навіть принципові нововведення в зовнішньому вигляді чи функціонуванні програми повинні торкнутися одного чи двох фрагментів коду.

• Намагайтеся прогнозувати найбільш ймовірні джерела змін і дозвольте їм впливати на як можливо меншу кількість компонентів програми. Найбільш загальними причинами змін є інтерфейс, формати обміну інформацією, вид вихідних даних.

• Намагайтеся ізолювати і зменшити залежність програмного забезпечення від апаратури. Гарний проект повинний передбачати можливість зміни архітектури.

• Зменшення кількості зв'язків між фрагментами програми знижує взаємозалежність між ними і збільшує імовірність того, що кожен компонент удасться змінити з мінімальним впливом на інші.

• Акуратно заносьте записи про процес розробки і про дискусії, що проводилися навколо принципових рішень, у проектну документацію. Майже напевно колектив, відповідальний за супровід системи і розробку наступних версій, буде відрізнятися від команди, що розробила первісну версію програми. Проектна документація дозволить у наслідку довідатися про мотиви прийнятих рішень і допоможе уникнути витрат часу на обговорення питань, що вже були вирішені.

Поведінка і стан – інкапсуляція

Ми вже бачили, що компоненти характеризуються своєю поведінкою, тобто тим, що вони повинні робити. Але компоненти також зберігають деяку інформацію.

Поведінка компонента – це набір дій, ним здійснюваних. Повний опис поведінки компонента іноді називають протоколом.

Говорячи про стан компонента, мають на увазі його внутрішній зміст. Стан не є статичним і може змінюватися з часом.

ООП розглядає інкапсуляцію як головну мету. Добре розроблений об'єкт намагається інкапсулювати (поєднати) весь стан і поведінку, необхідні для виконання задачі, і в той же час ховає якнайбільше деталей внутрішньої будови.

2.4. Класи і методи

Ми використовували термін екземпляр для позначення представника класу. Відповідно ми будемо використовувати термін змінна екземпляра для позначення внутрішньої змінної, що міститься в екземплярі. Кожен екземпляр має свою власну сукупність перемінних. Ці значення не повинні змінюватися клієнтами прямо, а тільки за допомогою методів, асоційованих із класами.

Об'єкт є, таким чином, комбінацією стану і поведінки. Стан описується змінними екземпляра, у той час як поведінка характеризується методами. Зовні клієнти можуть довідатися тільки про поведінку об'єктів. Зсередини доступна повна інформація про те, як методи забезпечують необхідну поведінку, змінюють стан і взаємодіють з іншими об'єктами.

Різновиди класів

Класи в ООП мають кілька різних форм і використовуються для різних цілей. Наступні категорії охоплюють велику частину класів:

- управління даними;
- джерела даних чи посередники в передачі даних;
- класи для перегляду даних;
- допоміжні класи, або класи, що спрощують проектування.

Цей список не є вичерпним, однак він цілком підходить для навчальних цілей.

При проектуванні класів треба притримуватися так званих принципів Парнаса:

• Оголошення класу повинне забезпечувати клієнта всією інформацією, необхідною для успішної роботи з класом, і ніякою іншою.

• Методам класу повинна бути доступна вся інформація, необхідна для виконання їхніх обов'язків, і ніяка інша.

Описання класів у мовах Java і C++

У мові Java клас – основна структура даних, розгляд особливостей мови C++ виходить за рамки цього посібника і є частиною курсу "Основи програмування і алгоритмічні мови". Тому тут розглянемо лише відмінності при описанні класів у цих двох мовах.

• Оскільки відсутні препроцесор, глобальні змінні, та enumeration-типи даних, то символьні константи можуть бути створені шляхом опису й ініціалізації локальних змінних з використанням ключового слова **final**. Такі "термінальні" значення не можуть згодом змінюватися і тим самим виявляються еквівалентними символьним константам.

• Реалізація методів приводиться безпосередньо усередині визначення класу, а не будь-де в іншім місці. (Це дозволено в мові C++ як опцію, але є обов'язковим для мови Java.)

• Замість розбивки опису класу на **private** і **public** ці ключові слова приєднуються в явному виді до кожної змінної чи методу.

• За винятком конструкторів (які розпізнаються в силу того факту, що їхні імена збігаються з назвою класу) усі методи повинні мати значення, що повертається.

Крім того, зауважимо, що, хоча в Java дозволено змішувати оголошення змінних і методів, насправді, розробникам рекомендується розділяти описання методів і змінних в класах. Так, наприклад, стандарти Sun Microsystems рекомендують спочатку в класі описувати всі змінні, потім конструктори, і далі – всі інші методи. Проте автори робіт [7,8] та деякі середовища програмування рекомендують спочатку наводити конструктори, методи, і лише в кінці – змінні. Головне при цьому – єдність стилю: оберіть його для себе – і притримуйтесь постійно.

Приклади описань класів у мові Java були наведені у розділі 1.

