Тема 4. Класи стандартної бібліотеки

Лекція 6

Бібліотеки і пакети

Бібліотека Java – це набір скомпільованих класів. Зазвичай бібліотека – це .jar-файл (rt.jar, наприклад). Але свою особисту бібліотеку можна зробити і просто в будь-якому каталозі. Крім того, бібліотека може бути zip-файлом.

Щоб використовувати будь-який готовий клас, потрібно підключити бібліотеку, в якій цей клас знаходиться. Компілятор Java знаходить шлях до бібліотек класів в змінній оточення CLASSPATH. У IntelliJ IDEA бібліотеки можна додавати в діалозі Project Structure (Ctrl-Alt-Shift-S) на вкладках Libraries і Global Libraries. Саме там вказується використовувана бібліотека JDK, що містить всі стандартні пакети Java.

Пакет – це група класів всередині бібліотеки. Ім'я пакета визначає загальний простір імен для цих класів. Наприклад, ми можемо створити власний клас `String` у власному пакеті `myclasses`, і тоді він не буде конфліктувати зі стандартним класом `String`.

Стандартна бібліотека Java містить наступні ***основні пакети***:

- `java.lang` – основний пакет, який підключається автоматично і містить найбільш загальні класи: `Object`, `System`, `String`, класи виключень, потоків і т.д;
- `java.io` – пакет підтримки введення/виведення, що містить класи для роботи з файлами і потоками введення-виведення;
- `java.util` – містить безліч корисних класів, що полегшують написання програм. У цьому пакеті містяться класи-колекції, класи по роботі з датою-часом, і ін .;
- `java.awt` – базові класи для створення графічного інтерфейсу;
- `javax.swing` – розширений пакет класів візуальних компонент (з'явився в Java 2.0);
- `java.applet` – класи для створення аплетів.

Щоб використовувати клас з пакета, його потрібно підключити за допомогою оператора **import**:

```
import java.util.ArrayList;
. . .
ArrayList objList = new ArrayList();
```

Можна одним рядком підключити всі класи з пакета, наприклад,

```
import java.util.*;
```

Пакети можуть бути вкладені один в другий. Оскільки в проекті кожен пакет являє собою папку з java-файлами, то вкладені пакети представляють собою вкладені папки. Ім'я вкладеного пакету складається за принципом <пакет>. <вкладений_пакет>. <ще_вкладений_пакет>.

Створення пакета

Для того щоб створити власний пакет необхідно виконати наступні дії:

1. Вибрати ім'я пакета. Згідно загальноприйнятому правилу, в імені пакета повинна бути присутня web-адреса (домен) фірми-розробника (або електронна адреса окремого розробника) в зворотному порядку. Наприклад, якщо адреса розробника my@prov.ua, то для пакета його допоміжних класів він використовує ім'я ua.prov.my.util. Загальною угодою є іменування пакетів тільки маленькими буквами.

2. Створити необхідну структуру каталогів: а) створити каталог бібліотеки, наприклад, c:\javaproj; б) в ньому створити підкаталоги, що відповідають ієрархії пакетів, наприклад c:\javaproj\ua\ prov\my\util.

3. Підключити бібліотеку до проекту (або задати шлях до бібліотеки в змінній classpath). Якщо пакет створюється засобами IDE безпосередньо в тому ж проекті, де він буде використовуватися, то цей крок не потрібен.

4. На початку кожного java-файлу пакета вказати оператор **package**, наприклад:

```
package ua.prov.my.util;
class MyClass {public void use() {} }
```

5**.** Підключити і використовувати клас із пакета, наприклад:

```
import ua.prov.my.util.MyClass;
...
MyClass a = new MyClass();   a.use();
```

Основні класи пакета java.lang

Клас Math

Клас `Math` необхідний для виконання математичних операцій. Також він містить дві константи: `E` (константа Ейлера) і `PI` (число π). Всі методи цього класу статичні. Нижче наведені деякі з них.

– abs – повертає абсолютне значення числа;
– sin, cos, tan, asin, acos, atan – обчислюють тригонометричні функції від числа;
– atan2 (x, y) – повертає арктангенс числа x/y;
– ceil – найближче більше ціле число;
– floor – найближче менше ціле число;
– rint – округлює число;
– round – округлює число до значення в типі int або long;
– sqrt, pow, exp, log – обчислюють стандартні математичні функції;
– min, max – вибирають мінімальне/максимальне з двох чисел;
– random – повертає випадкове число від 0 до 1.

Клас System

Клас `System` представляє саму JVM. Конструктор `System()` оголошений `protected`, тому створювати об'єкти класу `System` не можна. Всі методи і властивості цього класу є статичними:

– стандартні потоки введення, виведення і помилок.

```
System.in, System.out, System.err
```

– поточний час і лічильник часу високої точності

```
// Час в мілісекундах з 1.01.1970
long t = System.currentTimeMillis();
// Кількість наносекунд з моменту запуску ОС.
// Використовується тільки для точного заміру продуктивності
long t = System.nanoTime();
```

– метод для отримання змінних стану середовища із ОС `getenv()`. Наприклад, отримання значення змінної PATH виглядає наступним чином

```
Map<String,String> env =  System.getenv();
String val =  System.getenv( "PATH" );
```

– управління параметрами самої JVM за допомогою методів:

```
String val =  getProperty( "key" );
```

```
setProperty( "key", "value" );
```

