# ПАКЕТИ І ІНТЕРФЕЙСИ

Лекція 5.1

доц. кафедри Інформатики
Сінельнікова Т.Ф.

ХНУРЕ, Кафедра Інформатики

informatika@kture.kharkov.ua

- Пакети. Визначення пакета
- Пакет и підпакет
- Використання CLASSPATH
- Захист і керування доступом
- Імпорт пакетів
- Інтерфейси
- Визначення інтерфейсу
- Реалізація інтерфейсів
- Приклад класу, який реалізує інтерфейс
- Приклад виклику методу
- Часткові реалізації
- Застосування інтерфейсів
- Розширення інтерфейсів (приклад)

# Пакети. Визначення пакета

- Пакети - це контейнери для класів, які використовуються для збереження простору імен класів розділеним на іменовані області.
- Наприклад, ви маєте можливість створити клас з ім'ям List, який можна зберігати у вашому власному пакеті, не побоюючись, що він зіткнеться з деяким іншим класом з ім'ям List, що зберігаються десь в іншому місці.
- Пакети зберігаються ієрархічним способом і явно імпортуються в визначення нових класів.

# Пакет и підпакет

- Розробники Java включили в мову додаткову конструкцію - пакети (packages). Всі класи Java розподіляються по пакетах. Крім класів пакети можуть включати в себе інтерфейси і вкладені підпакети (subpackages).
- Утворюється деревовидна структура пакетів і підпакетів. Ця структура в точності відображається на структуру файлової системи. Всі файли з розширенням class (містять байт-коди), що утворюють пакет, зберігаються в одному каталозі файлової системи. Підпакети зібрані в підкаталоги цього каталога.
- Кожен пакет утворює одне простір імен (namespace). Це означає, що всі імена класів, інтерфейсів і підпакетів в пакеті повинні бути унікальні. Імена в різних пакетах можуть збігатися, але це будуть різні програмні одиниці.

# Пакет и підпакет

- Таким чином, жоден клас, інтерфейс або підпакет не може виявитися відразу в двох пакетах. Якщо треба використовувати два класи з однаковими іменами з різних пакетів, то ім'я класу уточнюється ім'ям пакету: пакет.класс. Таке уточнене ім'я називається повним ім'ям класу (fully qualified name).
- Всі ці правила збігаються з правилами зберігання файлів і підкаталогів в каталогах.
- Пакетами користуються ще й для того, щоб додати до вже наявних прав доступу до членів класу private, protected і public ще один, "пакетний" рівень доступу.
- Якщо член класу не відмічений жодним з модифікаторів private, protected, public, то, за умовчанням, до нього здійснюється пакетний доступ (default access), а саме, до такого члену може звернутися будь-який метод будь-якого класу з того ж пакета. Пакети обмежують і доступ до класу цілком - якщо клас не помічений модифікатором public, то всі його члени, навіть відкриті, public, не буде видно з інших пакетів.

# Пакети

- У загальному випадку вихідний файл Java може містити будь-яку (або все) з наступних чотирьох внутрішніх частин:
- одиночний package-оператор (не обов'язково); будь-яке число import-операторів (не обов'язково);
- одиночне оголошення загального класу (потрібно);
- будь-яке число приватних класів пакета (не обов'язково).

# Визначення пакета

- Створити пакет дуже легко: просто включіть оператор package в початок вихідного файлу Java. Будь-які класи, оголошені в межах того файлу, будуть належати вказаною пакету. Оператор package визначає простір імен, в якому зберігаються класи. Якщо ви опускаєте інструкцію package, імена класу поміщаються в пакет за замовчуванням (default package), який не має жодного імені. (Тому-то ви і не повинні були хвилюватися щодо пакетів до теперішнього часу.) У той час як пакет за замовчуванням хороший для коротких прикладів програм, він неадекватний для реальних додатків. У більшості випадків ви самі будете визначати пакет для свого коду.
- **Загальна форма ІНСТРУКЦІЇ package**

package pkg;

- Тут pkg - ім'я пакета. Наприклад, наступна інструкція створює пакет з ім'ям MyPackage.

package MyPackage;

- Щоб зберігати пакети, Java використовує каталоги файлової системи. Наприклад, class-файли для будь-яких класів, які ви оголошуєте як частина пакета MyPackage, повинні бути збережені в каталозі з іменем MyPackage.
- Пам'ятайте, що регістр суттєвий, і ім'я каталога має точно відповідати імені пакета. Одну і ту ж package-інструкції можуть включати кілька файлів. Вона просто вказує, якому пакету належать класи, визначені у файлі. Це не виключає приналежності інших класів в інших файлах до того ж самому пакету. Більшість реальних пакетів містять багато файлів.