2.5. Повідомлення, екземпляри й ініціалізація

У процесі пересилання повідомлень маємо три чітко виділювані компоненти: одержувач (об'єкт, якому посилається повідомлення), селектор повідомлення (текст, що ідентифікує конкретне повідомлення) і список

аргументів, що використовуються при реакції на повідомлення. Хоча ці поняття є центральними для ООП, проте термінологія і синтаксис, використовувані в різних мовах, варіюються в широких межах. У відповідності до направлення посібника докладніше розглянемо як вирішуються ці питання у мові Java.

Синтаксис пересилання повідомлень у мові Java

Синтаксис для пересилання повідомлень у мові Java майже ідентичний використовуваному в C++. Єдина помітна відмінність полягає в тому, що псевдозмінна **this** у мові C++ позначає вказівник на об'єкт, а в мові Java є посиланням на об'єкт (оскільки в мові Java немає вказівників!).

Синтаксична форма складається з імені одержувача, за яким стоїть крапка, селектора повідомлення (який повинний відповідати імені методу для класу одержувача) і списку аргументів (у круглих дужках).

Якщо theCard описаний як екземпляр класу Card, то наступний оператор наказує гральній карті відобразити себе в заданому вікні в точці з координатами 25 і 37:

```
theCard.draw(win, 25, 37);
```

Навіть якщо аргументи не потрібні, круглі дужки однаково необхідні, щоб відрізнити виклик метода від звертання до полів даних. Нижченаведений код визначає, чи лежить карта лицевою стороною догори:

```
if ( theCard.faceUp() ) {
        ...
}
else {
        ...
}
```

Компілятор перевіряє правильність пересилання повідомлення, звертаючись до опису класу одержувача. Спроба передати повідомлення, що не є частиною класу, відзначається як помилка при компіляції.

Метод класу, що описаний з типом **void**, може використовуватися як оператор (тобто як виклик процедури). Методи, що мають значення, яке повертається, інших типів, викликаються у відповідності зі стилем мови C і як оператори (процедури), і як функції (подібно звертанням до традиційних функцій).

З кожним методом асоційована псевдозмінна, у якій зазначений одержувач повідомлення. У мові Java вона називається **this** і є посиланням на реальну змінну з ім'ям одержувача.

Створення і ініціалізація об'єктів у мові Java

Основне розходження між Java і C++ (у відношенні питань, функціонування об'єктів) полягає в тому, що Java використовує автоматичне "збирання сміття", звільняючи тим самим програміста від необхідності займатися управлінням пам'яттю. Усі значення автоматично звільняються, коли вони стають недоступними для середовища виконання програми.

Усі змінні типу "об'єкт" у мові Java спочатку одержують значення **null**. Об'єкти створюються оператором **new**. На відміну від C++ у мові Java круглі дужки повинні використовуватися в операторі навіть тоді, коли не потрібно ніяких аргументів:

```
// створити п'ятірку списів
Card aCard = new Card (Card.spade, 5);
// створити карту за замовчуванням
Card bCard = new Card ();
```

Поняття конструктора в мові Java аналогічно C++ з тим виключенням, що конструктори в Java не підтримують ініціалізаторів. Крім того, треба мати на увазі, що всі об'єкти у мові Java розміщуються у динамічній пам'яті, і конструктори викликаються тільки разом з оператором **new**. На відміну від C++ конструктор у мові Java може викликати інші конструктори того ж об'єкта (з іншими параметрами), що найчастіше дозволяє виключити з декількох конструкторів загальні оператори (у мові C++ для цього потрібно було створювати окремий спеціальний метод). Конструктор при цьому повинний викликатися за допомогою ключового слова **this**:

```
class NewClass {
        NewClass(int i) {
                // виконати ініціалізацію 1-го типу
                ...
        }
        NewClass(int i, int j) {
                // перш за все викликати перший конструктор
                this(i);
                // продовжити ініціалізацію
                ...
        }
}
```

2.6 Відношення між класами

Між класами існують три звичайних відношення.

- Залежність ("використовує" – "uses-a")

- Агрегування ("містить" – "has-a")
- Наслідування ("є" – "is-a")

Відношення ***залежності*** (dependence – "uses-a") найбільш очевидне і поширене. Наприклад, клас Bird (Птах) використовує клас Bough (Гілка), оскільки птахи створюють гнізда на гілках дерев. Проте клас Wing (Крило) не залежить від класу Bough, оскільки об'єкти класу Wing ніколи не цікавляться станом гілок дерева. Таким чином, клас залежить від іншого класу, якщо його методи маніпулюють об'єктами цього класу.

> ***Зауваження.*** *При проектуванні програм намагайтеся мінімізувати кількість взаємозалежних класів. Якщо клас А не знає про існування класу В, він тим більше нічого не знає про будь-які його зміни! (Це означає, що будь-які зміни, внесені в клас В, не вплинуть на клас А.) У термінах програмного забезпечення це означає мінімізацію зв'язків (coupling) між класами.*

Відношення ***агрегування*** (aggregation – "has-a") достатньо просто зрозуміти, оскільки воно є цілком конкретним. Наприклад, об'єкт Bird містить об'єкти класу Wing. Агрегування означає, що клас А містить об'єкти класу В.

