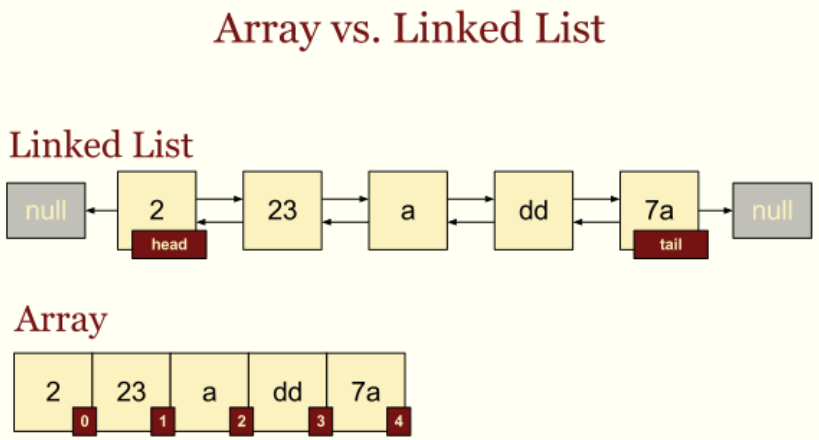
ПОШИРЕНІ СТРУКТУРИ ДАНИХ З COLLECTIONS FRAMEWORK

Питання 5.2.

Поняття списку

- ***Список*** – впорядкована колекція, яку також називають послідовністю.
  - Збереження та доступ до елементів може відбуватись за допомогою цілочисельних індексів.
  - Деякі елементи можуть дублюватись або дорівнювати null (якщо це допускає реалізація списку).
- Списки описуються за допомогою інтерфейсу List, чий узагальнений тип – List<E>.
  - Інтерфейс List розширює Collection та переоголошує успадковані методи, частково просто для зручності.
  - Також переозначаються методи iterator(), add(), remove(), equals(), hashCode(), щоб доповнити контракт додатковими умовами.
  - Наприклад, контракт для методу add() інтерфейсу List's вказує, що цей метод додає елемент у кінець списку, а не просто в колекцію.

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)



Доступні методи

Метод	Опис
void add(int index, E e)	Вставляє елемент ***e*** у список на позицію index. Зсовує наступні елементи вправо. Викидає • UnsupportedOperationException, коли список не підтримує add(), • ClassCastException, коли відповідний ***e*** клас непідходящий для списку, • IllegalArgumentException, коли певна властивість ***e*** не дає додатись елементу у список, • NullPointerException, коли e містить null-посилання, а список не підтримує null-елементи, • java.lang.IndexOutOfBoundsException, коли index < 0 або index >= size()
boolean addAll(int index, Collection <? extends E> c)	Вставляє всі елементи ***c*** у список, починаючи з позиції index та в порядку їх повернення ітератором ***c***. Зсуває поточний і наступні елементи вправо. Викидає • UnsupportedOperationException, коли список не підтримує addAll() • ClassCastException, коли клас одного з елементів ***c*** не підходить списку • IllegalArgumentException, коли деяка властивість елементу не дає додатись йому до списку • NullPointerException, коли c містить null-посилання чи один з елементів є null, а список не підтримує null-елементи • IndexOutOfBoundsException, коли index < 0 або index >= size()
E get(int index)	Повертає елемент, що зберігається у списку в позиції index. Викидає IndexOutOfBoundsException, коли index < 0 або index >= size()

Доступні методи

`int indexOf(Object o)`	Returns the index of the first occurrence of element `o` in this list, or `-1` when this list doesn't contain the element. This method throws `ClassCastException` when `o`'s class is inappropriate for this list and `NullPointerException` when `o` contains the null reference and this list doesn't support null elements.
`int lastIndexOf(Object o)`	Returns the index of the last occurrence of element `o` in this list, or `-1` when this list doesn't contain the element. This method throws `ClassCastException` when `o`'s class is inappropriate for this list and `NullPointerException` when `o` contains the null reference and this list doesn't support null elements.
`ListIterator<E> listIterator()`	Returns a list iterator over the elements in this list. The elements are returned in the same order as they appear in the list.
`ListIterator<E> listIterator(int index)`	Returns a list iterator over the elements in this list starting with the element located at `index`. The elements are returned in the same order as they appear in the list. This method throws `IndexOutOfBoundsException` when `index` is less than 0 or `index` is greater than `size()`.
`E remove(int index)`	Removes the element at position `index` from this list, shifts any subsequent elements to the left, and returns this element. This method throws `UnsupportedOperationException` when this list doesn't support `remove()` and `IndexOutOfBoundsException` when `index` is less than 0 or `index` is greater than or equal to `size()`.

