Тема 3. Класи в Java

Лекція 5

Опис класу

Загальна форма оголошення класу виглядає наступним чином:

```
модифікатори class им'я_класу
     extends ім'я_базового_класу                      // не обов'язково
     implements ім'я_інтерфейсу1, ім'я_інтерфейсу2    // не обов'язково
{
     // Оголошення змінних (полів)
     модифікатори  тип  ім'я_змінна1;
     модифікатори  тип  ім'я_змінна2 = ініц_знач;

     // Оголошення методів
     модифікатори тип ім'я_методу1(список_параметрів) { тіло_методу; }

}
```

Опис класу починається з ключового слова `class`, після якого записується ім'я класу. Загальноприйняті угоди (Code Conventions) рекомендують починати ім'я класу з великої літери, на відміну від імен методів і змінних, які починаються з маленької літери.

Перед словом `class` можуть знаходитись *модифікатори* класу (class modifiers) – ключові слова `public`, `abstract`, `final`, `strictfp` – які змінюють властивості класу. Ці модифікатори обговорюються далі в цій лекції.

Тіло класу виділяється фігурними дужками і містить всі змінні (поля), методи, вкладені класи і інтерфейси.

Оголошення *змінної* (поля) в класі відбувається так само, як і всередині методу: вказується її тип, ім'я та, можливо, початкове значення. Оголошення змінної може починатися з одного або декількох необов'язкових модифікаторів: `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`. З модифікаторами ми будемо знайомитися в міру необхідності.

При описі *методу* вказується тип значення, що повертається, або слово `void`, потім, через пробіл, ім'я методу, потім, в дужках, список параметрів. Після цього в фігурних дужках пишуться оператори, які повинні виконуватися, коли викликається метод.

Опис методу може починатися з модифікаторів `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`. Їх ми також будемо розглядати по необхідності. У списку параметрів через кому перераховуються тип і ім'я кожного параметра. Список параметрів може бути відсутнім, але дужки зберігаються.

Ключове слово **extends** служить для *успадкування* існуючого класу. У Java заборонено *множинне успадкування*. Будь-який клас неявно успадковує клас **Object.**

Модифікатори класу

При оголошенні класу можливо використання наступних модифікаторів:

- **public** – відкритий клас, який доступний ззовні пакета, в якому він визначений. Такий клас повинен бути оголошений у файлі .java, ім'я якого збігається з ім'ям класу. За замовчанням (якщо модифікатор `public` не вказано) клас доступний тільки зсередини свого пакета. *Пакетом* в Java називається іменована група файлів. Використання пакетів ми розглянемо в наступній лекції.
- **final** – фінальний клас, успадкування якого заборонено. Наприклад, фінальними є багато класів стандартної бібліотеки (такі як System, String).
- **abstract** – абстрактний клас, в якому хоча б один метод не реалізований.

При оголошенні вкладеного класу можна також використовувати модифікатори `protected`, `private`, `static`.

Модифікатори способу доступу

Java підтримує 4 різних рівня доступу, але має всього 3 модифікатори способу доступу. Ці модифікатори вказуються для кожної змінної (поля) в класі і методу окремо і дозволяють задавати 4 варіанти прав доступу:

- **public** – означає, що даний елемент доступний без будь-яких обмежень;
- **private** – доступ дозволений тільки з даного класу;
- **protected –** доступ дозволений з даного класу і з усіх класів-нащадків, а також з усіх класів даного пакету.
- *без модифікатора* – якщо жоден з модифікаторів public, private, protected не вказано, то доступ дозволений з усіх класів даного пакету.

Методи

Як і C++, Java підтримує ***перевантажені методи***. Вимоги до перевантаження методів ті ж.

Всі методи, крім статичних і конструкторів, є ***віртуальними***. Тому вони можуть бути перевизначені в похідних класах.

За аналогією зі змінними нестатичні методи називаються також *методами реалізації*, а статичні – *методами класу*.