> ***Зауваження.*** *Деякі спеціалісти з методології нехтують концепцією агрегування і віддають перевагу використанню відношення "асоційованості" ("association"). З точки зору моделювання це розумно. Однак для програмістів відношення "містить" має більший сенс.*

Відношення ***наслідування*** (inheritance – "is-a") передає відношення між конкретним і більш загальним класом. Наприклад, клас Parrot (Папуга) є похідним від класу Bird (Птах). Клас Parrot має спеціальні методи (наприклад, "розмовляти, як людина"), у той самий час, як інші його методи (наприклад, "літати") наслідують методи класу Bird. Зокрема, якщо клас А розширює клас В, то кажуть, що клас А наслідує методи класу В, маючи, крім них ще й додаткові можливості. Більш докладно наслідування розлядається у наступному підрозділі.

Багато програмістів використовують символи мови UML (Unified Modeling Language) для зображення діаграм класів (class diagrams),

що описують відношення між класами. Приклад такої діаграми наведений на рис. 2.2

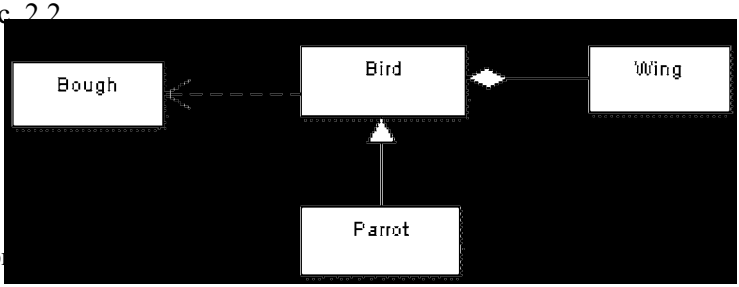

Рис. 2.2. Пр

Тут класи зображені за допомогою прямокутників, а відношення між ними – різними стрілками. За допомогою програм, що підтримують моделювання на мові UML, можна створити каркас програми на мові Java, чи на іншій об'єктно-орієнтованій мові про-грамування. В табл. 2.1 показані основні позначення, що прийняті в мові UML.

Таблиця 2.1. **Позначення, прийняті в мові UML для характеристики відношень між класами**

Відношення	Позначення в мові UML
Наслідування	
Залежність	
Агрегування	
Асоційованість	
Направлена асоційованість	

Зауваження. *Для створення діаграм на мові UML створено багато інструментів. Деякі постачальники пропонують дуже потужні (і дуже дорогі) програми, покликані стати основним засобом проектування. Відмітимо серед них програми Rational*

Rose (http://www.rational.com/products/rose/index.jsp) та Together (http://www.togethersoft.com/products/controlcenter/index.jsp). Як альтернативу можна назвати програму ArgoUML, що створена на принципах відкритого коду (http://argouml.tigris.org/). Якщо вам треба просто намалювати діаграму, приклавши мінімум зусиль, спробуйте застосувати програму Violet (http://horstmann.com/violet).

2.7. Наслідування

Першим кроком при вивченні ООП було усвідомлення задачі як взаємодії програмних компонентів. Наступним кроком у вивченні ООП стане організація класів у виді ієрархічної структури, заснованої на принципі наслідування. Під наслідуванням ми розуміємо можливість доступу представників дочірнього класу (підкласу) до даних і методів батьківського класу (суперкласу).

У мовах програмування наслідування означає, що поведінка і дані, зв'язані з дочірнім класом, завжди є розширенням (тобто більшою множиною) властивостей, зв'язаних з батьківськими класами. Підклас має усі властивості батьківського класу і, крім того, додаткові властивості. З іншого боку, оскільки дочірній клас є більш спеціалізованою (чи обмеженою) формою батьківського класу, він також, у визначеному змісті, буде звуженням батьківського класу. Це діалектичне протиріччя між наслідуванням як розширенням і наслідуванням як звуженням є джерелом великої сили, властивій даній техніці, і в той же час викликає деяку плутанину.

Наслідування завжди транзитивне, так що клас може унаслідувати риси суперкласів, що відстоять від нього на кілька рівнів. Наприклад, якщо собаки Dog є підкласом класу ссавців Mammal, а ссавці Mammal є підкласом класу тварин Animal, тоді клас собак Dog наслідує властивості і ссавців Mammal, і усіх тварин Animal (рис. 2.3).

