### 3.3. Ієрархія класів

#### 3.3.1. Основи наслідування

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

У мові *Java* оголошення класу, який є спадкоємцем базового класу, матиме вигляд *class ім’я\_підкласу extends ім’я\_суперкласу {*

*// тіло класу*

*}*

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

*class A {*

*int i,j;*

*void showij( ){*

*System.out.println ("i та j:" +i +", "+j);*

*}*

*} // завершення класу А // створення підкласу*

*class B extends A {*

*int* k*;*

*void show*k*( ){*

*System.out.println ("*k*:" +*k*);} void sum( ){*

*System.out.println ("i + j + k:" +(i + j + k)); }*

*} // завершення класу В class SimpleІnheritance { public static void main(String args[ ]){*

1. *superOb=new A( );*
2. *subOb=new B( );*

*// використання базового класу безпосередньо superOb.i = 10; superOb.j = 20; superOb.showij();*

*// підклас має доступ до public членів базового класу subOb.i=7; subOb.j=8; subOb.k=9; subOb.showij( ); subOb.showk( );*

*subOb.sum( );*

*} }*

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

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

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

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

*// Клас, який є підкласом Box class BoxWeight extends Box { double weight;*

*// конструктор BoxWeight*

*BoxWeight(double w, double h, double d, double m){ width=w; height=h; depth=d;*

*weight=m;*

*}*

*} // завершення класу BoxWeight*

*class DemoBoxWeight {*

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

*BoxWeight mybox= new BoxWeight(10,20,15,34.3); double vol;*

*vol=mybox1.volume( );*

*System.out.println ("Об’єм:" +vol );*

*}*

*}*

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

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

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

Підклас може викликати конструктор базового класу за формою:

*super (список\_параметрів);*

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

Дещо краща версія класу *BoxWeight* може бути такою:

*// використання оператора super( ) class BoxWeight extends Box { double weight;*

*BoxWeight( double w, double h, double d, double m){*

*// виклик конструктора базового класу*

*super(w, h, d);*

*weight= m;*

*} }*

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

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

*BoxWeight (BoxWeight ob){ super(ob);*

*weight =ob.weight;*

*}*

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

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

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

*super.елемент;*

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

*class A { int i; } class B extends A {*

*int i; // ця змінна і перекриває і в А*

*B(int a, int b){ super.i=a; // і в класі А i=b; // і в класі B*

*}*

*void show( ){*

*System.out.println ("і в базовому класі:"+super.i);*

*System.out.println ("і в підкласі:"+i);*

*}*

*}// завершення класу В*

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

*subOb.show ();*

*}*

*}*

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

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

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

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

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

*final void meth( ) {*

*System.out.println("Цей метод не можна перекривати");*

*}*

*}*

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

З іншого боку, методи, оголошені як *final*, дають змогу покращити реалізацію програми. Компілятор організовує звертання до них як до виклику вбудованого коду (фактично заміняє виклик методу на копію реального коду, який є в цьому методі), тому що вони не будуть перекриватися в підкласі. Отже, ліквідовуються накладні витрати, спричинені викликом звичайного методу.

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

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

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

#### 3.3.2. Поліморфізм

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

Зауважимо, що перекривання методів відбувається тільки тоді, коли цілковито збігаються ім’я, типи і кількість параметрів. Інакше це зводиться до перевантаження методів. Тобто для підкласів викликають або його метод, або метод базового класу.

Залежно від того, які параметри задано під час виклику.

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

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

*}*

*class B extends A { void callme( ){ System.out.println ("метод callme класу B"); }*

*}*

*class C extends A {*

*void callme( ){ System.out.println ("метод callme класу C"); }*

*}*

*class Dispatch { public static void main(String args[ ]){*

1. *a=new A( );*
2. *b=new B( );*
3. *c=new C( );*

*A r; // посиляння на тип A r=a;*

*r.callme( ); // виклик методу для об’єкта класу А r=b;*

*r.callme( ); // виклик для об’єкта класу B r=c;*

*r.callme( ); // виклик для об’єкта класу C*

*} }*

У цьому прикладі змінну *r* визначено як змінну класу *А* − базового класу для класів *В,С*. Як бачимо, викликається версія методу *callme()*, визначена типом об’єкта. Якби все визначалося типом посилання *r*, то завжди викликався б метод *callme()* класу *А*.

