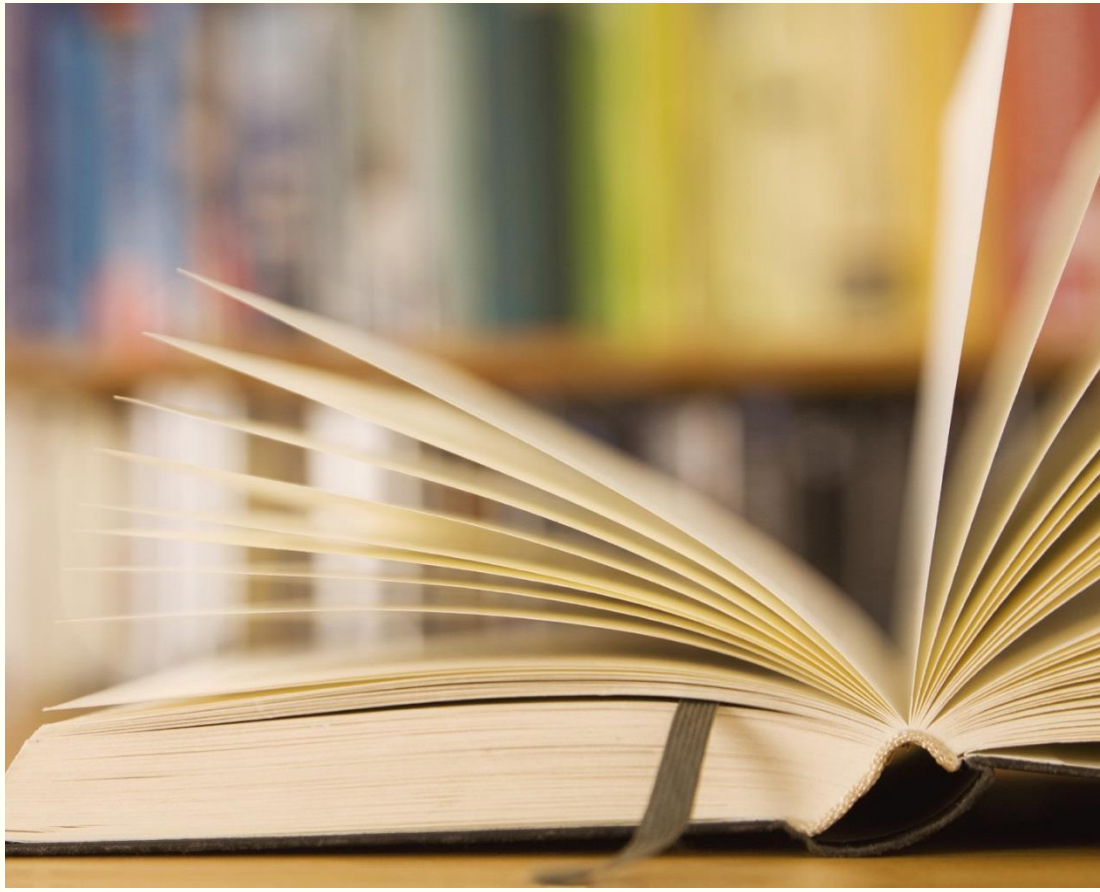
КОЛЕКЦІЇ ТА КЛАСИЧНИЙ ВВІД-ВИВІД

Лекція 05
Java-програмування



План лекції

- Архітектура Collections Framework.
- Поширені структури даних з Collections Framework.
- Функціональні операції з колекціями за допомогою функцій вищого порядку.
- Файловий ввід-вивід у мові програмування Java.
- Робота з потоками даних, райтерами та рідерами.