# Визначення пакета

- Можна створювати ієрархію пакетів. Для цього необхідно просто відокремити кожне ім'я пакета від стоїть вище за допомогою операції "точка". Загальна форма інструкції багаторівневого пакета: *package pkg1 [. pkg2 [. pkg3]];*
- Ієрархія пакетів повинна бути відображена у файловій системі вашої системи розробки Java-програм. Наприклад, пакет, оголошений як *package java.awt.image;*
- повинен бути збережений в каталозі java / awt / image, java \ awt \ image або java: awt: image файлової системи UNIX, Windows або Macintosh, відповідно.
- Намагайтеся ретельно вибирати імена пакетів. Не можна перейменовувати пакет без перейменування каталогу, в якому зберігаються класи.

# Використання CLASSPATH

- Розміщенням кореня будь ієрархії пакетів у файловій системі комп'ютера управляє спеціальна змінна оточення CLASSPATH.
- При збереженні всіх класи в одному і тому ж неіменовані пакеті, який використовується за умовчанням, дозволяє просто компілювати вихідний код і запускати інтерпретатор Java, вказуючи (як його параметра) ім'я класу на командному рядку.
- Даний механізм працював, тому що заданий за замовчуванням поточний робочий каталог зазвичай вказується у змінній оточення CLASSPATH, яка визначається для виконавчої (run-time) системи Java за замовчуванням.
- Проте все стає не так просто, коли включаються пакети.

# Використання CLASSPATH

- Припустимо, що ви створюєте клас з ім'ям packTest в пакеті з ім'ям test. Так як ваша структура каталогів повинна відповідати вашим пакетам, ви створюєте каталог з ім'ям test і розміщуєте вихідний файл PackTest.java всередині цього каталогу.
- Потім ви призначаєте test поточним каталогом і компілюєте PackTest.java. Це призводить до збереження результату компіляції (файлу PackTest.class) в каталозі test, як це і повинно бути.
- Коли ви спробуєте виконати цей файл за допомогою інтерпретатора Java, то він виведе повідомлення про помилку "can't find class PackTest" (не можливо знайти клас PackTest).
- Це відбувається тому, що клас тепер збережений в пакеті з ім'ям test. Ви більше не можете звернутися до нього просто як до PackTest.

# Використання CLASSPATH

- Ви повинні звернутися до класу, перераховуючи ієрархію його пакетів і розділяючи пакети точками. Цей клас повинен тепер назватися test. PackTest. Проте якщо ви спробуєте використовувати test.PackTest, то будете все ще отримувати повідомлення про помилку "can't find class test / PackTest" (не можливо знайти клас test / PackTest).
- Причина того, що ви все ще приймаєте повідомлення про помилки, прихована у вашій змінної CLASSPATH. Вершину ієрархії класів встановлює CLASSPATH. Проблема в тому, що, хоча ви самі перебуваєте безпосередньо в каталозі test, в CLASSPATH ні він, ні, можливо, вершина ієрархії класів, не встановлені.
- У цей момент у вас є дві можливості: перейти по каталогам вгору на один рівень і випробувати команду Java test.PackTest або додати вершину ієрархії класів в змінну оточення CLASSPATH. Тоді ви будете здатні використовувати команду Java test.PackTest з будь-якого каталогу, і Java знайде правильний class-файл.
- Наприклад, якщо ви працюєте над вашим вихідним кодом в каталозі C: \ myjava, то встановіть в CLASSPATH наступні шляхи:

```
C:\myjava;C:\java\classes
```

# Захист і керування доступом

- Пакети додають ще один вимір до управління доступом. Java забезпечує достатньо рівнів захисту, щоб допустити добре розгалужений контроль над видимістю змінних і методів в межах класів, підкласів та пакетів
- Класи і пакети, з одного боку, забезпечують інкапсуляцію, а з іншого - підтримують простір імен і області видимості змінних і методів.
- Пакети діють як контейнери для класів та інших залежних пакетів. Класи діють як контейнери для даних і коду. Клас - найдрібніший модуль абстракції мови Java.
- Через взаємодії між класами і пакетами, Java адресує чотири категорії видимості для елементів класу:
- підкласи в тому ж пакеті;
- непідкласи в тому ж пакеті;
- підкласи в різних пакетах;
- класи, які не знаходяться в тому ж пакеті і не є підкласами.