`E set(int index, E e)`	Replaces the element at position `index` in this list with element `e` and returns the element previously stored at this position. This method throws `UnsupportedOperationException` when this list doesn't support `set()`, `ClassCastException` when `e`'s class is inappropriate for this list, `IllegalArgumentException` when some property of `e` prevents it from being added to this list, `NullPointerException` when `e` contains the null reference and this list doesn't support null elements, and `IndexOutOfBoundsException` when `index` is less than 0 or `index` is greater than or equal to `size()`.
`List<E> subList(int fromIndex, int toIndex)`	Returns a view (discussed later) of the portion of this list between `fromIndex` (inclusive) and `toIndex` (exclusive). (If `fromIndex` and `toIndex` are equal, the returned list is empty.) The returned list is backed by this list, so nonstructural changes in the returned list are reflected in this list and vice versa. The returned list supports all of the optional list methods (those methods that can throw `UnsupportedOperationException`) supported by this list. This method throws `IndexOutOfBoundsException` when `fromIndex` is less than 0, `toIndex` is greater than `size()`, or `fromIndex` is greater than `toIndex`.

- Таблиця відповідає інтерфейсу ListIterator, який більш гнучкіший, ніж суперінтерфейс Iterator:
  - ListIterator постачає методи для проходу по списку в будь-якому напрямку, змінюючи список впродовж ітерації та отримуючи поточну позицію ітератора в списку.

- **Зауважте!** Екземпляри Iterator та ListIterator, які повертають методи iterator() та listIterator() в класах реалізації інтерфейсу List (ArrayList та LinkedList), є *fail-fast*:
  - Коли список структурно змінюється (наприклад, при виклику add()) після створення ітератору будь-яким чином, крім через власні методи ітератора add() та remove(), ітератор викидає виняток ConcurrentModificationException.
  - Тому перед обличчям concurrent modification, ітератор не може працювати швидко та чисто, ризикуючи мати довільну поведінку в невизначений момент у майбутньому.

Методи ListIterator

- **void add(E e)** виконує вставку об’єкту e в список, по якому відбувається прохід.
  - Елемент вставляється відразу перед наступним елементом у списку (якщо такий є), який повертається за допомогою next(), а також після попереднього елементу (виклик previous()).
  - UnsupportedOperationException, коли даний ітератор списку не підтримує add(),
  - ClassCastException, коли клас елементу e не підходить для списку
  - IllegalArgumentException, коли деяка властивість e не дає йому додаватись до списку.
- **boolean hasNext()** повертає true, коли при обході списку в прямому напрямку ітератору ще доступні елементи.
- **boolean hasPrevious()** повертає true, коли при обході списку в зворотному напрямку ітератору ще доступні елементи.
- **E next()** повертає наступний елемент у списку та зміщує позицію курсора.
  - Викидає NoSuchElementException, коли наступний елемент відсутній.
- **int nextIndex()** повертає індекс елементу, який буде отримано після відповідного виклику next(), або розмір списку в кінці цього списку.
- **E previous()** повертає попередній елемент у списку та зміщує курсор назад.
  - Викидає NoSuchElementException, коли попереднього елемента не існує.