Тут `key` може приймати значення:

java.class.path – Java class path;
java.library.path – List of paths to search when loading libraries;
os.name – Operating system name;
os.version – Operating system version;
file.separator – File separator ("/" on UNIX);
path.separator – Path separator (":" on UNIX);
line.separator – Line separator ("\n" on UNIX);
user.name – User's account name;
user.home – User's home directory;
user.dir – User's current working directory;

Клас String

`String` – це дуже корисний клас, який надає методи для роботи з рядками. Перелічимо найбільш важливі з його методів:

1. Клас `String` реалізує всі методи `Object`;
2. `int length()` – повертає довжину рядка (кількість символів в ньому);
3. `boolean isEmpty()` – перевіряє чи порожній рядок;
4. `String replace(a, b)` – повертає рядок, де символ a (літерал або змінна типу `char`) замінений на символ b;
5. `String toLowerCase(); StringtoUpperCase()` – повертає рядок, де всі символи початкової строки перетворені до рядкових (прописних);
6. `int indexOf(ch); int lastIndexOf(ch)` – повертає індекс першого (останнього) входження символу ch в рядок, або –1 якщо символ не знайдений.
7. `int indexOf(ch,n)` – те ж, починаючи з індексу n.
8. `char charAt(n)` – повертає код символу, що знаходиться в рядку під індексом n.

Розглянемо тепер більш детально деякі найбільш часто використовувані методи.

`public char charAt (int index)`

Повертає символ із зазначеним зміщенням в рядку. Відлік йде від 0. Для вилучення одного символу використайте `getChars()`. Приклад:

```
String testString = "Котеня";
char myChar = testString.charAt(2);
tv.setText(Character.toString(myChar)); // виводить третій символ - т
```

`public int compareTo(String string)`

Порівнює два рядки на «більше-менше» в алфавітному порядку символів Unicode. Може використовуватися при сортуванні. Якщо рядки співпадають, то результат дорівнює 0; якщо перший рядок менший за другий, то результат менше нуля, і нарешті, якщо перший рядок більший за другий, то результат більше нуля. Регістр символів враховується: символи верхнього регістру (великі літери) вважаються «менше» символів нижнього регістру (малих). Приклад:

```
String testString = "Котеня";
if (testString.compareTo("котеня") == 0) {           // Умова не виконується
    tvInfo.setText("Рядки рівні");
} else {
    tvInfo.setText("Рядки не рівні. Повернуто "
            + testString.compareTo("котеня"));       // повертає -32
}
```

Тепер відсортуємо масив рядків за допомогою бульбашкового методу .

```
String[] text = { "First", "Second", "Third", "Fourth" };
for (int j = 0; j < text.length; j++) {
     for (int i = j + 1; i < text.length; i++) {
          if (text[i].compareTo(text[j]) < 0) {
               String temp = text[j];
               text[j] = text[i];
               text[i] = temp;
          }
     }
     System.out.print(text[j] + " ");
}
```

В результаті отримаємо:

```
First Fourth Second Third
```

`public void getChars(int start, int end, char[] buffer, int index)`

Цей метод призначений для вилучення під-рядка із рядка. Треба вказати індекс початку під-рядка, що вилучається, (start) та індекс символу, наступного за кінцем під-рядка (end). Масив, який приймає виділені символи, знаходиться в параметрі buffer. Індекс в масиві, починаючи з якого буде записуватися під-рядок, передається в параметрі index. Масив повинен бути достатнього розміру, так щоб в ньому помістилися всі символи зазначеного під-рядка.

```
String unusualCat = "Котеня по імені Мяв";
int start = 5;
int end = 12;
char[] buf = new char[end - start];
unusualCat.getChars(start, end, buf, 0);
System.out.println(new String(buf));
```

`public String replace(CharSequence target, CharSequence replacement)`
`public String replaceAll (String regularExpression, String replacement)`
`public String replaceFirst (String regularExpression, String replacement)`

Ці методи міняють символ чи послідовність символів target на replacement, наприклад:

```
String testString = "кит";
System.out.println(testString.replace("и", "і"));     // результат – "кіт"
String s = "001234-cat";
String s = s.replaceFirst ("^0*", "");                // результат – "1234-cat"
```

Розглянемо наступний приклад роботи з рядками і символами. Нехай необхідно згенерувати випадковий рядок із заданого набору символів.

```
// Набір символів
private static final String mCHAR = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
private static final int STR_LENGTH = 9; // довжина генерованого рядка

Random random = new Random();
StringBuffer randStr = new StringBuffer();
for (int i = 0; i < STR_LENGTH; i++) {
    int number = random.nextInt(mCHAR.length()-1);
    char ch = mCHAR.charAt(number);
    randStr.append(ch);
}
System.out.println(randStr.toString());
```

Класи – колекції

Для зберігання великої кількості однотипних даних можуть використовуватися масиви, але вони не завжди є ідеальним рішенням. По-перше, довжина масиву задається заздалегідь. Якщо кількість елементів заздалегідь невідома, доведеться або виділяти пам'ять «із запасом», або робити складні дії з виділення наново пам’яті для масиву. По-друге, елементи масиву мають жорстко задане розміщення в його комірках, тому, наприклад, видалення елемента з масиву не є простою операцією.