У всіх методах реалізації можна використовувати ключове слово **this** – воно позначає посилання на поточний об'єкт (для якого викликаний даний метод). Зазвичай посилання this використовується в двох випадках:

1) якщо метод повинен повернути посилання на об'єкт, для якого він був викликаний (`return this;`), наприклад:

```
class SomeClass {

     int state = 0;                        // Стан об’єкта
     // ...
     SomeClass change( int newState ) {  // Метод, що змінює стан об’єкта
          state  =  newState;              // Змінити поточний об’єкт
          return this;                     // Повернути посилання на нього
     }
}

public static void main(String[] args) {
     SomeClass obj = new SomeClass();     // Використовуємо клас SomeClass
     obj.change(1).change(2);             // Перший виклик change() повертає
                                          // посилання на об’єкт, який можна
                                          // далі використовувати
}
```

2) для звертання до полів об’єкта, якщо їх імена співпадають з іменами формальних параметрів методу, наприклад:

```
class SomeClass {

     int state = 0;                        // Стан об’єкта
     // ...
     void change( int state ) {           // Ім’я параметра співпадає з іменем поля
          this.state  =  state;           // Зліва – поле; зправа - параметр
     }
}
```

Ключове слово **super** можна використовувати в методах реалізації довільного класу (не статичних) для посилання на змінні (поля) і методи базового класу. Наприклад:

```
class My {                    // extends Object
     public String toString() { return "My : " super.toString(); }
     public out() { System.out.println( super.toString); }
}
```

Конструктори класу

Конструктор класу (class constructor) – це спеціальний метод, ім'я якого співпадає з іменем класу, і який відповідає за створення об'єктів класу.

Особливості конструктора:

- Конструктор є в будь-якому класі. *Конструктор за замовчанням* (default constructor) створюється автоматично, але такий конструктор не робить нічого, крім виклику конструктора базового класу. Нагадаємо, що за замовчанням всі класи успадковуються від Object.
- Конструктор виконується автоматично при створенні екземпляра класу.
- Конструктор не повертає ніякого значення. Тому в його описі не пишеться навіть слово `void`. У той же час конструктор може мати модифікатор способу доступу (`public`, `protected` або `private`).
- Конструктор не є віртуальним методом, тому його не можна успадковувати або перевизначити в похідному класі.
- Тіло конструктора, крім звичайних операторів, може починатися:
  - з виклику одного з конструкторів базового класу, для цього записується слово `super()` з параметрами в дужках, якщо вони потрібні;
  - з виклику іншого конструктора того ж класу, для цього записується слово `this()` з параметрами в дужках, якщо вони потрібні.
- У класі може бути кілька перевантажених конструкторів, які відрізняються типом та/або кількістю параметрів.

У всьому іншому конструктор подібний до звичайного методу: в ньому дозволяється записувати будь-які оператори, навіть оператор `return`, але тільки порожній, без всякого значення, що повертається.

При ***створенні об'єкта*** (наприклад, `new MyClass(prm)`) відбуваються наступні дії:

- виділяється пам'ять під об'єкт і створюється посилання на нього;
- викликається конструктор базового класу (якщо явного виклику `super()` немає, то автоматично викликається конструктор базового класу без параметрів);
- поля (змінні реалізації) об'єкта похідного класу ініціалізуються значеннями, зазначеними при оголошенні (за замовчанням – нулями);
- виконується тіло конструктора похідного класу (якщо воно є);
- створене посилання на об'єкт використовується у виразі, наприклад присвоюється змінній.

Таким чином, якщо в класі не оголошено жодного конструктора, і не вказано базовий клас (`extends`), то при створенні екземплярів викликався конструктор без параметрів класу Object.