Методи ListIterator

- int previousIndex() повертає індекс елементу, який буде повернуто в результаті виклику previous(), або -1, коли знаходимось на початку списку.
- void remove() видаляє зі списку останній елемент, який повертає next() або previous().
  - Це можна зробити лише раз, викликавши next() або previous().
  - Крім того, можливість доступна лише тоді, коли add() не викликався після останнього next() чи previous().
  - UnsupportedOperationException, коли ітератор списку не підтримує remove()
  - IllegalStateException, коли ні next(), ні previous() не викликались або remove() чи add() уже були викликані після останнього виклику next() або previous().
- void set(E e) заміняє останній елемент, що повертають next() або previous() з елементом e.
  - Викликається лише тоді, коли ні remove(), ні add() не викликались після останнього виклику next() або previous().
  - UnsupportedOperationException, коли ітератор списку не підтримує set(),
  - ClassCastException, коли e не підходить для списку,
  - IllegalArgumentException, коли деякі властивості e не дають елементу додатись у список,
  - IllegalStateException, коли ні next(), ні previous() не викликались або remove() чи add() вже викликались після останнього виклику next() чи previous().

- Екземпляр ListIterator не має поняття поточного елементу.
  - Замість нього введено курсор для переміщення по списку.
  - Методи nextIndex() та previousIndex() повертають позицію курсору, яка завжди лежить між їх результатами виклику.
- Ітератор списку для списку довжиною *n* має *n*+1 можливу позицію курсору (ілюстрація ^):

```
                    Element(0)   Element(1)   Element(2)   ... Element(n-1)
cursor positions:  ^            ^            ^            ^                 ^
```

- Ви можете комбінувати виклики next() та previous(), проте будьте обережними.
  - Майте на увазі, що перший виклик previous() повертає той же елемент, що й останній виклик next().

Зауважте!

- Опис методу subList() з таблиці посилається на поняття представлення (*view)*, which is a list that is backed by another list.
  - Зміни, здійснені в представленні, відображаються в this backing list.
  - The view can cover the entire list or, as subList()'s name implies, only part of the list.
- Метод subList() корисний для компактного виконання *range-view* operations над списком.
  - Наприклад, list.subList(fromIndex, toIndex).clear(); видаляє range of elements зі списку, починаючи з елементу в позиції fromIndex до останнього елементу (toIndex – 1).
- **Обережно!** Значення представлення стає невизначеним, коли зміни вносяться в backing list. Тому використовувати subList() краще лише тимчасово, коли потрібно виконати sequence of range operations on the backing list.

ArrayList

- Клас ArrayList забезпечує реалізацію списку на основі внутрішнього масиву.
  - У результаті доступ до елементів списку швидкий, проте через велику кількість переміщень при вставці чи видаленні елементу, значно сповільнюється.
- Конструктори ArrayList:
  - ArrayList() створює порожній списковий масив з початковою місткістю (*capacity) на* 10 елементів.
    - При повному заповненні виділяється більший масив, у який поточні елементи копіюються.
    - Новий масив заміняє старий.
    - Процес повторюється при додаванні елементів у списковий масив.
  - ArrayList(Collection<? extends E> c) створює списковий масив, що містить елементи колекції c в тому порядку, в якому вони повертаються ітератором цієї колекції.
    - Викидає NullPointerException, коли c містить null reference.
  - ArrayList(int initialCapacity) створює порожній списковий масив на initialCapacity елементів.
    - Викидає IllegalArgumentException, якщо значення initialCapacity від’ємне.

Демонстрація

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo
{
   public static void main(String[] args)
   {
      List<String> ls = new ArrayList<String>();
      String[] weekDays = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
      for (String weekDay: weekDays)
         ls.add(weekDay);
      dump("ls:", ls);
      ls.set(ls.indexOf("Wed"), "Wednesday");
      dump("ls:", ls);
      ls.remove(ls.lastIndexOf("Fri"));
      dump("ls:", ls);
   }

   static void dump(String title, List<String> ls)
   {
      System.out.print(title + " ");
      for (String s: ls)
         System.out.print(s + " ");
      System.out.println();
   }
}
```

- ArrayListDemo створює списковий масив та масив скорочених назв днів.
  - Далі список заповнюється іменами, передається в стандартний вивід.
  - Змінюємо один елемент списку, знову виводимо список, видаляємо елемент і востаннє виводимо список.
  - Розширений цикл for у методі dump() використовує iterator(), hasNext() та next() за кулісами