У програмуванні широко використовуються такі структури даних як стек, черга, список, множина і т. д., об'єднані загальною назвою колекція. ***Колекція*** – це група елементів з операціями додавання, вилучення та пошуку елемента. Механізм цих операцій істотно розрізняється залежно від типу колекції. Наприклад, елементи *стека* строго впорядковані, додавання нового елемента може відбуватися тільки в кінець послідовності, а отримати можна тільки той елемент, що знаходиться в кінці (тобто той, що був доданий останнім). *Черга*, навпаки, дозволяє отримати лише перший елемент (елементи додаються в один кінець послідовності, а вилучаються з іншого). Інші колекції (наприклад, *список*) дозволяють вилучити елемент з будь-якого місця послідовності, а *множина* взагалі не впорядковує елементи і дозволяє (крім додавання і видалення) тільки дізнатися, чи міститься в ній даний елемент, але не дізнатися номер його позиції.

Java має бібліотеку стандартних колекцій, які зібрані в пакеті `java.util`, тому немає необхідності програмувати їх самостійно.

При роботі з колекціями головне уникати помилки початківців – треба користуватися найбільш універсальною колекцією – наприклад, писати код з розрахунком на обробку будь-якого списку, замість того щоб розраховувати тільки на випадок стека. Але якщо для логіки роботи програми критично, що дані зберігаються саме в стеці (тобто з'являються і обробляються в зворотній послідовності), слід використовувати саме стек. Тоді не можливо буде порушити логіку обробки даних, звернувшись безпосередньо до середини послідовності, а значить, шанс появи важко вловимих помилок, різко зменшується.

Щоб вибрати колекцію, яка найкраще підходить до кожної задачі, необхідно знати особливості кожної з них. Ці знання є обов'язковими для будь-якого програміста, оскільки без застосування тих чи інших колекцій не обходиться жодна сучасна задача.

Отже, всі класи колекцій визначені в пакеті `java.util`. Кожна колекція – це ***шаблон*** (generic type), тому може зберігати елементи будь-якого ***типу-посилання***. Примітивні типи в колекції зберегти неможливо, замість них потрібно використовувати класи-обгортки (наприклад `Double` замість `double`, `Integer` замість `integer` і т. д.).

Колекції об'єктів розбиті на кілька великих категорій, описаних наступними **інтерфейсами** (рис. 4.1):

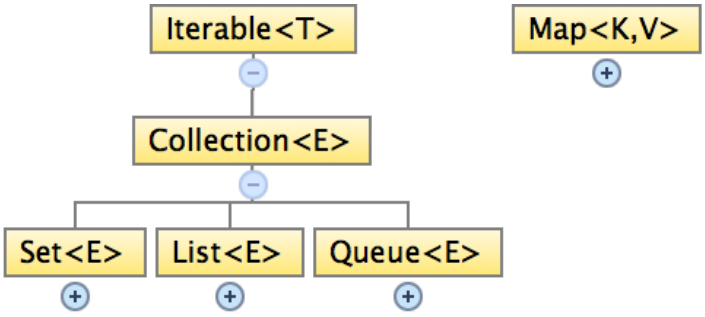

Рис. 4.1 Інтерфейси колекцій JCF

Існує два базових інтерфейси, реалізації яких представляють сукупність всіх класів колекцій: `Collection` і `Map`.

Collection<E> – це колекція, що містить набір об'єктів (елементів). Тут визначено основні методи для маніпуляції елементами, такі як вставка (*add, addAll*), видалення (*remove, removeAll, clear*), пошук (*contains*).

Інтерфейс `Collection<E>` визначає абстрактну колекцію, яка підтримує (але не реалізує) такі основні операції:

- додати елемент в колекцію `add(E e);`
- отримати розмір колекції `size();`
- перевірити, чи є елемент в колекції (рівний в сенсі `e.equals()`) `contains (E e);`
- вибрати елемент за номером `get(int i);`
- замінити елемент за номером `set(int i, E e);`
- видалити з колекції елемент за значенням `remove(Object o);`
- перетворити колекцію на масив `E arr[] = (E[]) coll.toArray();`
- отримати ітератор `iterator();`

Також є можливість перебрати елементи будь-якої колекції за допомогою циклу *for each*.

Цей інтерфейс уточнюється трьома більш конкретними інтерфейсами: List, Set і Queue.

- **List<E>** – описує упорядкований список. Елементи списку пронумеровані, починаючи з нуля, і до будь-якого елементу можна звернутися за його номером у списку (індексом). Інтерфейс `List` є спадкоємцем інтерфейсу `Collection`, тому містить всі його методи і додає до них кілька власних:

`add (int index, Object item)`	вставляє елемент item в позицію index, при цьому список розсовується (всі елементи, починаючи з позиції index, збільшують свій індекс на 1);
`get (int index)`	повертає об'єкт, що знаходиться в позиції index;
`indexOf (Object obj)`	повертає індекс першої появи елемента obj в списку;
`lastIndexOf (Object obj)`	повертає індекс останньої появи елемента obj в списку;
`add (int index, Object item)`	замінює елемент, що знаходиться в позиції index, іншим об'єктом item;
`subList (int from, int to)`	повертає новий список, що є частиною даного (починаючи з позиції from до позиції (to-1) включно).

- **Set<E>** – множина об'єктів (невпорядкований список без повторень).