Приклад. Клас дробів, що має кілька конструкторів. Змінні реалізації цього класу зберігають чисельник і знаменник дробу, а конструктори ініціалізують ці змінні.

```
public class Ration
     // extends Object                 // Наслідується за замовчанням
{
     protected int num = 0, den = 1;  // Змінні реалізації
     Ration() {}                      // Конструктор за замовчанням нічого не робить
     Ration(int num, int den) {       // Конструктор з параметрами
          // super();                 // Тут можна викликати конструктор базового
класу
          this.num = num;
          this.den = den==0? 1 : den;
     }
     public Ration( Ration r ) {      // Копіюючий конструктор
          this( r.num, r.den );       // просто викликає конструктор з параметрами
     }
```

```
    public String toString() {       // Перетворення дробу на рядок
        return "" + r.num + "/" + r.den;
    }

    // Створення об'єктів-дробів
    public static void main(String[] args) {
        Ration r1 = new Ration();       // Викликано конструктор без параметрів
        Ration r2 = new Ration(1,2);    // Викликано конструктор з параметрами
        Ration r3 = new Ration(r2);     // Створено копію r2 з допомогою
                                        // копіюючого конструктора
        System.out.println(r3);         // Виводимо "1/2"
    }
}
```

У цьому прикладі реалізований один з методів класу `Object` – `toString()`. Він не має параметрів і викликається автоматично всякий раз, коли необхідно перетворити об'єкт нашого класу на рядок. Наприклад в виразі `System.out.println(r3);` спочатку об'єкт `r3` приводиться до типу `String`, а потім викликається метод `ptrintln()`.

Метод finalize()

Метод **finalize()** визначений в класі `Object` і успадковується всіма іншими класами. Цей метод для кожного об'єкта викликається автоматично перед утилізацією цього об'єкта збирачем сміття. Зазвичай в методі `finalize()` звільняються ресурси, зайняті об'єктом (файли, сокети і т. п.). Як типовий приклад, розглянемо клас, який в конструкторі відкриває файл, і в методі `finalize()` звільняє цей файл.

```
    class My {
        FileInputStream aFile = null;         // Потік читання із файла

        My(String filename) {                 // Конструктор
            try {
                aFile = new FileInputStream(filename);  // Відкриття файла
            } catch (java.io.FileNotFoundException e) {
                System.err.println("Could not open file " + filename);
            }
        }

        protected void finalize() throws Throwable {
            // Обов'язково викликаємо finalize() для базового класу, інакше об'єкт
            // може некоректно видалитись
            super.finalize();
            // Звільняємо всі ресурси, виділені в похідному класі
            if (aFile != null) aFile.close();
        }
    }
```

Модифікатор static

Змінні і методи, які оголошені без модифікатора **static**, називають ***змінними (методами) реалізації (instance variables)***.

Змінні і методи, оголошені як **static**, називають ***змінними (методами) класу (class variables).*** Статичний метод може звертатися тільки до статичних полів і методів.

Статичними оголошуються змінні і методи, які не залежать від конкретного об'єкта. Тому до статичних змінних і методів можна звертатися, не створюючи жодного екземпляра класу. Наприклад, ми можемо користуватися методами класу `Math`, не створюючи його екземпляри, а просто записуючи `Math.abs(x)`, `Math.sqrt(x)`. З цієї ж причини метод `main()` завжди оголошується статичним. Також ми багато разів використовували статичну змінну `out` в класі `System`, наприклад: `System.out.println()`.