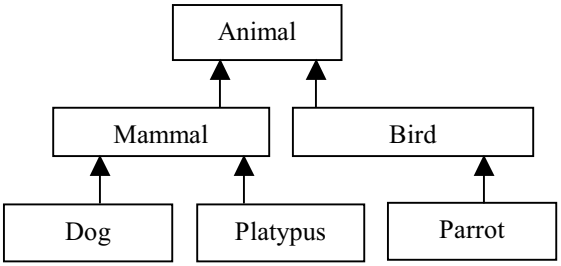

Рис. 2.3. Приклад ієрархії наслідування

Клас Animal є суперкласом. Це не означає, що він має перевагу над своїми підкласами чи має ширші функціональні можливості. *Насправді все навпаки*: підклас *ширше*, ніж суперклас. Наприклад, аналізуючи код класу Bird, легко бачити, що він інкапсулює більше даних і містить більше методів, ніж його суперклас Animal.

Примітка. *Префікси* ***супер-*** *та* ***під-*** *прийшли у програмування з теорії множин, що є частиною комп'ютерних наук і математики. Множина всіх тварин містить в собі множину всіх птахів. В цьому випадку кажуть, що множина тварин є* ***супермножиною*** *(superset) по відношенню до множини птахів. Іншими словами, множина всіх птахів є* ***підмножиною*** *(subset) множини всіх тварин.*

Ускладнюючою обставиною в нашому інтуїтивному описі наслідування є той факт, що підкласи можуть перевизначати поведінку наслідувану від батьківського класу. Наприклад, клас качкодзьобів Platypus перевизначає процедуру розмноження, наслідувану від класу ссавців Mammal, оскільки качкодзьоби відкладають яйця. У цьому розділі ми коротко торкнемося механізму перевизначення.

Розглянемо зв'язок між типом даних, зв'язаним з батьківським класом, і типом даних, зв'язаним з дочірнім класом. Можна стверджувати наступне:

- Представники підкласу повинні володіти всіма областями даних батьківського класу.
- Представники підкласу повинні забезпечувати виконання, принаймні через наслідування (якщо немає явного перевизначення), усіх функціональних обов'язків батьківського класу. Це не суперечить тому, що в нового класу можуть з'явитися додаткові обов'язки.
- Представник дочірнього класу може імітувати поведінку батьківського класу і повинний бути нерозрізненним від представника батьківського класу в подібних ситуаціях.

Трохи далі, розбираючи різні варіанти наслідування, ми побачимо що ці твердження не завжди вірні. Проте вони дають гарний опис ідеалізованого підходу до наслідування. Тому формалізуємо такий підхід у виді принципу підстановки.

Принцип підстановки *стверджує, що якщо є два класи A і B такі, що клас B є підкласом класу A (можливо, відстоїть в ієрархії на кілька ступіней), тоді ми повинні мати можливість підставити представника класу B замість представника класу A у будь-якій ситуації, причому без видимого ефекту.*

Форми наслідування

Можна виділити декілька основних форм наслідування. Наведемо їх у вигляді наступного списка [2]:

1. Спеціалізація. Дочірній клас є більш конкретним, приватним чи спеціалізованим випадком батьківського класу. Іншими словами, дочірній клас є підтипом батьківського класу. При породженні підкласу для спеціалізації новий клас є спеціалізованою формою батьківського класу, але задовольняє специфікаціям батька у всіх істотних моментах. Таким чином, для цієї форми цілком виконується принцип підстановки. Разом з наступною категорією (наслідування для специфікації) спеціалізація є найбільш ідеальною формою наслідування, до якої повинна прагнути гарна програма.

2. Специфікація. Батьківський клас описує поводження, що реалізується в дочірньому класі, але залишено нереалізованим у батьківському. Фактично це спеціальний випадок породження підкласу, що спеціалізує, за винятком того, що підкласи є не удосконаленням існуючого типу, а, скоріше, реалізацією неповної, абстрактної специфікації. У таких випадках батьківський клас іноді називають абстрактно специфікованим класом.

3. Конструювання. Дочірній клас використовує методи, надані батьківським класом, але не є підтипом батьківського класу (реалізація методів порушує принцип підстановки). У мовах зі статичними типами даних косо дивляться на породження підкласів для конструювання, оскільки воно часто прямо порушує принцип підстановки (з'являються підкласи, що не є підтипами). З іншого боку, будучи швидким і легким способом побудови нових абстракцій, воно широко використовується в мовах з динамічними типами даних.

4. Узагальнення. Дочірній клас модифікує чи перевизначає деякі методи батьківського класу з метою одержання об'єкта більш загальної категорії. Як правило, варто уникати породження підкласу для узагальнення, користаючись перевернеою ієрархією типів і породженням для спеціалізації. Однак це не завжди можливо.

5. Розширення. Дочірній клас додає нові функціональні можливості до батьківського класу, але не змінює наслідуване поводження. Оскільки функціональні можливості батька залишаються недоторканими і доступними, породження підкласу для розширення не суперечить принципу підстановки і, отже, такі підкласи завжди будуть підтипами.

6. Обмеження. Дочірній клас обмежує використання деяких ме-

тодів батьківського класу. Оскільки породження підкласу для обмеження є явним порушенням принципу підстановки й оскільки воно будує підкласи, що не є підтипами, його варто уникати, де тільки можливо.