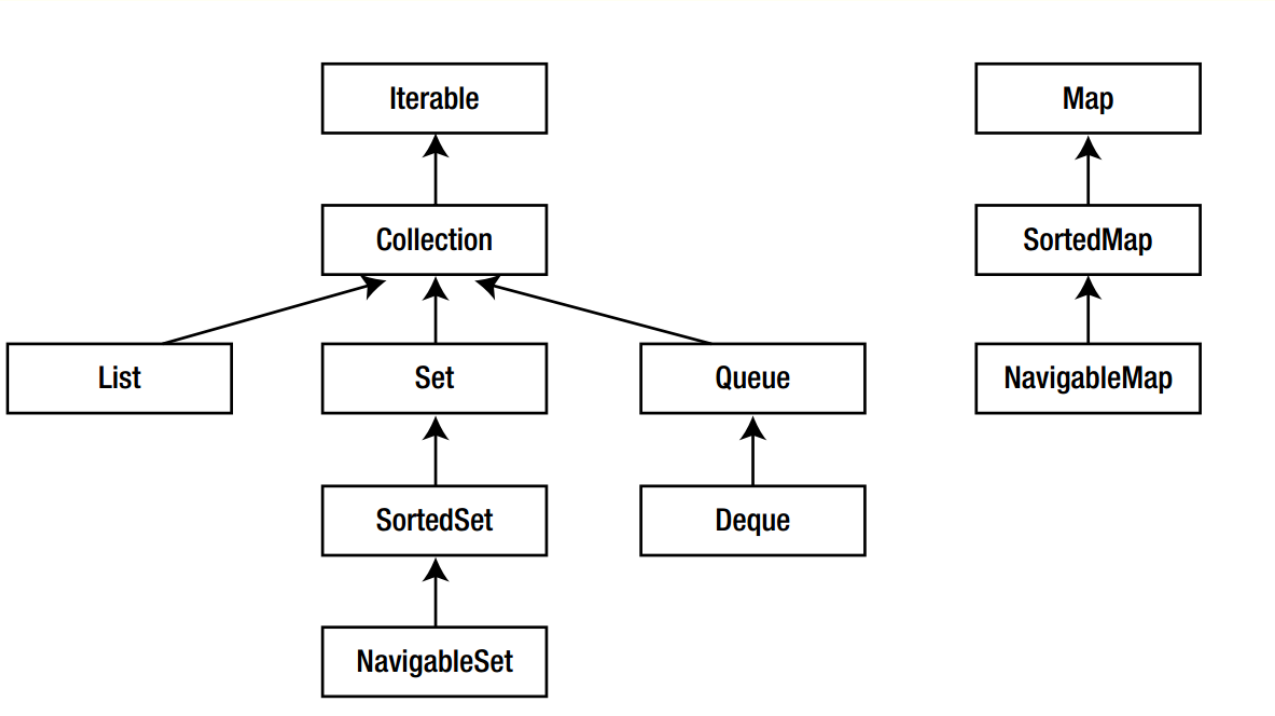
АРХІТЕКТУРА COLLECTIONS FRAMEWORK

Питання 5.1.

Вступ

- Додатки часто мають управляти наборами (collections) об’єктів.
- Хоч для цього можна використовувати масиви, вони не завжди є хорошим вибором.
  - Наприклад, масиви мають фіксовані розміри, що значно ускладнює визначення оптимального розміру при зберіганні змінної кількості об’єктів.
  - Також масиви можуть індексуватись лише цілочисельними значеннями, що робить їх непідходящими для відображення довільних об’єктів у інші об’єкти.
- Стандартна бібліотека класів постачає Collections Framework та legacy utility APIs для управління колекціями в додатках.
  - **Зауважте!** Java’s Concurrency Utilities API suite розширює Collections Framework.
- *Collections Framework – група типів* (в основному міститься в пакеті java.util), які пропонують стандартну архітектуру для представлення та маніпуляції колекціями (наборами об’єктів, що зберігаються в екземплярах спроектованих для цього класів).

Архітектура Collections Framework



- Архітектура фреймворку базується на 3 секціях:
  - ***Базові інтерфейси*** *(Core interfaces)*: інтерфейси для маніпулювання колекціями незалежно від їх реалізації.
  - ***Класи реалізації*** *(Implementation classes)*: класи, що пропонують реалізацію різних базових інтерфейсів для підвищення продуктивності та інших вимог.
  - ***Допоміжні класи*** *(Utility classes)*: класи з методами для сортування масивів, отримування synchronized collections та ін.

Архітектура Collections Framework

- Класи реалізації для фреймворка:
  - ArrayList, LinkedList,
  - TreeSet, HashSet, LinkedHashSet, EnumSet,
  - PriorityQueue, ArrayDeque,
  - TreeMap, HashMap, LinkedHashMap, IdentityHashMap, WeakHashMap та EnumMap.
- Назва кожного конкретного (не абстрактного) класу закінчується ім'ям відповідного йому базового (core) інтерфейсу.
  - **Зауважте!** Додаткові класи реалізації є частиною concurrency utilities.
- Класи реалізації також включають абстрактні класи
  - AbstractCollection, AbstractList, AbstractSequentialList, AbstractSet, AbstractQueue, AbstractMap.
  - Вони пропонують каркасну реалізацію базових інтерфейсів для підтримки створення конкретних класів реалізації.
- Також фреймворк постачає два допоміжних класи: Arrays та Collections.