Приклад. Для створеного раніше класу дробів `Ration` реалізуємо метод складання дробів двома способами: методом класу і методом реалізації.

```
public Ration add(Ration b) {          // Метод реалізації
     den = den * b.den;                // Змінюємо поточний об'єкт (this)
     num = den*b.num + num*b.den;
     simpl();                          // Спрощуємо дріб
     return this;                      // Повертаємо посилання на об'єкт
}

public static Ration add(Ration a, Ration b) {          // Метод класу
     // Створюємо новий об'єкт як копію a, змінюємо і повертаємо його
     return new Ration(a).add(b);
}

// ...

public static void main(String[] args) {
     Ration r1 = new Ration(1,2);
     Ration r2 = new Ration(3,2);
     Ration r3 = new Ration(1/6);
     // Визиваємо метод реалізації (від об'єкта r3)
     System.out.println( r3.add(r1) );  // Змінюється r3 і виводиться "2/3"
     // Викликаємо метод класу (від самого класу)
     System.out.println( Ration.add(r1, r2) );    // r1 не змінюється,
                                                  // і виводиться "2/1"
}
```

Для того, щоб статичний метод не залежав від об'єкта класу, він повинен відповідати кільком вимогам:

- в статичному методі не можна використовувати посилання `this` і `super`;
- в статичному методі не можна використовувати нестатичні поля і методи;
- статичні методи не можуть бути абстрактними;
- статичні методи можуть бути перевизначені в похідних класах тільки як статичні.

Модифікатор final

Цей модифікатор, в залежності від того, де він використаний, може мати різний зміст. Коротко розглянемо всі можливі випадки.

1. Для змінних реалізації `final` означає, що ініціалізація змінної повинна проводитися або при оголошенні, або в конструкторі. Після ініціалізації значення фінальної змінної не може бути змінено.

2. Для змінних класу (статичних) `final` має схоже значення, але така змінна повинна ініціалізуватись тільки при оголошенні. На практиці статичні фінальні змінні використовуються для оголошення констант. Наприклад, в класі дробів може бути оголошена константа `PI` – раціональне число, близьке до реального значення π :

```
public static final Ration PI = new Ration(31415, 10000);
```

3. Для параметра методу модифікатор `final` означає, що даний параметр не можна змінювати всередині методу.

4. Для методу `final` означає, що цей метод не може бути перевизначений в похідних класах. Якщо `final` поставити перед визначенням класу, то всі методи цього класу стають фінальними.

Навіщо ж буває необхідно позначати метод або клас модифікатором `final` і тим самим забороняти їх перевизначення в підкласах? Зазвичай це робиться з метою безпеки: ви можете бути впевнені, що метод не буде перевизначений, і завжди буде виконувати саме ті дії, які ви задали. Зрозуміло, такий метод не може бути абстрактним. Наприклад, фінальними визначені класи `Math`, `String`, `Integer`. В результаті, наприклад, ми можемо бути впевнені, що метод `Math.cos(x)` обчислює саме косинус числа x, і цю поведінку методу неможливо змінити за допомогою

успадкування та поліморфізму. Для повної безпеки, поля, оброблювані фінальними методами, слід зробити закритими (`private`).

Примітка: Якщо перед оголошенням деякої змінної або методу потрібно поставити кілька модифікаторів, то специфікація мови (JLS) рекомендує перераховувати їх в наступному порядку:

```
public, protected, private, static, final, transient, volatile.
```

Успадкування і поліморфізм

В Java успадкування і поліморфізм використовуються так само, як і в інших об'єктно-орієнтованих мовах (C++, C#), тому розглянемо їх дуже коротко.

Успадкування – це відношення між класами, при якому один клас (*нащадок або підклас*) розширює функціональність іншого (*батька, суперкласу або базового класу*). Це означає, що нащадок автоматично переймає всі поля і методи батька, а також додає деякі свої.

Успадкування зазвичай виникає, коли всі об'єкти одного класу одночасно є об'єктами іншого класу (відношення *загальне-часткове*). Наприклад, всі об'єкти класу Студент є об'єктами класу Людина. У цьому випадку говорять, що клас Студент успадковується від класу Людина.

Зауважимо, що якщо клас A є нащадком класу B, а клас B є нащадком класу C, то клас A є також нащадком класу C. Наприклад, в Java всі класи безпосередньо або опосередковано успадковані від класу `Object`.