7. Варіювання. Дочірній і батьківський класи є варіаціями на одну тему, і зв'язок "клас-підклас" довільна. Звичайно, однак, кращою альтернативою є виділення загального коду в абстрактний клас і породження обох класів від цього загального предка.

8. Комбінування. Дочірній клас наслідує риси більш ніж одного батьківського класу. Це – множинне наслідування; що є досить проблемним питанням і реалізується у мові Java за допомогою інтерфейсів.

Наслідування у мові Java

Java йде далі усіх у поділі понять підкласу і підтипу. Підкласи з'являються за допомогою ключового слова extends (розширює). Це ключове слово означає, що створюється новий клас, похідний від існуючого. Існуючий клас називається суперкласом, базовим або батьківським, а знову створюваний – підкласом, похідним або дочірнім. Серед програмістів на мові Java найбільш широко розповсюджені терміни "суперклас" та "підклас".

Приклад 2.1 Запис наслідування у Java

```
class Window {
        // . . .
}
class TextEditWindow extends Window {
        // . . .
}
```

Передбачається, що підкласи є підтипами (хоча це припущення не завжди вірне). Це означає, що представник підкласу може бути присвоєним змінній, оголошеній з типом батьківського класу. Методи дочірнього класу, що мають ті ж імена, що і методи батька, перевизначають наслідувану поведінку. Ключове слово **protected** може бути використане для вказівки методів і даних, доступних тільки усередині класу чи підкласу, але не вхідних у більш загальний інтерфейс.

Усі класи походять від єдиного предка Object. Якщо батьківський клас не зазначений явно, то передбачається клас Object.

Альтернативна форма породження підтипів даних базується на інтерфейсі. Інтерфейс визначає протокол визначеної поведінки, а не реа-

лізацію. У цьому відношенні він подібний абстрактному батьківському класу. У наступному прикладі приведений інтерфейс, що описує об'єкти, які можуть читати і писати у вхідний/вихідний потік.

Приклад 2.2 Запис наслідування на основі інтерфейсу

```
public interface Storing {
        void writeOut(Stream s);
        void readFrom(Stream s);
}
```

Інтерфейс визначає новий тип. Це означає, що змінні можуть бути оголошені просто з ім'ям інтерфейсу. А клас може вказати, що він реалізує протокол, визначений інтерфейсом. Представники класу можуть присвоюватися змінним, оголошеним з типом інтерфейсу, точно так само, як представники дочірнього класу можуть присвоюватися змінним, оголошеним з типом батьківського класу:

Приклад 2.3 Використання інтерфейсу

```
public class BitImage implements Storing {
        void writeOut (Stream s) {
                // . . .
        }
        void readFrom (Stream s) {
                // . . .
        };
};
```

Хоча мова Java підтримує тільки одиночне наслідування (наслідування винятково від одного батьківського класу), клас може вказувати, що він підтримує кілька інтерфейсів (реалізує множинний інтерфейс). Багато проблем, для яких у мові C++ довелося б використовувати множинне наслідування, у мові програмування Java вирішуються через множинні інтерфейси. Інтерфейсам дозволено розширювати інші інтерфейси, у тому числі і множинні, через указівку ключового слова **extends**.

У мові Java ідея породження підкласу для специфікації формалізована через модифікатор **abstract**. Якщо клас оголошений як **abstract**, то з нього повинні породжуватися підкласи. Не дозволяється створювати представників абстрактного класу, можна тільки породжувати підкласи. Методи теж можуть бути оголошені як **abstract**, і в такому ви-

падку вони не зобов'язані мати реалізацію. Таким чином, оголошення класу як **abstract** забезпечує, що він буде використовуватися тільки як специфікація поведінки, а не у вигляді конкретних об'єктів:

Приклад 2.4 Оголошення абстрактного класу

```
abstract class Storable {
        public abstract writeOut();
}
```

Навпаки, модифікатор **final** указує, що з класу не можуть породжуватися підкласи, або метод, до якого відноситься цей модифікатор, не може бути змінений. Тим самим користувачу гарантується, що поведінка класу буде такою, яка визначена, і не буде модифікована при породженні наступних підкласів:

Приклад 2.5 Оголошення класу, що не дозволяє наслідування

```
final class NewClass extends OldClass {
        ...
}
```

2.8. Поліморфізм

Слово поліморфізм грецького походження й означає приблизно "багато форм" (poly – багато, morphos – форма). Слово morphos має відношення до грецького бога сну Морфею (Morphus), що міг являтися сплячим людям у будь-якій формі, у якій тільки побажає, і, отже, був воістину поліморфним. У біології поліморфні види – це ті (на зразок Homo Sapiens), що характеризуються наявністю різних форм чи розцвічень. У хімії поліморфні з'єднання можуть кристалізуватися принаймні в двох різних формах (наприклад, вуглець має три кристалічні форми – графіт, діамант та нещодавно винайдений фулерен). У мовах програмування поліморфний об'єкт – це сутність (змінна, аргумент функції), що зберігає під час виконання програми значення різних типів. Поліморфні функції – це ті функції, що мають поліморфні аргументи.