У множині:

а) повторне додавання об'єкта не змінює множину;

б) порядок елементів при будь-якому способі перебору множини – довільний.

Інтерфейс `Set` успадкований від інтерфейсу `Collection`, але ніяких нових методів не додає. Змінюється тільки поведінка методу `add (Object item)` – він не стане додавати об'єкт item, якщо той вже присутній у множині.

- **Queue<E>** – інтерфейс, який описує чергу. Елементи можуть додаватися в чергу тільки з одного кінця, а виходити з іншого (аналогічно черзі в магазині). Інтерфейс `Queue` так само успадкований від інтерфейсу `Collection`. Специфічні для черги методи:

`poll()`	повертає перший елемент і видаляє його з черги;
`peek()`	повертає перший елемент черги, не видаляючи його;
`offer(Object obj)`	додає в кінець черги новий елемент і повертає true, якщо вставка вдалася.

На рис. 4.2 показані інтерфейси, які стосуються `Collection`, та їхні методи.

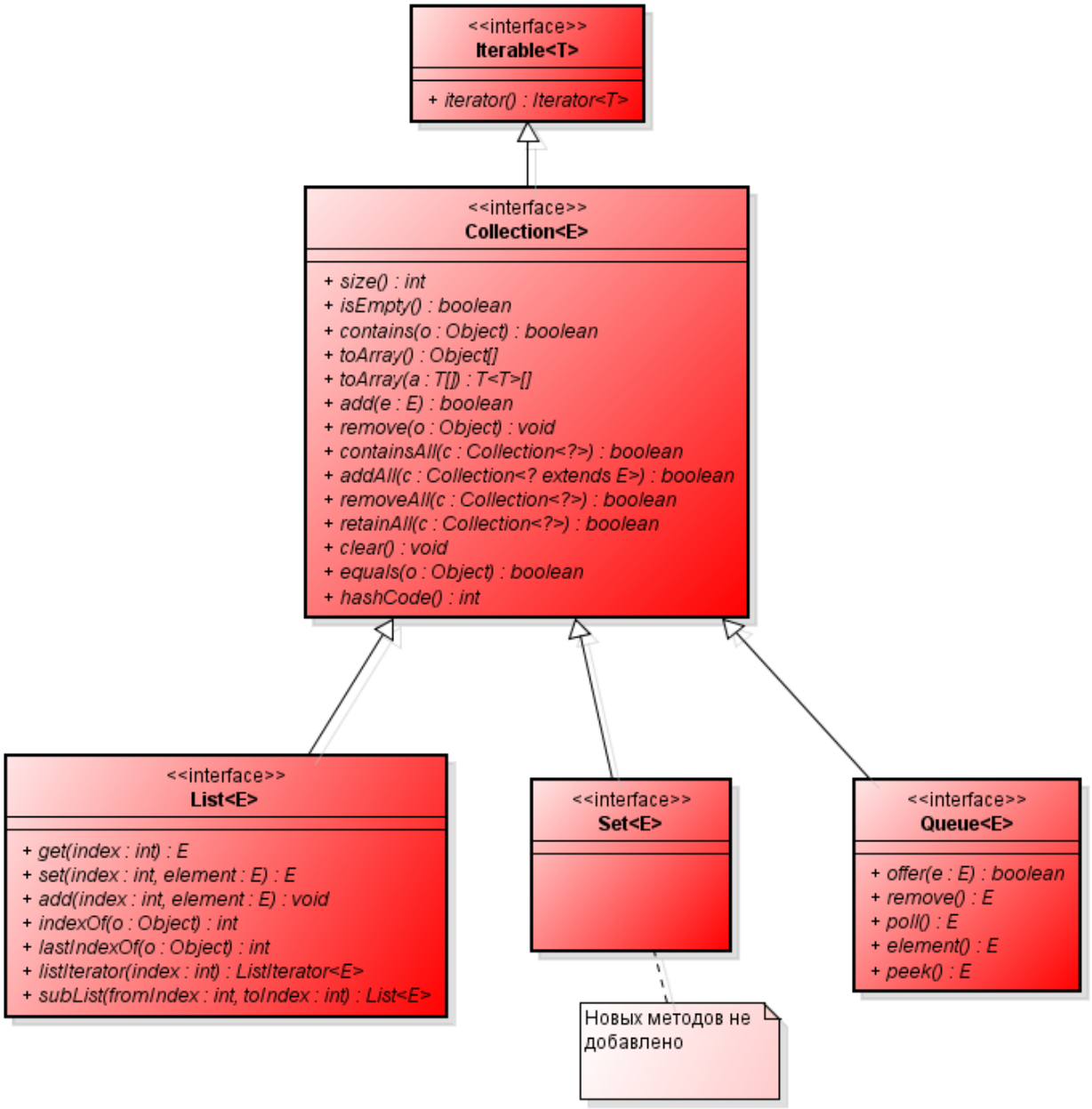

Рис. 4.2. Верхня частина ієрархії `Collection`

Map <K, V> описує колекцію, що складається з пар "*ключ - значення*". Для кожного ключа зберігається тільки одне значення, що відповідає математичному поняттю однозначної функції або відображення. Тут K вказує тип ключів, а V – тип збережених значень. Таку колекцію часто називають ще *словником* (dictionary) або *асоціативним масивом* (associative array). Зверніть увагу, що `Map` не відноситься до інтерфейсу `Collection` і є самостійним інтерфейсом. В інтерфейсі `Map` основною операцією є пошук *значення по ключу* – і ця операція виконується дуже швидко, за постійний час, так само як і додавання/видалення пар ключ-значення. Також із відображення (`Map`) можна отримати множину (`Set`) ключів і список (`List`) значень.