# Захист і керування доступом

Доступ до членів класів

| | Private | Без модифікатора | Protected | Public |
|---|---|---|---|---|
| Той же клас | Yes | Yes | Yes | Yes |
| Підклас того ж пакета | No | Yes | Yes | Yes |
| Непідклас того ж пакета | No | Yes | Yes | Yes |
| Другий підклас пакета | No | No | Yes | Yes |
| Другий непідклас пакета | No | No | No | Yes |

# Імпорт пакетів

- В неіменованому пакеті умовчання немає класів ядра Java, всі стандартні класи зберігаються в кількох іменованих пакетах.
- Щоб забезпечити видимість деяких класів або повних пакетів Java, використовується оператор import. Після імпортування на клас можна посилатися прямо, використовуючи лише його ім'я.
- У вихідному файлі Java оператор import слід негайно після оператора і package (якщо він використовується) і перед будь-якими визначеннями класу. Загальна форма оператора import:

*import pkgl [. pkg2]. (classname | *);*

Тут pkgi - ім'я пакета верхнього рівня, pkg2 - ім'я підлеглого пакета всередині зовнішнього пакета (імена розділяються крапкою). Нарешті, ви визначаєте або явне ім'я класу, або зірочку (*), яка вказує, що компілятор Java повинен імпортувати повний пакет.

Наступний кодовий фрагмент показує використання обох форм:

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

# Інтерфейси

- Творці мови Java надійшли радикально - заборонили множинне спадкування взагалі. При розширенні класу після слова extends можна написати тільки одне ім'я суперкласу. За допомогою уточнення super можна звернутися тільки до членів безпосереднього суперкласу.
- Але що робити, якщо все-таки при породженні треба використовувати кілька предків?
- У таких випадках використовується ще одна конструкція мови Java-інтерфейс.
- Інтерфейс (interface), на відміну від класу, містить тільки константи і заголовки методів, без їх реалізації.
- Інтерфейси розміщуються в тих же пакетах і подпакетах, що і класи, і компілюються теж в class-файли.

# Інтерфейси. Визначення інтерфейсу

- За допомогою ключового слова interface Java дозволяє повністю відокремити інтерфейс від його реалізації.
- Використовуючи інтерфейс, можна визначити набір методів, які можуть бути реалізовані одним або декількома класами.
- Сам інтерфейс в дійсності не визначає ніякої реалізації.
- Хоча інтерфейси подібні абстрактним класах, вони мають додаткову можливість: клас може реалізовувати більше одного інтерфейсу.
- На противагу цьому клас може успадковувати тільки один суперклас (абстрактний або інший).

# Інтерфейси

- Інтерфейси розроблені для підтримки динамічного виклику методів під час виконання.
- Зазвичай для виклику методу одного класу з іншого потрібно, щоб обидва класи були присутні під час компіляції і компілятор Java міг перевірити сумісність сигнатур методів.
- В ієрархічній ж багаторівневою системою, де функціональні можливості забезпечуються довгими ланцюжками пов'язаних в ієрархію класів, цей механізм неминуче використовується все більшим і більшим числом підкласів.
- Інтерфейси виключають визначення методу чи набору методів з ієрархії успадкування.
- Так як інтерфейси знаходяться поза ієрархії класів, то для незв'язаних в цій ієрархії класів з'являється інша, більш ефективна можливість реалізації інтерфейсів.

# Визначення інтерфейсу

- Загальна форма інтерфейсу виглядає так:

**access interface name {**
**return-type *method-name1(parameter-list) ;***
**return-type *method-name2(parameter-list)* ;**
**type *final-varname1* = value;**
**type *final-varname2* = value;**
**return-type *method-nameN (parameter-list) ;***
**type *final-vaxnameN* = value;**

access - специфікатор доступу (або public або не використовується). Якщо ніякого специфікатору доступу не включено, тоді використовується доступ за замовчуванням, і інтерфейс доступний тільки іншим членам пакета, в якому він оголошений.

При оголошенні з public інтерфейс може використовуватися будь-яким іншим кодом, іменем інтерфейсу може бути будь-допустимий ідентифікатор.

Оголошені методи не мають тіл. Вони закінчуються крапкою з комою після списку параметрів.

Кожен клас, який включає інтерфейс, повинен реалізувати всі його методи.

# Приклад визначення інтерфейсу

```
interface Callback  {
void callback(int param);
}
```

- Тут оголошено простий інтерфейс, що містить один метод з ім'ям callback (), який має єдиний цілий параметр.

# Реалізація інтерфейсів

- Загальна форма класу, який включає implements пропозицію, виглядає приблизно так:

*access class classname [extends superclass]*

*[implements interface [, interface ...]] {// / тіло-класу*

- Тут access - специфікатор доступу (public або не використовується). Якщо клас реалізує більше одного інтерфейсу, вони розділяються комою. Якщо клас реалізує два інтерфейси, які оголошують один і той же метод, то клієнти будь-якого інтерфейсу будуть використовувати один і той же метод.
- Методи, які реалізують інтерфейс, повинні бути оголошені як public.

# Приклад класу, який реалізує інтерфейс

- class Client implements Callback {

```
// Реалізація Callback-інтерфейсу
public void callback(int p) {
System.out.println
("callback викликаний з аргументом “+p);
} }
```

- Зверніть увагу, що callback () оголошений з специфікатором доступу public.