В об'єктно-орієнтованих мовах програмування поліморфізм – природний наслідок:

- відносини "бути екземпляром";
- механізму пересилання повідомлень;
- наслідування;
- принципу підстановки.

Одне з найважливіших достоїнств об'єктно-орієнтованого підходу складається в можливості комбінування цих засобів. У результаті вихо-

дить багатий набір технічних прийомів спільного і багаторазового використання коду. Чистий поліморфізм має місце, коли та сама функція застосовується до аргументів різних типів. У випадку чистого поліморфізму є одна функція (тіло коду) і декілька її інтерпретацій. Інша крайність спостерігається, коли маємо безліч різних функцій (тобто тіл коду) з одним ім'ям. Така ситуація називається перевантаженням, або поліморфізмом ad hoc. Між цими двома полюсами лежать методи, що перевизначаються та відкладені методи.

Перевантаження

Ми говоримо, що ім'я функції перевантажене, якщо маємо два (і більш) коди, зв'язані з цим ім'ям. Помітьте, що перевантаження є обов'язковою частиною перевизначення методів, але ці два терміни не ідентичні, і перевантаження може відбуватися без перевизначення. При перевантаженні поліморфним є ім'я функції – воно багатозначне. Ще один спосіб уявити собі перевантаження і поліморфізм: уявіть, що є єдина абстрактна функція, що викликається з аргументами різного типу, а фактичний виконуваний код залежить від типу аргументів. Той факт, що компілятор часто може визначити правильну функцію на етапі компіляції (у мовах зі строгим контролем типів даних) і, отже, генерувати тільки потрібний код – це просто оптимізація.

Як приклад близький до мов програмування, розглянемо розробку бібліотеки класів, що представляють структури даних загального виду. Ряд структур (множини, набори, словники, черги з урахуванням пріоритетів) може бути використаний для збереження сукупності елементів. Для кожної структури буде визначений метод add, що додає до неї новий елемент. Отже, дві зовсім різні функції використовуються для виконання семантично аналогічних дій над різними типами даних. Така ситуація часто зустрічається в програмуванні, і не тільки об'єктно-орієнтованому.

Інший стиль перевантаження, при якому процедурам (функціям, методам) у тому самому контексті дозволяється використовувати спільно одне ім'я, а двозначність знімається за рахунок аналізу числа і типів аргументів, називається параметричним перевантаженням. Вона присутня в C++ і Java, а також у деяких директивних мовах (наприклад, Ada) і в багатьох мовах, заснованих на функціональній парадигмі. Ми вже бачили приклади такого перевантаження для функцій-конструкторів.

Перевантаження присутнє у всіх інших формах поліморфізму, що ми розглядаємо: перевизначення, відкладені методи, чистий поліморфізм. Воно також часто корисне при "звуженні концептуального простору", тобто при зменшенні кількості інформації, яку необхідно пам'ятати програмісту. Часто ця турбота про пам'ять програміста не менш важлива, ніж

зниження вимог до пам'яті комп'ютера, що досягається при спільно використовуваному коді.

Перевизначення

Наведемо приклад істотних елементів цієї техніки. В одному з класів (звичайно в абстрактному надкласі) є визначений для конкретного повідомлення загальний метод, що наслідується і використовується підкласами. Однак, принаймні в одному підкласі, визначений метод з тим же ім'ям. Це перекриває доступ до загального методу для екземплярів даного підкласу (чи ж у випадку уточнення робить доступ до методу багатоступінчастим). Ми говоримо, що другий метод перевизначає перший. Перевизначення часто відбувається прозоро для користувача класу, і, як і у випадку перевантаження, дві функції представляють семантично як одну сутність.

Поліморфізм у Java

Мова Java підтримує як ієрархію підкласів (із ключовим словом **extends**), так і ієрархію підтипів (із ключовим словом **implements**). Змінні можуть бути оголошені чи через клас, чи через інтерфейс. Усі змінні є поліморфними. Змінна, описана як клас, може містити значення, що відносяться до будь-якого підкласу. Змінна, оголошена як інтерфейс, може зберігати значення будь-якого класу, що реалізує цей інтерфейс.

Відкладені методи реалізуються в мові Java через ключове слово **abstract**. Методи, описані як **abstract**, не мають тіла; вони закінчуються крапкою з комою. Абстрактні методи повинні обов'язково перевизначатися в підкласах. Клас, що містить у собі абстрактний метод, повинний у свою чергу бути описаний як абстрактний. Не можна створити екземпляри абстрактних класів.

Приклад 2.6 Використання абстрактного класу

```
abstract class Shape {
        // нижче повинно перевизначатися
        public abstract draw();
        // ...
}
class Triangle extends Shape {
        public draw() {
                // намалювати трикутник
        }
        // ...
}
```

Пригадаємо, що цікавою властивістю мови Java є модифікатор **final**, що у деякому змісті протилежний ключовому слову **abstract**. Клас чи метод, описаний як **final**, не може породжувати підкласи або перевизначитися.