Успадкування спрощує роботу програміста. Наприклад, якщо в програмі необхідно ввести новий клас «Привілегійований користувач», то його можна створити на основі вже існуючого класу «Користувач», не програмуючи заново всі поля і методи, а лише додавши ті, яких не вистачало в базовому класі.

Для того, щоб один клас був нащадком іншого, необхідно при його оголошенні після імені класу вказати ключове слово `extends` і назву суперкласу.

Щоб не виникало неоднозначностей з успадкуванням методів, в Java заборонено *множинне успадкування.*

Приклад. На базі класу дробів Ration створимо новий клас, в якому змінимо спосіб перетворення дробу на рядок.

```
public class RationEx extends Ration
{
     // Змінні num і den наслідуються із класу Ration

     // Конструктори не наслідуються – їх треба описувати заново,
     // але можна викликати конструктор базового класу
     RationEx(int num, int den) {         // Конструктор з параметрами
          super(num, den);                // Виклик конструктора базового класу
     }
     public RationEx( Ration r ) {        // Копіюючий конструктор
          super( r );                     // Виклик конструктора базового класу
     }

     // Перевизначений метод
     @Override
     public String toString() {           // Нове перетворення дробу в рядок
          return (double)r.num / r.den;
     }
}
```

У цьому прикладі перевизначенний метод позначений анотацією `@Override`. Ця анотація не обов'язкова, але дуже корисна. Якщо її поставити, компілятор буде перевіряти правильність перевизначення (тобто чи збігається сигнатура нового методу з методом в базовому класі), і це дозволить уникнути важко вловимих помилок.

Використання поліморфізму також розглянемо на простому прикладі. Збережемо різні об'єкти класів `Ration` і `RationEx` в один масив і виведемо весь вміст цього масиву. Виводитися наші об'єкти будуть по-різному, але в коді програми про це піклуватися не потрібно.

```
public static void main(String[] args) {
     // Створюємо масив об’єктів різних класів
     // Це можна зробити тільки якщо всі вони наслідуються від класу,
     // який використано в оголошені масиву!
     Ration rr[] = { new Ration(1,2), new RationEx(1,2),
                     new Ration(3,2), new RationEx(3,2) };
     // Виводимо всі об’єкти незалежно від типу
     for (Ration r : rr) {
          System.out.println( r );
     }
}
```

Масив типу `Ration` може містити як об'єкти класу `Ration`, так і будь-якого похідного класу. У циклі ми отримуємо посилання на черговий елемент масиву типу `Ration`, але, якщо за цим посиланням знаходиться об'єкт класу `RationEx`, то для перетворення на рядок буде викликаний перевизначений метод з класу `RationEx`, в чому можна переконатися, виконавши цей приклад.

Клас Object

Як уже неодноразово говорилося, клас `Object` є коренем ієрархії класів Java. Він містить загальні методи, які застосовуються для всіх класів. Ці методи дуже важливі, тому що вони визначають основи поведінки об'єкта. Нові класи повинні перевизначити методи `Object`, якщо їх не влаштовує стандартна реалізація. Як це зробити, ми розглянемо на прикладі нашого класу дробів.

`public boolean equals(Object obj);`

Цей метод порівнює поточний об'єкт з об'єктом `obj`. Він реалізований в класі `Object` як тотожність: об'єкт дорівнює тільки самому собі. Будь-яка реалізація цього методу повинна встановлювати певне *відношення еквівалентності* об'єктів.

Наприклад, для класу `Ration` метод `equals()` можна реалізувати наступним чином:

```
@Override
public boolean equals(Object obj) {
     if (this == obj)           // Об’єкт порівнюється сам з собою
          return true;
     if (obj == null)           // Об’єкт порівнюється з с null
          return false;
     if (getClass() != obj.getClass())      // obj – класу Ration?
          return false;
     Ration other = (Ration) obj;           // Приводим тип до Ration
     if (den != other.den)                  // Тепер можна порівняти поля
          return false;
     if (num != other.num)
          return false;
     return true;
}
```

Зверніть увагу, що `obj` має тип `Object`, тому перш ніж звертатися до його полів, потрібно привести його до типу `Ration`. Однак в якості `obj` може бути переданий об'єкт абсолютно іншого класу, або `null` – обидва ці випадки потрібно заздалегідь перевірити, інакше можливі помилки.