Comparable vs. Comparator

- Реалізація колекції зберігає елементи в певному порядку (arrangement): несортованому або відсортованому за певним критерієм.
  - Відсортована реалізація колекції зберігає за замовчуванням елементи в ***природному порядку (natural ordering)***.
  - Наприклад, для об'єктів java.lang.String природний порядок – ***лексикографічний, або словарний (dictionary, alphabetical)***.
- Метод equals() лише може визначати еквівалентність двох елементів і не може диктувати природний порядок.
  - Замість цього element класи мають реалізовувати інтерфейс java.lang.Comparable<T> та його метод int compareTo(T o).
  - Цей інтерфейс вважається частиною Collections Framework, навіть незважаючи на те, що він знаходиться в пакеті java.lang.
- compareTo() порівнює аргумент o з поточним елементом (для якого метод викликається) та:
  - Повертає від'ємне значення, коли поточний елемент має передувати o.
  - Повертає 0, коли поточний елемент та o - однакові.
  - Повертає додатне значення, коли поточний елемент має слідувати за o.

Правила реалізації compareTo()

- При реалізації методу compareTo() інтерфейсу Comparable потрібно слідувати деяким правилам:
  - compareTo() *має бути рефлексивним*: *x*.compareTo(*x*) повертає 0.
  - compareTo() *має бути симетричним*: *x*.compareTo(*y*) == -*y*.compareTo(*x*).
  - compareTo() *має бути транзитивним*: якщо *x*.compareTo(*y*) > 0 і *y*.compareTo(*z*) > 0 істинні, то *x*.compareTo(*z*) > 0 – теж істинний.
- compareTo() має викидати java.lang.NullPointerException, коли в метод передається null.
  - потреби перевіряти на null немає, оскільки цей метод викидає NullPointerException при спробі доступу до неіснуючих членів null reference.

Компаратор

- Інколи Вам може бути потрібним збереження в колекції об’єктів, відсортованих у певному порядку, відмінному від природного.
  - Тоді потрібно постачати *компаратор* для забезпечення впорядкованості.
  - ***Компаратор*** – об’єкт, чий клас реалізує інтерфейс Comparator.
- Цей інтерфейс (узагальнений тип Comparator<T>) постачає пару методів:
  - int compare(T o1, T o2) порівнює обидва аргументи для впорядкування. Повертає 0, коли o1 = o2, від’ємне значення, коли o1 < o2, додатне – коли більше.
  - boolean equals(Object o) повертає true, коли o також є Comparator та накладає те ж упорядкування. Інакше метод поверне false.
- Comparator оголошує метод equals().
  - Цей метод може повертати true лише тоді, коли заданий об’єкт також є компаратором і передбачає те ж упорядкування, як і компаратор.
  - Вам не потрібно переозначувати equals(), проте роблячи так, Ви можете покращити швидкодію, дозволяючи програмам визначати, що два окремі компаратори мають однаковий порядок.

Iterable та Collection

- Більшість базових інтерфейсів походять від Iterable та його підінтерфейсу Collection.
  - Їх узагальнені типи – Iterable<T> та Collection<E>.
- Iterable описує будь-який об’єкт, що може повернути об’єкти, які він містить, у певній послідовності.
  - Інтерфейс оголошує метод Iterator<T> iterator(), який повертає екземпляр Iterator для ітерування по всіх внутрішніх (contained) об’єктах.
- Collection представляє колекцію об’єктів – елементів.
  - Інтерфейс постачає методи, які спільні для підінтерфейсів Collection та на яких багато колекцій базуються.