```
ls: Sun Mon Tue Wed Thu Fri Sat
ls: Sun Mon Tue Wednesday Thu Fri Sat
ls: Sun Mon Tue Wednesday Thu Sat
```

LinkedList

- Клас LinkedList забезпечує реалізацію списку, яка базується на зв’язних вузлах (linked nodes).
  - Оскільки потрібно виконувати обхід по зв’язках, доступ до елементів списку повільний.
  - Проте для вставки/видалення потрібно лише змінити node references – це швидкі операції.
- Вузол (*node*) – фіксована послідовність значення та зв’язаних локацій у пам’яті.
  - На відміну від масивів, де кожна комірка містить єдине значення примітивного типу або reference supertype, вузол може зберігати багато значень різних типів.
  - Також він може зберігати зв’язки (*links* – посилання на інші вузли).
  - Клас Node описує простий вузол, який складається з одного поля name та одного посилання next.

```
class Node
{
   String name; // value field
   Node next;   // link field
}
```

Зв’язний список

```
Node first = new Node();
first.name = "First node"; // You
Node last = new Node();
last.name = "Last node";
last.next = null;
first.next = last;
Node temp = first;
while (temp != null)
{
   System.out.println(temp.name);
   temp = temp.next;
}
```

- Зауважте, що next має той же тип, що і клас, в якому оголошено цю змінну.
  - Це дозволяє екземпляру вузла зберігати посилання на інший екземпляр вузла (наступний вузол).
  - Отримані вузли зв’язані між собою.
- Код зліва створює 2 об’єкти типу Node та зв’язує перший з другим.
  - Також демонструється обхід зв’язного списку слідуючи за полем next кожного об’єкту Node.
  - Обхід закінчується, коли next міститиме null reference, який сигналізує про кінець списку

First node
Last node

```java
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class LinkedListDemo
{
   public static void main(String[] args)
   {
      List<String> ls = new LinkedList<String>();
      String[] weekDays = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
      for (String weekDay: weekDays)
         ls.add(weekDay);
      dump("ls:", ls);
      ls.add(1, "Sunday");
      ls.add(3, "Monday");
      ls.add(5, "Tuesday");
      ls.add(7, "Wednesday");
      ls.add(9, "Thursday");
      ls.add(11, "Friday");
      ls.add(13, "Saturday");
      dump("ls:", ls);
      ListIterator<String> li = ls.listIterator(ls.size());
      while (li.hasPrevious())
         System.out.print(li.previous() + " ");
      System.out.println();
   }

   static void dump(String title, List<String> ls)
   {
      System.out.print(title + " ");
      for (String s: ls)
         System.out.print(s + " ");
      System.out.println();
   }
}
```

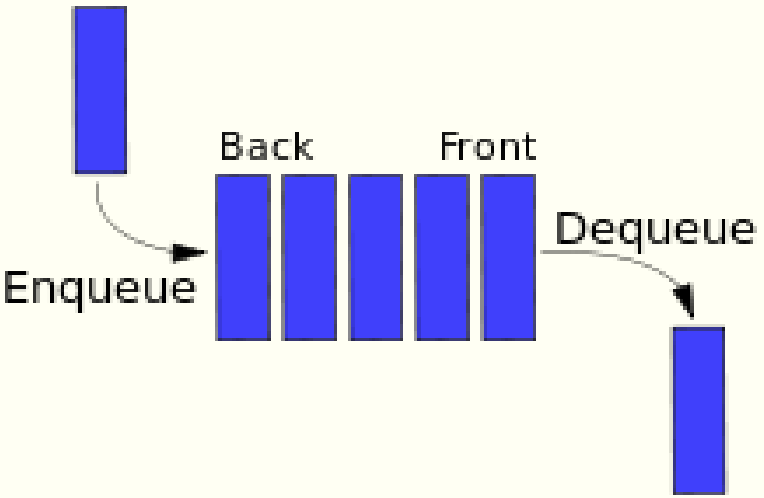
- Конструктори LinkedList:
  - LinkedList() створює порожній зв’язний список.
  - LinkedList(Collection<? extends E> c) створює зв’язний список, що містить елементи с в порядку їх повернення ітератором.
    - Викидає NullPointerException, коли с містить null reference.