Кожен інтерфейс колекції реалізується декількома класами:

- Інтерфейс `List<E>` реалізується класами `LinkedList<E>` і `ArrayList<E>`, які ми розглянемо далі.
- Інтерфейс `Set<E>` реалізується класами `HashSet<E>` (звичайна множина) і `TreeSet<E>` (впорядкована множина).
- Інтерфейс `Map<K, V>` реалізується класами `HashMap<K, V>`, `LinkedHashMap<K, V>`, `TreeMap<K, V>`.

Класи, що реалізують один інтерфейс, мають однаковий набір методів, і тому використовуються в програмі абсолютно аналогічно. Але, з іншого боку, кожен клас використовує власне внутрішнє представлення даних, через що він може більш ефективно виконувати одні операції, а інші – менш ефективно. Ці особливості кожного класу потрібно враховувати при виборі, який з них використовувати, оскільки в разі невдалого вибору робота програми може сповільнитися в десятки і навіть сотні разів. Далі розглянемо найбільш часто використовувані класи-колекції.

ArrayList<E>

Це найбільш часто використовувана колекція. `ArrayList` реалізує роботу зі списком за допомогою звичайного масиву, довжина якого автоматично збільшується при додаванні нових елементів. Оскільки `ArrayList` використовує масив, то час доступу до елементу за індексом мінімальний (на відміну від `LinkedList`).

З іншого боку, при видаленні довільного елемента зі списку, всі елементи, що знаходяться «правіше», зміщуються на одну комірку масиву вліво. Це досить тривала операція. Важливо пам'ятати, що при видаленні і додаванні елементів реальний розмір масиву (його ємність, capacity) не змінюється. Якщо при додаванні елемента виявляється, що масив повністю заповнений, буде створено новий масив розміром (n * 3) / 2 + 1, де n – попередній розмір масиву, в нього будуть поміщені всі елементи зі старого масиву і додано один новий елемент.

Таким чином `ArrayList` має постійний час доступу за номером і лінійний час для всіх інших операцій. Розглянемо роботу зі списком `ArrayList` на прикладі.

Приклад. Створимо список дробів (тобто об'єктів створеного раніше класу `Ration`) і покажемо різні способи перебору списку а також сортування та випадкове перемішування елементів.

```
ArrayList<Ration> lst = new ArrayList<Ration>(10);
for(int i=0; i<5; i++ )
     lst.add( Ration.random(10) );
Ration r1  = lst.get(0);              // Отримати перший елемент
lst.add(2, new Ration (0,1) );        // Вставити в 3-ю позицію
lst.add(2, new Ration (0,1) );        // Вставити в 3-ю позицію
lst.remove(2);                        // Видалити 3-й елемент
lst.set( lst.size()-1, new Ration (1,1) );  // Заміна елемента
out( lst.toString() );


// Пройти по колекції, використовуючи цикл for
for( Ration r : lst) {
     r.add(new Ration(1,1));                // Елемент колекції змінюється
     r = Ration.add(r, new Ration(1,1));    // Не впливає на елементи
                                            // колекції
}
out( lst.toString() );

// Перебрати колекцію, використовуючи ітератор
Iterator<Ration> it = lst.iterator();
while(it.hasNext()) {
     it.next().add(new Ration(1,1));
}
out( lst.toString() );

Collections.shuffle(lst);    // Перемішати елементи
out( lst.toString() );

Collections.sort(lst);       // Відсортувати у відповідності до compareTo()
out( lst.toString() );
```

LinkedList<E>

Реалізує ті ж методи, що і `ArrayList`, але для внутрішнього представлення даних використовує двонаправлений список. Доступ до довільного елементу за номером в такому списку вимагає переглядати список від самого початку, і тому здійснюється за лінійний час. При цьому доступ до першого і останнього елементів списку завжди здійснюється за константний час, оскільки посилання на перший та останній елементи постійно зберігаються в об'єкті `LinkedList`. В цілому ж, як за швидкістю роботи, так і по споживаній пам'яті `LinkedList` в більшості випадків програє `ArrayList`, але цей клас незамінний у випадках, коли часто доводиться вставляти і видаляти елементи в середині списку, або ж загальна кількість елементів змінюється в широких межах.

HashMap<V,K>

Як вже говорилося, даний клас реалізує *асоціативний масив*, тобто набір пар виду *ключ-значення*. При цьому ключі не повторюються. У будь-який момент можна отримати елемент-значення, асоційоване (тобто що знаходиться в парі) із заданим ключем. У вигляді асоціативного масиву зручно зберігати, наприклад, рядок, отриманий з бази даних. В цьому випадку ключем буде назва стовпця, а значенням – значення цього стовпця в даному рядку. Можна сприймати асоціативний масив як різновид звичайного масиву, в якому індексами, за якими здійснюється доступ до елементів, є не цілі числа, а довільні об'єкти – найчастіше рядки.