Таблиця методів інтерфейсу Collection

Метод	Опис
boolean add(E e)	Додає елемент ***e*** до колекції. Повертає true, якщо колекція змінилась у результаті; інакше – false. Метод викидає: ***java.lang.UnsupportedOperationException***, коли add() не підтримується ***ClassCastException***, коли клас об'єкту ***e*** недоречний для колекції ***java.lang.IllegalArgumentException***, коли деяка властивість об'єкту ***e*** запобігає його додаванню в колекцію ***NullPointerException***, коли ***e*** містить null-посилання, а колекція не підтримує null-елементи ***java.lang.IllegalStateException***, коли елемент неможливо додати цього разу через обмеження вставки. Цей виняток сигналізує про невчасний виклик методу, часто викидається при вставці елементу в заповнену обмежену (bounded) чергу.
boolean addAll(Collection<? extends E> c)	Додає всі елементи колекції ***c***. Повертає true, якщо колекція змінилась у результаті; інакше – false. Метод викидає аналогічні винятки, як і з методом add().
void clear()	Видаляє всі елементи з колекції. Викидає UnsupportedOperationException, коли дана колекція не підтримує clear().

Метод	Опис
boolean contains(Object o)	Повертає true, якщо колекція містить ***o***; інакше – false. Метод викидає: ***ClassCastException***, коли клас об'єкту ***o*** недоречний для колекції ***NullPointerException***, коли ***o*** містить null-посилання, а колекція не підтримує null-елементи
boolean containsAll(Collection<?> c)	Повертає true, якщо колекція містить усі елементи з колекції ***c***; інакше – false. Метод викидає аналогічні винятки, як і з методом contains().
boolean equals(Object o)	Порівнює ***o*** з даною колекцією та повертає true, якщо вони рівні; інакше – false.
int hashCode()	Повертає хеш-код колекції. Однакові об'єкти мають однаковий хеш-код
boolean isEmpty()	Повертає true, якщо колекція не містить елементів; інакше – false.
Iterator<E> iterator()	Повертає екземпляр Iterator для ітерування по всіх елементах колекції. Впорядкованість елементів не гарантується, якщо відповідний клас цього не передбачає. Цей метод з інтерфейсу Iterable переоголошується в інтерфейсі Collection для зручності.

`boolean remove(Object o)`	Removes the element identified as `o` from this collection. Returns true when the element is removed; otherwise, returns false. This method throws `UnsupportedOperationException` when this collection doesn't support `remove()`, `ClassCastException` when the class of `o` is inappropriate for this collection, and `NullPointerException` when `o` contains the null reference and this collection doesn't support null elements.
`boolean removeAll(Collection<?> c)`	Removes all of the elements from this collection that are also contained in collection `c`. Returns true when this collection is modified by this operation; otherwise, returns false. This method throws `UnsupportedOperationException` when this collection doesn't support `removeAll()`, `ClassCastException` when the class of one of `c`'s elements is inappropriate for this collection, and `NullPointerException` when `c` contains the null reference or when one of its elements is null and this collection doesn't support null elements.
`Object[] toArray()`	Returns an array containing all of the elements stored in this collection. If this collection makes any guarantees as to what order its elements are returned in by its iterator, this method returns the elements in the same order. The returned array is "safe" in that no references to it are maintained by this collection. (In other words, this method allocates a new array even when this collection is backed by an array.) The caller can safely modify the returned array.
`<T> T[] toArray(T[] a)`	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it's returned in the array. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection. This method throws `NullPointerException` when `null` is passed to `a`, and `java.lang.ArrayStoreException` when `a`'s runtime type is not a supertype of the runtime type of every element in this collection.

Особливості методів з Collection

`boolean retainAll(Collection<?> c)`	Retains all of the elements in this collection that are also contained in collection `c`. Returns true when this collection is modified by this operation; otherwise, returns false. This method throws `UnsupportedOperationException` when this collection doesn't support `retainAll()`, `ClassCastException` when the class of one of `c`'s elements is inappropriate for this collection, and `NullPointerException` when `c` contains the null reference or when one of its elements is null and this collection doesn't support null elements.
`int size()`	Returns the number of elements contained in this collection, or `java.lang.Integer.MAX_VALUE` when there are more than `Integer.MAX_VALUE` elements contained in the collection.

- Деякі методи можуть викидати UnsupportedOperationException.
  - Наприклад, метод add() викидає його при спробі додати об'єкт до *immutable* (unmodifiable) колекції.
- Деякі методи з Collection можуть викидати ClassCastException.
  - Наприклад, метод remove() при спробі видалити елемент (entry, also known as mapping) з tree-based map, якщо потрібні ключі типу String, проте задаються іншого типу.
- Методи add() та addAll() викидають IllegalArgumentException, коли певна властивість (*property, attribute*) елементу, яку слід додати, не дає цьому елементу додатись в колекцію.
  - Наприклад, методи add() та addAll() класу сторонньої (third-party) колекції можуть викидати це виключення, коли detect від'ємні Integer-значення.