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

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

#### 3.3.3. Абстрактні класи

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

*abstract тип ім’я\_методу(список\_параметрів);*

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

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

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

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

*// Використання абстрактних методів і класів*

*abstract class Figure { double dim1, dim2;*

*Figure (double a; double b){ // конструктор класу dim1= a; dim2= b;*

*}*

*abstract double area( ); // абстрактний метод*

*}*

*class Rectangle extends Figure { //прямокутник*

*Rectangle (double a; double b){ super (a, b);*

*}*

*double area( ) { return dim1 \* dim2;*

*}*

*}*

*class Triangle extends Figure { //трикутник*

*Triangle (double a; double b){ super (a, b);*

*}*

*double area( ) { return dim1 \* dim2/2;*

*}*

*}*

*class Abstract Areas {*

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

*Rectangle r=new Rectangle(9,5);*

*Triangle t=new Triangle(10,8);*

*Figure figref ; // оголошення посилання*

*// без створення об’єкта*

*figref = r;*

*System.out.println("Площа прямокутника:"+figref.аrea( )); figref = t;*

*System.out.println("Площа трикутника:"+figref.аrea( ));*

*}*

*}*

#### 3.3.4. Інтерфейси

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

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

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

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

***Визначення інтерфейсу.*** Загальний вигляд визначення інтерфейсу:

*доступ interfase ім’я { тип ім’я методу 1 (список\_параметрів); тип ім’я методу 2 (список\_параметрів); тип змінна\_константа 1=значення; тип змінна\_константа 2=значення;*

*//…*

*тип ім’я методу N (список\_параметрів); тип змінна\_константа N=значення; }*

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

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

*доступ class ім’я\_класу [extends суперклас] implements інтерфейс [, інтерфейс] {*

*//тіло класу*

*}*

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

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

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

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

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

ключового слова *extends*. Наприклад, *interfase B extends A [,C] {*

*//тіло інтерфейсу*

*}*

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

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

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

#### 3.3.5. Внутрішні класи

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

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

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

*public interface Destination { String readLabel();*

*}*

У файлі *Parcel4.java* розміщено програмний код:

*// визначення класу в тілі методу public class Parcel4 { public Destination dest(String s) { // початок методу*

*// початок внутрішнього класу*

*сlass PDestination implements Destination { private String label;*

*private PDestination(String whereTo) { label = whereTo;*

*}*

*// завершення конструктора внутрішнього класу*

*public String readLabel()*

*// реалізація методу інтерфейсу*

*{ return label; }*

*} // завершення внутрішнього класу*

*PDestination return new PDestination(s); } // завершення методу dest*

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

*Parcel4 p = new Parcel4();*

*Destination d = p.dest("Танзанія");*

*}*

*} // завершення класу Parcel4*

У наведеному коді метод *dest* повертає посилання на інтерфейс *Destination,* однак чітко тип об’єкта визначити неможливо (клас *PDestination* ззовні методу невідомий). Незважаючи на те, що клас *PDestination* оголошено у методі, в якому також створюється об’єкт цього класу, цей об’єкт буде доступним і після виходу з методу (через посилання на інтерфейс).

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

*public interface Contents { int value();*

*}*

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

*// Метод повертає внутрішній клас без імені public class Parcel6 { public Contents cont() { return new Contents() { private int i = 11; public int value() { return i; }*

*}; // Крапка з комою засвідчує*

*// про закінчення оператора*

*} //завершення методу cont public static void main(String[ ] args) {*

*Parcel6 p = new Parcel6();*

*Contents c = p.cont();*

*}*

*} // завершення класу*

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

*class MyContents() implements Contents { private int i = 11;*

*public int value() { return i; } }*

*return new MyContents();*

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

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

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

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

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

#### 3.3.6. Відображення у програмах відношень між об’єктами

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

− є (*is a*) – папуга є птахом;

− має (*has a*) – папуга має крила;

− використовує (*uses a*) – папуга використовує сідало; − створює (*creates a*) – папуга створює (несе) яйця.