`protected native Object clone() throws CloneNotSupportedException;`

Цей метод створює копію об'єкта. Реалізація за замовчанням *захищена* (`protected`) і ззовні класу не доступна. Так зроблено тому, що в класі `Object` нічого копіювати, і, відповідно метод `clone()` просто не потрібен.

Якщо у вашому класі метод `clone()` не перевизначений зі способом доступу `public`, то ваші об'єкти також не можна буде копіювати (при спробі виклику `clone()` буде генеруватися виключення).

Існує угода, що `clone()` повертає новий об'єкт, такий, що виконуються дві умови:

```
x.clone().getClass() == x.getClass()
x.clone().equals(x) == true
```

тобто «клон» повинен бути того ж класу, що й вихідний об'єкт і повинен бути «рівним» йому в розумінні методу `equals()`.

Друга умова означає, що при перевизначенні `clone()` також потрібно перевизначити і метод `equals()`, і так, щоб ці два метода один одному відповідали.

Приклад. Реалізація `clone()` в класі `Ration`.

```
@Override
public Object clone() {
     return new Ration(this);
}
```

```
public native int hashCode();
```

Цей метод використовується класами-колекціями (`HashMap`, `HashSet`) для отримання ***хеш-коду*** об'єкта, тобто цілого числа, яке можна використовувати для швидкого пошуку цього об'єкта. Необхідність перевизначати `hashCode()` виникає, якщо планується зберігати об'єкти в ***хеш-таблиці***.

Метод `hashCode()` завжди повинен бути узгоджений з методом `equals()` – інакше можлива втрата даних з хеш-таблиці:

- рівні об'єкти повинні повертати однакові хеш-коди;
- однак для різних об'єктів коди можуть збігатися (така ситуація називається колізією).

Сама реалізація `hashCode()` повинна відповідати таким основним вимогам – інакше зберігання об'єктів в хеш-таблиці буде неефективним:

- кожен біт даних об'єкта повинен істотно впливати на хеш-код;
- хеш-коди всіх можливих об'єктів повинні бути розподілені по всьому діапазону типу `int` рівномірно;
- кількість колізій повинна бути мінімальна.

Перевизначити `hashCode()` можна або вручну, або скориставшись засобами генерації вихідного коду в InelliJ IDEA: *Alt-Insert → Generate hashCode() and equals().*

Наприклад, для класу Ration буде згенеровано наступний код:

```
@Override
public int hashCode() {
     final int prime = 31;
     int result = 1;
     result = prime * result + den;
     result = prime * result + num;
     return result;
}
```

Тут результат формується шляхом «перемішування» бітів з двох полів об'єкта-дробу.

Якщо клас містить дійсну змінну реалізації `double d;` то в `hashCode()` будуть використані всі біти цієї змінної:

```
     temp = Double.doubleToLongBits(d);
```

```
result = prime * result + (int) (temp ^ (temp >> 32));
```

Якщо ваш клас має поля-масиви, наприклад `double[] doubles`, то можна використовувати готову реалізацію `hashCode()` в класі `Arrays`:

```
result = prime * result + Arrays.hashCode(doubles);
```

`public String toString();`

Цей метод `Object` вже розглядався. Він перетворює об'єкт на рядок. Реалізація за замовчанням формує рядок, що містить позначення типу та хеш-код об'єкта. Наприклад: `MyClass@e53108`, `[I@e53108`. У багатьох класах цей метод потрібно перевизначити.