Інтерфейс `Map` містить методи для роботи з асоціативним масивом:

`size()`	повертає кількість елементів (пар) в масиві;
`containsKey(Object key)`	перевіряє, чи існує в масиві елемент з ключем key;
`containsValue(Object value)`	перевіряє, чи існує в масиві елемент зі значенням value;
`get(Object key)`	повертає значення, що відповідає ключу key;
`put(Object key, Object value)`	додає в масив елемент з ключем key і значенням value. Якщо елемент з таким ключем вже існує в масиві, то його значення просто змінюється;
`values()`	повертає значення всіх елементів масиву у вигляді колекції (тобто, результат, що повертається, має тип `Collection`);
`remove(Object key)`	видаляє елемент з ключем key, повертаючи значення цього елемента (якщо він є) і null, якщо такого елемента немає;
`clear()`	очищає масив;
`isEmpty()`	перевіряє, чи не порожній масив.

Покажемо основні прийоми роботи на прикладі колекції, в якій ключами є об'єкти нашого класу дробів `Ration`, а значеннями – рядки. Зверніть увагу, для того щоб клас можна було використовувати в якості ключа `Map`, в ньому повинні бути перевизначені методи `hashCode()` і `equals()`, як обговорювалося в попередній лекції.

Приклад. Прийоми роботи з `Map`

```
// Створюємо Map
HashMap<Ration, String> map = new HashMap<Ration, String>(20);
Ration r;

// Додавання пар
for(int i=0; i<5; i++ )
     map.put( r = new Ration(i,i+1), r.toString() );

// Заміна і видалення значень
r = new Ration(3,4);
map.put(r, "aaa");
map.remove( new Ration (2,3));
out( map.toString() );
```

```
// Map не является Collection, тому для нього не можна використовувати
// ітератори і цикл виду for( Ration r : map)

// Знайти значення за ключем
if ( map.containsKey(r) )
     out( map.get( r ) );

// Прохід за допомогою ключів
Set<Ration> set = map.keySet();
for(Ration rr : set) {
     out("key = " + rr + " value = " + map.get(rr) );
}

// Пошук за значенням
out( "" + map.containsValue( "3/4" ) );

// Прохід по значенням
// (отримувана колекція містить копії посилань на значення)
Collection<String> col = map.values();
String ss = "";
for(String s : col) ss += " " + s;
out( ss );

// Зміна map через колекцію col
col.remove( "3/4" );
out( map.toString() );
```

Застарілі колекції

Наступні класи колекцій є застарілими, і їх використання не рекомендується, але не забороняється.

1. `Enumeration` – аналог інтерфейсу `Iterator`.

2. `Vector` – аналог класу `ArrayList`; підтримує впорядкований список елементів, що зберігаються у "внутрішньому" масиві.

3. `Stack` – клас, похідний від `Vector`, в який додані методи включення (push) і виключення (pop) елементів, так що список може трактуватися в термінах, прийнятих для опису структури даних стека.

4. `Dictionary` – аналог інтерфейсу `Map`, хоча є абстрактним класом, а не інтерфейсом.

5. `Hashtable` – аналог реалізації `HashMap`.

Всі методи `Hashtable`, `Stack`, `Vector` є ***синхронізованими***, що робить їх менш ефективними в додатках з одним потоком виконання.

Класс java.util.Collections

Містить декілька зручних методів для роботи з колекціями:

`emptyList(); emptyMap(); emptySet();` – створення порожніх колекцій, тільки для читання;

`min(collection); max(collection);` – пошук мінімального або максимального елемента в колекції. Елементи порівнюються за допомогою методу `compareTo()`, тобто вони повинні реалізовувати інтерфейс `Comparable`;

`sort(lst);` – сортування елементів колекції в порядку, що встановлюється методом порівняння `compareTo()`;

`shuffle (lst);` – випадкове перемішування елементів колекції;

`binarySearch(lst,key);` – бінарний пошук елемента в колекції. Працює набагато швидше звичайного пошуку, але застосовується тільки в тому разі, якщо колекція попередньо відсортована;

`indexOfSublist(lst,lstSub);` – знайти заданий підсписок lstSub в списку lst;

`unmodifiableList (list); unmodifiableSet (list); unmodifiableMap (map);`
– повертають *незмінне представлення* колекції.

**** Синхронізовані колекції***

Отримати синхронізовані об'єкти колекцій можна за допомогою статичних методів `synchronizedMap()` і `synchronizedList()` класу `Collections`.

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

Синхронізовані обгортки колекцій `synchronizedMap()` і `synchronizedList()` іноді називають *умовно* потоко-безпечними: всі методи цих класів окремо потокобезпечні, тобто не можуть бути перервані іншим потоком. Однак необхідно пам'ятати, що *послідовність викликів цих методів* цілком може бути перервана іншим потоком, що може стати причиною конкуренції за дані або порушення цілісності даних. Тому таку послідовність викликів слід завжди розташовувати всередині синхронізованої секції (`synchronized`). Синхронізація потоків і критичні секції розглядаються далі в цьому курсі.

Таким чином, умовна безпека потоку, яку забезпечують `synchronizedMap()` і `synchronizedList()` представляє приховану загрозу: розробники вважають, що, раз ці колекції синхронізовані, значить, вони повністю потоко-безпечні, і нехтують належною синхронізацією складних операцій. В результаті, хоча такі програми і працюють при легкому навантаженні, але при серйозному навантаженні вони можуть почати викидати виключення `NullPointerException` або `ConcurrentModificationException`.