***Програмування відношення "є" між класами.*** Відношення "є" встановлюється між двома класами і вказує на те, що один з класів є особливим (спеціалізованим) типом іншого класу. Головним способом реалізації відношення "є" є створення підкласу. Наприклад, *public class Parrot extends Bird {…}*

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

***Програмування відношення типу "має".***

Відношення типу "має" інколи називають відношенням типу "є частиною" або "використовується для реалізації". Якщо класи *А* і

*В*, з’єднані відношенням "має" і клас *В* використовується декількома іншими класами, то його необхідно реалізувати у власному файлі класу, а в опис класу *А* розмістити покажчик на екземпляр класу *В*. Отже, якщо в проекті необхідно відобразити, що і птахи, і жуки, і кажани мають крила, то клас *В* треба оголосити так:

*public class Bird { private Wing leftWing;*

*private Wing rightWing;*

*…*

*}*

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

*public class Bird { class Wing { ... } private Wing leftWing; private Wing rightWing;*

*… }*

***Програмування відношення "використовує"***. Відношення "використовує" реалізують шляхом передачі об’єкта одного класу в методи іншого класу. Наприклад, щоб дати змогу папузі відпочити, можна записати такі оператори: *Perch aPerch = new Perch(); theParrot.land (aPerch);*

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

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

Наприклад:

*public class Parrot { ...*

*public boolean isSick(Veterinariаn theVet) { return theVet.check(this); }*

*}*

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

*Egg layEgg(){*

*Egg theEgg = new Egg();*

*return theEgg; }*

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

#### 3.3.7. Клас Object

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

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

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

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

Методи *getClass(), notify(), notifyAll() і wait()* оголошені в класі *Object* як *final*. Інші методи можуть бути перекриті в підкласах. У нових класах бажано перекривати методи *equals()* i *toString()* для врахування їх особливостей. Метод *toString()* викликається автоматично при виведенні інформації про об’єкт методом *println()*.

#### 3.3.8. Пакети: сховище класів

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

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

− один оператор пакета (не обов’язково);

− довільну кількість операторів імпорту (не обов’язково);

− одне оголошення загальнодоступного класу (обов’язково);

− довільну кількість *private*-класів у цьому пакеті (не обов’язково).

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

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

*package ім’я\_пакета;*

Усі класи, оголошені в цих файлах, будуть зачислені до цього пакета. Якщо оператор *package* відсутній у файлі, то імена класів цього файла, за замовчуванням, належать до пакета без імені. Існує домовленість, що для запису імен пакетів використовують лише малі літери.

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

*package mypackage;*

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

Можна створювати ієрархію пакетів, задаючи оператор:

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

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

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

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

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

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

*import pkg1 [.pkg2].(ім’я\_класу |\*);*

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

*import java.io.\*;*

На відміну від оператора *#include* мови *С++*, оператор *import* не пов’язує з програмою тих класів пакета, які в ній не використовуються. Символ "\*" впливає тільки на час компіляції, а на час виконання і розмір класу абсолютно не впливає.

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

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

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

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

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

− підкласи у тому ж пакеті;

− не підкласи у тому ж пакеті;

− підкласи у різних пакетах;

− класи, які не є підкласами і перебувають в інших пакетах.

Механізм керування доступом можна розглядати спрощено:

все, що оголошено як *public*, доступно всюди; все, що оголошено як *private* – за межами класу невидиме.

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

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

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

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

− кожен клас (за винятком внутрішніх) розмістити в окремий власний файл з тим же іменем і розширенням .*java;*

− створити кореневу директорію, в якій міститимуться усі ваші пакети;

− якщо до складу проекту зачислено понад 5−9 класів, бажано розмістити їх у підпакетах.

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

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

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

*Package pkgs[ ];*

*// повертає всі пакети,*

*// які в даний момент відомі програмі pkgs = Package.getPackages()*

*for (int i=0; i < pkgs.length(); i++)*

*System.out.println ( pkgs[i].getName ()+ " " + pkgs[i].getImplementationTitle()+" "*

*+pkgs[i].getImplementationVendor()+ " "*

*+pkgs[i].getImplementationVersion());*

*}*

*}*

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