Змінні класу в Java.

Мова Java використовує ключове слово **static** для позначення змінної класу. Поля даних і методи, описані в класі, можуть бути позначені як **static**, тоді вони будуть застосовні до самого класу, а не до його екземплярів. Статичні змінні можуть мати ініціалізатори, що виконуються при завантаженні визначення класу. Крім того, блок коду дозволяється позначати як **static**, тоді він також буде виконуватися під час завантаження. Наступний приклад ілюструє це. Тут описується статичний масив значень, що ініціалізується числами від 1 до 12:

Приклад 2.6 Використання статичних полів даних і блоків

```
class statTest {
        static final size = 12;
        static int arr[] = new int[size]; // об'явити масив
        static {
            // ці команди виконуються при завантаженні класу
            for (int i = 0; i < arr.length; i++) {
                arr[i] = i + 1;
            }
        }
}
```

Комбінація ключових слів static і final створює ініціалізоване поле, значення якого не може бути зміненим.

Як статичні змінні, так і статичні методи доступні через ім'я класу. Однак для зручності до них можна звернутися і через екземпляр класу (хоча це і не рекомендовано розробниками мови Java).

Заміщення у мові Java

У мові програмування Java, на відмінність від мов C++ і Object Pascal, немає необхідності повідомляти про перевизначення методу. Досить того, що новий метод має те ж саме ім'я, список аргументів і тип значення, що повертається, що і метод батьківського класу.

У мові Java можливо навіть заміщення полів даних (чого немає у C++ і Object Pascal). Однак такі заміщення не є динамічними, і при використанні поля даних його тип буде визначатися оголошеним (статичним) класом змінної, а не її динамічним значенням. Ця ситуація іноді описується як маскування в дочірньому класі змінної батька.

Як уже йшлося, у мові Java є цікава властивість – ключове слово **final**. Якщо воно застосовується до імені методу, то означає, що метод є "листом" ("термінатором") в ієрархічному дереві класу і не може бути надалі перевизначений будь-яким способом. Якщо це ключове слово зустрічається у визначенні класу, то це означає, що з класу не можуть породжуватися підкласи. Компілятору мови Java дозволене оптимізувати методи, описані за допомогою ключового слова **final**, перетворюючи їх в inline-функцію і підставляючи її явний код у точку виклику.

Висновки

В даному посібнику було розглянуто основні властивості об'єктно-орієнтованого підходу до програмування на основі мови Java. Звичайно, такий посібник не може розглядатися як підручник з мови Java, чи з ООП. Для успішного опанування цієї мови і ООП, треба опрацювати багато літератури і написати не один десяток програм. Автор сподівається, що тоді Ви зможете переконатися (як і він декілька років тому), що Java – одна з найкращих мов програмування, а ООП – найкращий підхід до програмування.

Установка та налаштування засобів програмування мовою Java

Засоби програмування мовою Java традиційно складаються з двох великих частин:

- Компілятор, середовище виконання, утиліти командного рядка, файли електронної документації (Java Software Development Kit – Java SDK)
- Середовища розробки

Для установки більшості з сучасних засобів розробки треба спочатку встановити Java SDK, після цього треба встановити середовище розробки.

1. Для установки Java SDK v 6.0 в операційних системах родини Windows запустіть на виконання інсталяційний пакет **jdk-6u3-windows-i586-p.exe** (його можна отримати з файлового архіву університету, або завантажити з сайту розробника – Sun Microsystems мережі Internet: http://java.sun.com) Для установки Java SDK в операційних системах родини Linux треба скористатися інсталяційним пакетом **jdk-6u3-linux-i586.bin** (отримати його можна там же)

2. Для комфортної роботи з Java SDK розпакуйте файли електронної документації з архіву **jdk-1_6_0-doc.zip** його можна також отримати з сайту розробника, чи з файлового архіву університету (Бажано розпаковувати в той самий каталог, котрий було створено при інсталяції компілятора, наприклад, якщо компілятор установлено в каталог *C:\Program Files\Java\jdk1.6.0*, то розпаковувати архів треба в нього, при цьому там буде створено підкаталог docs)

3. Після того треба встановити одне з середовищ програмування. Для Windows можна використовувати середовище BlueJ (про роботу з ним див. Додаток Б.) Також можна використовувати більш складні середовища NetBeans, або JBuilder. В процесі інсталяції, якщо з'явиться запит, вкажіть папку в якій установлено компілятор і система довідки.

Після інсталяції пакета Java SDK треба зробити ще один крок: додати ім'я каталогу jdk/bin до списку каталогів, де операційна система може знайти виконувані файли.