Приклад перевизначення `toString()` в класі `Ration`:

```
@Override
public String toString() {
     return super.toString() +
          " : Ration [num=" + num + ", den=" + den + "]";
}
```

`protected void finalize() throws Throwable;`

Перевизначення цього методу було розглянуто раніше. Метод повинен звільняти ресурси, зайняті об'єктом. В нашому прикладі клас `Ration` не використовує ніяких зовнішніх ресурсів, тому він не потребує реалізації методу `finalize()`.

Рефлексія і клас Class

Змінна-посилання може містити посилання на об'єкти різних класів, пов'язаних успадкуванням. ***Рефлексією*** називається здатність об'єкта описувати самого себе, і зокрема – повідомляти свій клас. Ця здатність внесена в самий базовий клас `Object`, який має:

- Поле **`class`**
- Метод **`public final native Class getClass()`**

Метод `getClass()` повертає об'єкт типу `Class` – клас даного об'єкта.

Також для об'єктів класу **`Class`** визначені

- Оператор **`insanceof`**;
- Метод `boolean isInsance (Object obj)`,

які використовуються для перевірки приналежності об'єкта класу.

Приклад використання поліморфізма і рефлексії

```
// Збережемо посилання на об’єкти різних класів в одному масиві
Object a[] = {new String("qqqq"),  new int[] {1,2,3}, new Object() };

// Обробимо цей масив, використовуючи поліморфізм
for( Object i : a) {
     // Вивести назву класу для кожного об’єкта
     System.out.print( i.getClass().toString() + "   ");

     // три способи перевірити клас об’єкта
     // if( i.getClass() == String.class ) {...}      // 1) i – рядок?
     // if ( String.class.isInstance(i) )  {...}      // 2) те ж

     if (i instanceof String ) {      // 3) i – типу String або спадкоємець String?

          // Щоб працювати  як із рядком потрібне приведення типу
          System.out.println( ((String)i).charAt(0));
```

```
    } else if (i instanceof int[]) {     // Якщо i - масив

        // Приведення типу до типу-масиву та обробка масиву
        // Не можна привести до типу Object[] !!!
        System.out.println(Arrays.toString( (int[]) i ));
    }
}
```

Примітка: щоб перевірити, чи є посилання масивом (будь-якого типу), можна використовувати конструкцію виду:

```
if( i.getClass().isArray() ) { ... }
```

Абстрактні класи та інтерфейси

Абстрактний клас - це клас в якому частина методів не реалізована. Він оголошується з модифікатором **`abstact`**. Цей же модифікатор повинен використовуватися для нереалізованих методів, наприклад:

```
public abstract class D {       // Абстрактний клас
   . . .
    protected int g1(int s) {   // Звичайний метод
       . . .
    }

    public abstract void g2(String str);          // Абстрактний метод
    . . .
}
```

Абстрактні класи використовуються, якщо деякі методи потрібно оголосити як загальні для всіх класів-нащадків, але надати загальну реалізацію неможливо або недоцільно. Нехай, наприклад, маємо клас «Фігура», і від нього успадковані класи «Прямокутник» і «Трикутник». В даному випадку доречно оголосити операцію знаходження площі в базовому класі «Фігура», але реалізовуватися вона буде в кожному похідному класі по-різному.

Всі абстрактні класи мають такі особливості:

- *Не можна створити об'єкт абстрактного класу* (наприклад «Фігура»). Необхідно, використовуючи цей клас як базовий, створити інший клас (наприклад «Трикутник»), в якому визначити всі абстрактні методи. Тоді можна буде створювати об'єкти похідного класу «Трикутник».
- З іншого боку, *не заборонено описувати змінні абстрактного класу*. Наприклад, користуючись поліморфізмом, можемо зберігати посилання типу «Трикутник» або «Прямокутник» в змінній типу «Фігура».