Виключення

Виключеннями або винятковими ситуаціями (станами) називаються помилки, що виникли в програмі під час її роботи (рис. 4.3).

Всі виключення в Java є об'єктами. Тому вони можуть породжуватися не тільки автоматично при виникненні виняткової ситуації, але і створюватися самим розробником.

Рис. 4.3. Типи помилок в програмі

Всі виключення мають спільного предка – клас **Throwable** (рис. 4.4). Його нащадками є підкласи **Exception** і **Error.**

Виключення (**Exception**) є результатом проблем в програмі, які в принципі можна вирішити і передбачити. Наприклад, якщо має місце цілочислове ділення на нуль, то цьому завжди можна запобігти, перевіривши операнди перед діленням.

Помилки (**Error**) представляють собою більш серйозні проблеми, які, відповідно до специфікації Java, не слід намагатися обробляти у власній програмі, оскільки вони пов'язані з проблемами рівня JVM. Наприклад, виключення такого роду виникають, якщо закінчилася пам'ять, доступна віртуальній машині. Навіть якщо програма містить обробку такої ситуації, вона все одно не зможе забезпечити додаткову пам'ять для JVM.

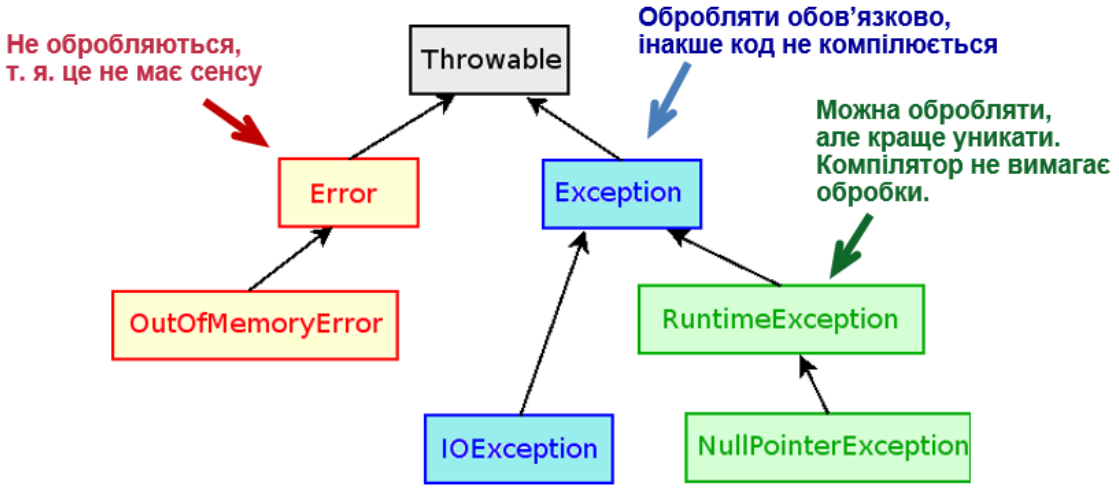

Рис. 4.4. Загальна ієрархія класів виключень

В Java всі виключення поділяються на два типи:

1) ***неконтрольовані виключення (unchecked),*** до яких відносяться помилки (клас **Error**) і виключення часу виконання (клас **RuntimeExceptions**, один з нащадків класу `Exception`). Неконтрольовані виключення не вимагають обов'язкової обробки, проте, при бажанні, можна обробляти виключення класу `RuntimeException`.

2) ***контрольовані виключення (checked)*** – це помилки, які можна і потрібно обробляти в програмі. Це все нащадки класу `Exception`, крім `RuntimeException` і його нащадків (рис. 4.5).