Зауважте!

- Чому методу remove() оголошується з аргументом типу Object, а не конкретним типом колекції?
  - Чому remove() не оголошується як boolean remove(E e)?
  - Чому методи containsAll(), removeAll(), retainAll() не оголошені з аргументом типу Collection<? extends E>, щоб переконатись, що collection argument містить лише той же тип, що і колекція, для якої ці методи викликані?

- Відповідь: підтримка зворотної сумісності.
  - Collections Framework були представлені до Java 5 – раніше дженериків.
  - Для компіляції legacy code, написаного до п’ятої версії, ці 4 методи оголошуються зі слабшими (weaker) обмеженнями на тип.

Iterator та Enhanced For

- Розширюючи Iterable, інтерфейс Collection наслідує той варіант методу iterator(), який дозволяє прохід по колекції.
- Метод iterator() повертає екземпляр класу, який реалізує інтерфейс Iterator (узагальнений тип Iterator<E>) з методами:
  - **boolean hasNext()** повертає true, коли цей екземпляр Iterator ще має елементи для повернення; інакше – false.
  - **E next()** повертає наступний елемент з колекції, пов'язаної з екземпляром Iterator або викидає NoSuchElementException, коли більше нічого повертати.
  - **void remove()** видаляє останній елемент, що повернувся за допомогою next(), з колекції, пов'язаної з екземпляром Iterator.
    - Цей метод може викликатись лише один раз на кожен виклик next().
    - Поведінка екземпляру Iterator не задана, коли відповідна колекція змінюється, поки ітерація в процесі виконання будь-яким іншим чином, ніж викликом remove().
    - Метод викидає UnsupportedOperationException, коли він не підтримується цим інтерфейсом Iterator, і IllegalStateException, коли remove() викликали без попереднього виклику next() або сталось декілька викликів remove() без відповідних викликів next().

Приклад проходу по колекції після виклику методу iterator() to return an Iterator instance:

```
Collection<String> col = ... // This code doesn't compile because of the ...
// Add elements to col.
Iterator iter = col.iterator();
while (iter.hasNext())
   System.out.println(iter.next());
```

- Цикл while викликає метод hasNext() ітератора для визначення того, чи продовжувати прохід (iteration).
  - Якщо так, метод next() повертає наступний елемент з відповідної (associated) колекції.
- Через велику поширеність такого коду в Java 5 представлено «синтаксичний цукор» 5 для циклу for.
  - Запис стає схожий на оператор foreach з інших мов програмування (Perl, C#) :

```
Collection<String> col = ... // This code doesn't compile because of the ...

// Add elements to col.
for (String s: col)
       System.out.println(s);
```

- Інтерпретація наступна: “для кожного об’єкту типу String у колекції col, присвоїти цей об’єкт змінній s на початку loop iteration.”

Зауважте!

- Покращений for також корисний у контексті масивів, де приховується індекс масиву:

```
String[] verbs = { "run", "walk", "jump" };
for (String verb: verbs)
   System.out.println(verb);
```

- Еквівалентний код

```
String[] verbs = { "run", "walk", "jump" };
for (int i = 0; i < verbs.length; i++)
   System.out.println(verbs[i]);
```

  - Обмеженість: для видалення елементу з колекції потрібен ітератор, до якого немає доступу.
  - Також неможливо користуватись при паралельному проході по декількох колекціях або заміні елементів колекції/масиву протягом обходу.

Автоупаковка (Autoboxing) та розпаковка (Unboxing)

- Частина розробників вважає, що Java має підтримувати лише посилкові типи, тому вони скаржаться на наявність примітивних типів.
  - Одна з областей, де проявляється відмінність - Collections Framework: Ви можете зберігати об’єкти в колекції, проте зі значеннями непримітивних типів.