На платформі Windows 9x додайте рядок, що наведений нижче в кінець файлу AUTOEXEC.BAT:

```
SET PATH="C:\Program Files\Java\jdk1.6.0\bin";%PATH%
```

На платформі Windows NT/200/XP відкрийте панель управління, оберіть піктограму "Система" там у вкладці "Дополнительно" натисніть

кнопку "Переменные среды". Оберіть PATH у вікні User Variables (Переменные пользователя) та додайте у початок цього рядка ім'я каталогу jdk/bin, наприклад так:

"C:\Program Files\Java\jdk1.6.0\bin";інше-те що було раніше

Збережіть свої установки.

В операційних системах Linux треба виконати аналогічні дії в залежності від встановленої оболонки.

Для перевірки правильності виконаних дій зробіть таке:

Відкрийте вікно оболонки (WinXP – Пуск-Выполнить-cmd.exe), у ньому введіть такий рядок:

java -version

та натисніть <ENTER> На екрані повинно з'явитися таке:

java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode)

якщо замість цього повідомлення з'явився рядок типу "java:command not found" треба повернутися і ще раз перевірити, чи правильно виконано інсталяцію.

Додаток Б

Робота з IDE BlueJ

Створення і виконання Java-програм у середовищі BlueJ

1. Створіть новий проект, для цього:

• Після запуску BlueJ у головному меню програми оберіть:

Проект -> Новый проект… / Project -> New Project…

• У вікні, що відкриється, введіть ім'я проекту (наприклад, **First**), та натисніть мишею кнопку **"Создать / Create"**.

2. У головному вікні середовища BlueJ, що відкриється при цьому, натисніть мишею кнопку **"Новый класс… / New Class…"**.

3. У діалоговому вікні, що відкриється, введіть ім'я класу, що створюється (наприклад, Lab1), та оберіть один зі стандартних шаблонів класів (наприклад, **"Класс с методом main(…)"**, або **"Класс"**). Підтвердьте свій вибір натисканням **"Ok"**.

4. Клацніть правою кнопкою миші по зображенню класу та у контекстному меню, що з'явиться оберіть **"Открыть редактор / Open Editor"**.

5. Введіть текст вашої програми. При цьому можна змінювати код, що був згенерований середовищем, додавати свої змінні і методи, видаляти код, створений за шаблоном.

6. Якщо клас призначений для виконання у автономному режимі, не забудьте додати метод **public static void main(String[ ] s)** (див. структуру Java-програми).

7. Для компіляції класу оберіть у контекстному меню класу пункт **"Компилировать / Compile"**. Зверніть увагу, що клас, код якого змінився з моменту останньої компіляції зображується заштрихованим. Після успішної компіляції штрихування зникає.

8. Створіть об'єкт скомпільованого класу. Для цього у контекстному меню оберіть виклик конструктора класу: **new Lab1** (або new <Ім'я вашого класу>). Підтвердьте створення об'єкта, для цього введіть його ім'я та натисніть кнопку **"Ok"** у діалоговому вікні, що відкриється. При цьому знизу вікна з'явиться умовне зображення об'єкта.

9. За допомогою контекстного меню об'єкта викликайте його методи, виконуйте його інспекцію та ін.

10. Якщо клас створювався, як такий, що може виконуватися автономно, то його можна виконати після компіляції, обравши у контекстному меню класу метод **main**.

Примітка. *Для запуску автономної програми на мові Java можна перейти в каталог, де знаходиться скомпільований код програми (файл з розширенням Java) та виконати таку команду (у режимі командного рядка):*

java <ім'я_класу_без_розширення>

Наприклад, java Lab1

Література

1. *Арнольд К., Гослинг Дж., Холмс Д.* Язык программирования Java. 3-е изд. – М.: Издательский дом "Вильямс", 2001.

2. *Бадд Т.* Объектно-ориентированное программирование в действии. – СПб.: Питер, 1997.

3. *Бишоп Д.* Эффективная работа: Java 2. – СПб.: Питер; К.: Издательская группа BHV, 2002.

4. Д*арвин Я., Java.* Сборник рецептов для профессионалов. – СПб.: Питер, 2002.

5. *Синтес А.* Освой самостоятельно объектно-ориентированное программирование за 21 день. – М.: Издательский дом "Вильямс", 2002.

6. *Эккель Б.* Философия Java. Библиотека программиста. – СПб.: Питер, 2001.

7. *Хабибуллин И.* Самоучитель Java 2. СПб.: BHV, 2007.

8. *Хорстманн К.С., Корнелл Г.* Библиотека профессионала. Java 2. Том 1. Основы. – М.: Издательский дом "Вильямс", 2004.

9. *Хорстманн К.С., Корнелл Г.* Библиотека профессионала. Java 2. Том 2. Тонкости программирования. – М.: Издательский дом "Вильямс", 2004.

10. http://java.sun.com – матеріали сайту компанії Sun Microsystems, Inc, що стосуються мови програмування Java.

11. http://www.BruceEckel.com – матеріали сайту Брюса Еккеля та фірми MindView, що спеціалізується на навчанні об'єктно-орієнтованим технологіям програмування.

ЗМІСТ