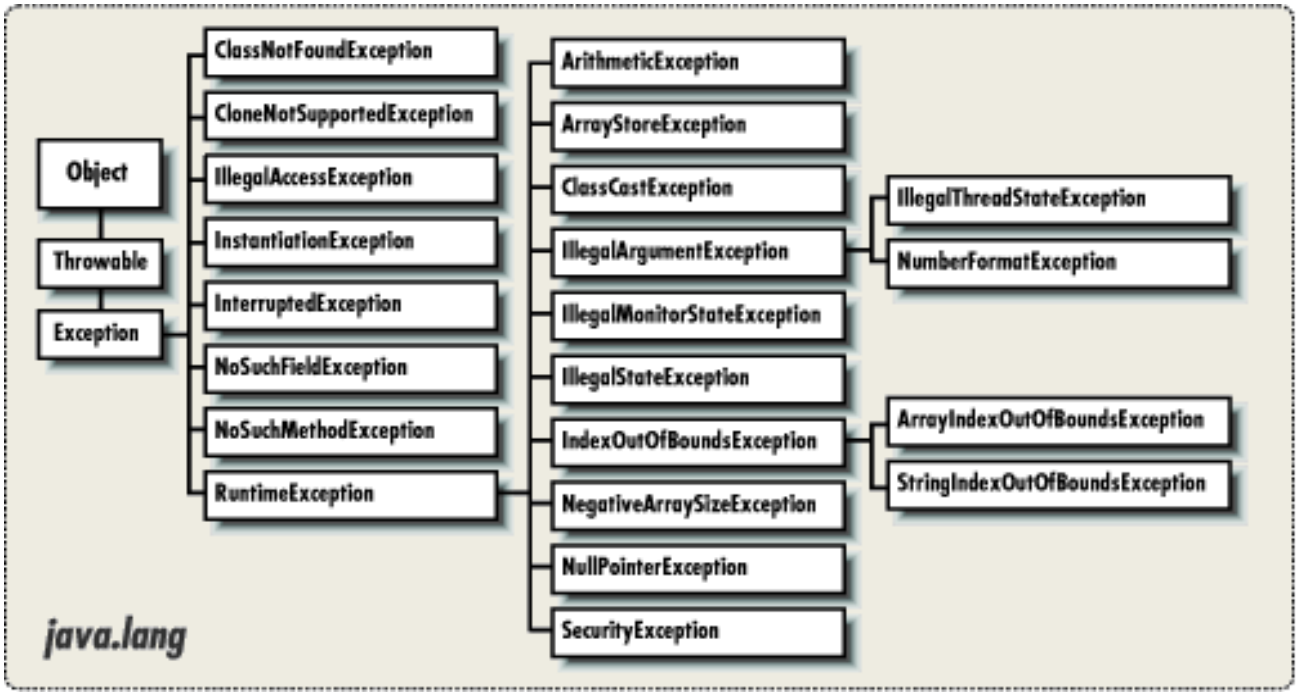

Рис. 4.5. Класи виключень

Наприкладр, код

```
int a = 4;
System.out.println(a/0);
```

генерує виключення класу `ArithmeticException`:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:4)
```

З повідомлення видно клас виключення, що відбулося – `ArithmeticException` і рядок коду, де воно відбулося.

До механізму обробки виключень в Java мають відношення 5 ключових слів: – **try, catch, throw, throws** і **finally**. Схема роботи цього механізму наступна. Ви намагаєтеся (`try`) виконати блок коду, і якщо при цьому виникає помилка, то система генерує (`throw`) виключення. Залежно від типу виключення ви можете його перехопити (`catch`) або передати далі за кодом. Також можна окремо визначити дії, які будуть виконуватися гарантовано (`finally`), незалежно від того, відбулася помилка чи ні.

Загальна форма блока з обробкою виключень має наступний вигляд:

```
try {
     // блок кода
} catch (ТипВиключення e1) {
     // обробник виключень типу ТипВиключення1
} catch (ТипВиключення2 e2) {
     // обробник виключень типу ТипВиключення2
     throw(e2);   // можливе повторне збудження виключення
} finally {
     // код, який виконується в будь-якому випадку
}
```

Приклад обробки різних видів виключень:

```
int[] m = {-1,0,1};
int a = 1;
Scanner sc = new Scanner(System.in);
try {
    a = sc.nextInt();
    m[a-1] = 4/a;
    System.out.println(m[a]);
} catch (ArithmeticException e) {
    System.out.println("Недопустима арифметична операція");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Недопустимий індекс масиву");
} catch (Exception e) {
    e.printStackTrace();
}
```

Якщо користувач введе 0, то виникне виключення класу `ArithmeticException` (ділення на нуль), і воно буде оброблено першим блоком catch.

Якщо користувач введе 3, то виникне виключення класу `ArrayIndexOutOfBoundsException` (вихід за межі масиву), і воно буде оброблено другим блоком `catch`.

Якщо користувач введе нецілe число, наприклад, 3.14, то виникне виключення класу `InputMismatchException` (невідповідність типу введеного значення). Спеціального блоку `catch` для такого класу виключень в нашій програмі немає, але воно буде оброблено в останньому блоці `catch(Exception e)`, оскільки клас `Exception` є батьківським для всіх контрольованих виключень, і, відповідно об'єкт класу `InputMismatchException` також є об'єктом класу `Exception`.

Оскільки весь механізм виключень побудований на ієрархії класів, *то спочатку треба намагатися обробити більш спеціалізовані виключення і лише потім більш загальні.*

Зупинимося трохи докладніше на блоці **finally**. Він може знаходиться після блоків `try` і `catch`. Вміщені в нього команди будуть виконуватися в будь-якому випадку, незалежно від того, чи відбулося виключення чи ні, і чи було воно оброблене. Наприклад, якщо виключення не було оброблене, то решта програми не виконуватиметься, і при цьому можуть залишитися незвільнені ресурси (незакриті потоки, розпочаті і не закінчені транзакції доступу до БД і т. п.). При цьому в блоці `finally` можна гарантовано звільнити всі ресурси.

Ключове слово **throws** вказується при оголошенні методу, щоб повідомити клієнту, які виключення цей метод може генерувати. Якщо `throws` зазначено, то при використанні методу цей тип виключення обов'язково потрібно обробляти, тобто виклик методу повинен обов'язково знаходитись всередині секції `try ... catch`. Наприклад, в класі `java.io.File` є метод створення нового файлу, який оголошений таким чином:

```
public boolean createNewFile() throws IOException
```

Він генерує виключення класу IOException, якщо файл не вдалося створити. IOException відноситься до контрольованих виключень, і тому його:

– потрібно обов'язково вказати при описі методу: `throws IOException`;
– обов'язково потрібно обробляти, наприклад:

```
File f = null;
Try {
   f = new File("test.txt");    // Створюємо об’єкт-файл

   f.createNewFile();           // Намагаємось створити фізичний файл
                                // з іменем test.txt
} catch(IOException e){         // Обробляємо можливі помилки
   e.printStackTrace();
}
```