- Хоч напряму зберегти значення примітивного типу в колекції неможливо, можна спочатку обгорнути (wrapping) його в об’єкті, створеному з класу-обгортки (wrapper class), наприклад, Integer, а потім зберегти екземпляр цього класу в колекції:

```
Collection<Integer> col = ...; // Цей код не компілюється,
int x = 27;  // оскільки значення 27 зберігається опосередковано
col.add(new Integer(x));  // за допомогою Integer-об’єкту.
```

- Зворотна ситуація теж обтяжлива.
  - Коли захочете отримати int з колекції col, потрібно викликати метод intValue() класу Integer (який наслідується від суперкласу Integer – java.lang.Number).
  - Потрібно задати int y = col.iterator().next().intValue(); для присвоєння 32-бітного integer змінній y.

Автоупаковка (Autoboxing) та розпаковка (Unboxing)

- Для обходу труднощів у Java 5 представлено автоупаковку та розпаковку, що дозволяють значенням примітивних типів поводитись схоже до об’єктів.
  - Ця “спритність рук” неповна, оскільки не можна задати вирази на зразок 27.doubleValue().
- *Автоупаковка* автоматично обгортає (*boxes,* wraps) значення примітивного типу в об’єкт відповідного класу-обгортки.
  - Наприклад, можна змінити третій рядок прикладу на col.add(x); - компілятор обгорне x в Integer-об’єкт.
- *Розпаковка* автоматично розпаковує (*unboxes,* unwraps) значення примітивного типу з об’єкту-обгортки whenever a reference is specified but a primitive-type value is required.
  - Наприклад, можна задати int y = col.iterator().next(); та мати розпаковку компілятором поверненого Integer-об’єкта в ціле число 27 до виконання присвоєння.
- Хоч автоупаковка та розпаковка були представлені для спрощення роботи з примітивними типами в контексті колекцій, їх можна використовувати й у інших контекстах;
  - Довільне використання може призводити до проблеми, яку важко зрозуміти без знань про те, як відбуваються процеси зсередини.

Розглянемо приклад

```
Integer i1 = 127;
Integer i2 = 127;
System.out.println(i1 == i2); // Output: true
System.out.println(i1 < i2);  // Output: false
System.out.println(i1 > i2);  // Output: false
System.out.println(i1 + i2);  // Output: 254
i1 = 30000;
i2 = 30000;
System.out.println(i1 == i2); // Output: false
System.out.println(i1 < i2);  // Output: false
System.out.println(i1 > i2);  // Output: false
i2 = 30001;
System.out.println(i1 < i2);  // Output: true
System.out.println(i1 + i2);  // Output: 60001
```

- Вивід очікуваний, за винятком одного прикладу.
  - Порівняння i1 == i2, в якому i1 та i2 містять значення 30000.
  - Замість повернення true, як у випадку з i1 та i2 по 127, повертається false.
  - Integer i1 = 127; конвертується в Integer i1 = Integer.valueOf(127);
  - Integer i2 = 127; конвертується в Integer i2 = Integer.valueOf(127);
- Відповідно до документації, метод valueOf() використовує кешування для покращення продуктивності.
- **Зауважте!** Метод valueOf() також використовується при додаванні значень примітивного типу в колекцію.
  - Наприклад, col.add(27) конвертується в col.add(Integer.valueOf(27)) .

Кешування методу valueOf()

- Integer підтримує внутрішній кеш унікальних Integer-об’єктів з маленьким значеннями (від -128 до 127).
  - Проте верхню межу можна змінити, присвоївши нову системній властивості java.lang.Integer. IntegerCache.high за допомогою методу setProperty(String name, String value) класу String.
- **Зауважте!** Класи-обгортки java.lang.Byte, java.lang.Long та java.lang.Short також підтримують внутрішній кеш унікальних об’єктів типу Byte, Long та Short відповідно.
  - У зв’язку з наявністю кешу кожен виклик Integer.valueOf(127) повертає посилання на той же Integer-об’єкт, тому i1 == i2 (що порівнює посилання) визначає як true.
  - Оскільки 30000 лежить поза дефолтним діапазоном, кожен виклик Integer.valueOf(30000) повертає посилання на новий Integer-об’єкт, тому i1 == i2 визначається false.
- На відміну від == та !=, які не розпаковують обгорнені значення до порівняння, такі оператори як <, > та + розпаковують їх до виконання операцій з ними.
  - i1<i2 конвертується в i1.intValue()<i2.intValue()
  - i1+i2 конвертується в i1.intValue() + i2.intValue().
- **Обережно!** Не слід припускати, що автоупаковка та розпаковка використовуються в контексті операторів == та !=.