# Реалізації доступу через інтерфейсні посилання

- Можна оголошувати змінні як об'єктні посилання, які використовують інтерфейсний тип, а не тип класу.
- У такій змінної можна зберігати всякий екземпляр будь-якого класу, який реалізує оголошений інтерфейс.
- Коли ви викликаєте метод через посилання такого роду, буде викликатися його правильна версія, заснована на актуальному примірнику інтерфейсу. Виконується метод відшукується динамічно (під час виконання), що дозволяє створювати класи пізніше коду, який викликає їх методи.
- Кодом виклику можна управляти через інтерфейс, нічого не знаючи про об'єкт виклику.

# Приклад виклику методу

class Testlface {

public static void main (String args []) {

Callback c = new Client ();

c callback (42);

Висновок цієї програми:

callback викликаний з аргументом 42

Зверніть увагу, що змінної c, оголошеної з типом інтерфейсу Callback, був призначений екземпляр класу client.

Хоча дозволяється використовувати c для звернення до методу callback (), вона не може звертатися до будь-яких інших членів класу client. Змінна інтерфейсного посилання володіє знаннями тільки методів, оголошених у відповідній інтерфейсній декларації.

# Часткові реалізації

- Якщо клас включає інтерфейс, але повністю не реалізує методи, визначені цим інтерфейсом, то цей клас повинен бути оголошений як abstract (абстрактний).

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + " " + b) ;
}
// ...
```

- Тут клас incomplete не реалізує callback () і повинен бути оголошений як абстрактний.
- Будь клас, який успадковує incomplete, повинен реалізувати callback () або оголосити себе як abstract.

# Застосування інтерфейсів

- Стек може мати фіксований розмір, або бути "зростаючим".
- Стек може також міститися в масиві, зв'язковому списку, двійковому дереві і т. д.
- Незалежно від того, як стек реалізований, інтерфейс стека залишається тим же самим.
- Методи push () і pop () визначають інтерфейс до стека незалежно від подробиць реалізації.
- Нижче показаний інтерфейс, що визначає цілий стек.

```
/ / Визначення інтерфейсу цілого стека,
interface IntStack {
void push (int item); / / запам'ятати елемент
int pop (); / / витягти елемент
}
```

# Змінні в інтерфейсах

- Можна використовувати інтерфейси для імпорту поділюваних констант у множинні класи просто оголошенням інтерфейсу, який містить змінні, початкові бажаними значеннями. Коли ви включаєте цей інтерфейс в клас (тобто "реалізуєте" інтерфейс), всі імена зазначених змінних опиняться в області їх видимості як константи.
- Якщо інтерфейс не містить методів, то будь-який клас, який включає такий інтерфейс, фактично не реалізує нічого.

```
import java.util.Random;
interface SharedConstants {
int N0=0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5; }

class Question implements SharedConstants {
     Random rand = new Random();
int ask() {
int prob = (int') (100 * rand.nextDoubleO ) ;
 if (prob < 30)
return NO;  // 60%
else if (prob < 60)
return YES;         // 30% else if (prob < 75)
return LATER;   // 15% else if (prob < 98)
return SOON;        // 13% else
return NEVER;       // 2% } }
```

```
class AskMe implements SharedConstants { static void
      answer(int result) {
switch(result) {
case NO:
System.out.println("Нет"); break;
case YES:
System.out.println("Да"); break;
case MAYBE:
System.out.println("Возможно"); break;
case LATER:
System.out.println("Позже"); break;
case SOON:
System.out.println("Вскоре"); break;
case NEVER:
System.out.println("Никогда"); break; } }

public static void main(String args[]) {
Question q = new Question();
answer (q. ask () ) ;
answer(q.ask());
answer(q.ask());
answer(q.ask()); } }
```

# Розширення інтерфейсів

- Один інтерфейс може успадковувати інший за допомогою ключового слова extends.
- Синтаксис - той же самий, що для наслідування класів.
- Коли клас реалізує інтерфейс, який успадковує інший інтерфейс, перший повинен забезпечити реалізацію для всіх методів, визначених у ланцюжку успадкування інтерфейсу.

# Розширення інтерфейсів (приклад)

```
/ / Один інтерфейс розширює інший,
interface A {
void meth1();
void meth2 (); }
// B тепер включає meth1() и meth2() , а сам він додає meth3().
interface B extends A {
void metn3(); }
// Цей клас повинен реалізувати все з A і B.
class MyClass implements B {
public void meth1() {
System.out.println("Реалізує meth1().");
}
public void meth2() {
System.out.println("Реалізує meth2()."); }
public void meth3() {
System.out.println("Реалізує meth3()."); } }

class IFExtend {
public static void main (String arg[].) { MyClass ob = new MyClass ();
ob.meth1();
ob.meth2();
ob.meth3(); } }
```