Інтерфейс – це повністю абстрактний клас, який не містить ніяких полів, крім констант (`static final` поля). Інтерфейс оголошується подібно класу, але з ключовим словом **interface.** Клас може реалізувати один або кілька інтерфейсів – для цього інтерфейси вказуються після ключового слова **implements**. Розглянемо приклад:

```
public interface Figure {       // Інтерфейс
   . . .
   double square();             // Абстрактні методи
}

// Клас, що реалізує інтерфейс
public class Triangle implements Figure {
   public double square() {     // Реалізація абстрактного методу
      // ...
   }
}
```

Усередині неабстрактного класу, що реалізує деякий інтерфейс, повинні бути реалізовані всі методи, описані в цьому інтерфейсі. Також можна створити абстрактний клас, який реалізує тільки деякі методи інтерфейсу. Як і з абстрактними класами, не можна створювати об'єкти інтерфейсів, але можна описувати змінні типу інтерфейсів, наприклад:

```
Figure f1 = new Triangle(), s1 = new Rectangle();
```

Отже, інтерфейси схожі з абстрактними класами, але, на відміну від них:

- інтерфейс не може містити ніяких полів, крім констант, а також неабстрактних методів;
- інтерфейси допускають множинне успадкування;
- всі методи інтерфейсу за замовчуванням вважаються `public`, але в похідному класі вказувати `public` обов'язково.

Інтерфейси для порівняння об'єктів

У Java є два стандартних інтерфейси **java.lang.Comparable** і **java.util.Comparator**, які суттєво спрощують пошук і сортування об'єктів в масивах і колекціях. Обидва інтерфейси забезпечують можливість порівнювати об'єкти між собою на «більше-менше». Розглянемо детально один з них – **Comparable**.

В інтерфейсі **Comparable** оголошений всього один метод

```
int compareTo(Object obj)
```

Він повинен порівнювати поточний об'єкт (`this`) з об'єктом obj. На відміну від методу `equals()`, який повертає `true` або `false`, `compareTo()` повертає:

• 0, якщо об'єкти рівні;
• Від'ємне значення, якщо поточний об'єкт менше параметра;
• Додатне значення, якщо поточний об'єкт більше параметра.

Стандартні класи, такі як `Integer`, `Long`, `Double`, `String`, `Date` та ін., вже реалізують інтерфейс `Comparable`.

Як параметр в `compareTo()` може передаватися об'єкт будь-якого класу. Це не дуже зручно, оскільки перед порівнянням потрібно перевіряти тип цього об'єкта.

Є також друга форма цього інтерфейсу, в якій явно вказується клас, з яким буде проводитися порівняння: `Comparable<Class>`.

Розглянемо обидва варіанти на прикладі класу дробів.

Приклад. Клас дробів, що реалізує інтерфейс `Comparable`. Тут порівняння дробів зведено до порівняння двох об'єктів класу `Double`.

```
class Ration
     implements Comparable        // Для порівняння з будь-якими об'єктами
{
     int num = 0, den = 1;        // Змінні реалізації
     // ...

     public int compareTo(Object r) {          // Реалізація Comparable
          // Необхідна перевірка типу і приведення типу перед порівнянням!
          if( !(r instanceof Ration) ) return -1;
          Ration rr = (Ration)r;
          return Double.valueOf((double)num/den).compareTo( (double)rr.num/rr.den );
     }
}
```

Приклад. Клас дробів, що реалізує інтерфейс `Comparable<Ration>`. У цьому випадку метод `compareTo()` виглядає простіше, тому що в ньому не потрібна перевірка типу параметру.

```
class Ration
     implements Comparable<Ration>    // Для порівняння тільки з об’єктами Ration
{
     int num = 0, den = 1;            // Змінні реалізації
     // ...

     // Реалізація Comparable<Ration>
     public int compareTo(Ration r) {
          return  Double.valueOf( (double)num/den ).compareTo(
                         (double)r.num/r.den );
     }
}
```