```
ls: Sun Mon Tue Wed Thu Fri Sat
ls: Sun Sunday Mon Monday Tue Tuesday Wed Wednesday Thu Thursday Fri Friday Sat Saturday
Saturday Sat Friday Fri Thursday Thu Wednesday Wed Tuesday Tue Monday Mon Sunday Sun
```

ЧЕРГИ (QUEUES)

Черги

- Черга – це колекція, чиї елементи зберігаються та отримуються у заданому порядку.
- Загальна класифікація черг:
  - *First-In, First-Out (FIFO) queue*: елементи додаються в хвіст черги, а видаляються – з голови.
  - *Last-In, First-Out (LIFO) queue*: елементи додаються та видаляються з одного кінця черги. Поводиться як стек.
  - Черга з пріоритетами: елементи додаються відповідно до їх природного порядку або згідно з компаратором, що постачається з реалізацією черги.
- Черга Queue (узагальнений тип Queue<E>) розширює інтерфейс Collection, переоголошуючи метод add() для підгонки контракту
  - Інші методи наслідуються з Collection.



Method	Description
`boolean add(E e)`	Inserts element `e` into this queue if it is possible to do so immediately without violating capacity restrictions. Returns true on success; otherwise, throws `IllegalStateException` when the element cannot be added at this time because no space is currently available. This method also throws `ClassCastException` when `e`'s class prevents `e` from being added to this queue, `NullPointerException` when `e` contains the null reference and this queue doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this queue.
`E element()`	Returns but doesn't also remove the element at the head of this queue. This method throws `NoSuchElementException` when this queue is empty.
`boolean offer(E e)`	Inserts element `e` into this queue if it is possible to do so immediately without violating capacity restrictions. Returns true on success; otherwise, returns false when the element cannot be added at this time because no space is currently available. This method throws `ClassCastException` when `e`'s class prevents `e` from being added to this queue, `NullPointerException` when `e` contains the null reference and this queue doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this queue.
`E peek()`	Returns but doesn't also remove the element at the head of this queue. This method returns null when this queue is empty.
`E poll()`	Returns and also removes the element at the head of this queue. This method returns null when this queue is empty.
`E remove()`	Returns and also removes the element at the head of this queue. This method throws `NoSuchElementException` when this queue is empty. This is the only difference between `remove()` and `poll()`.

Коментарі до таблиці

- Таблиця вказує на два набори методів:
  - Одні методи (наприклад, add()) викидають виключення, коли operation fails;
  - Інші методи (наприклад, offer()) повертають special value (false or null) in the presence of failure.
  - Ці методи корисні в контексті реалізацій черг з обмеженою місткістю, коли failure is a normal occurrence.
- **Зауважте!** Метод offer() краще використовувати замість add() для черг з обмеженою місткістю (capacity-restricted), оскільки offer() не викидає IllegalStateException.
- Java постачає багато класів реалізації Queue, більшість яких знаходяться в пакеті java.util.concurrent: зокрема, LinkedBlockingQueue та SynchronousQueue.
  - А пакет java.util забезпечує LinkedList та PriorityQueue як класи реалізації Queue.
- **Обережно!** Багато імплементацій Queue не дозволяють додавати null-елементи.
  - Краще уникати додавання null-елементу, оскільки він використовується as a special return value методів peek() та poll() для індикації порожності черги.

Клас PriorityQueue

- Клас PriorityQueue постачає реалізацію черги з пріоритетами, яка є чергою, що впорядковує елементи відповідно до природного порядку або згідно з компаратором, який забезпечується при інстанціюванні черги.
  - Черги з пріоритетами забороняють null-елементи та не дозволяють вставку не-Comparable об’єктів при природному впорядкуванні.
- Елемент у голові черги з пріоритетами є найменшим елементом відповідно до заданого впорядкування.
  - Коли кілька елементів є найменшими, довільним чином вибирається один з них.
  - Аналогічно, елемент у хвості (tail) черги з пріоритетами є найбільшим елементом.
- Черги з пріоритетами необмежені, проте мають місткість, яку визначає внутрішній масив, в якому зберігаються елементи черги.
  - Місткість (capacity) принаймні така ж, як і довжина черги, а також зростає автоматично при додаванні елементів у чергу з пріоритетами.

Конструктори PriorityQueue

- **PriorityQueue()** створює екземпляр PriorityQueue на 11 елементів, що впорядковані відповідно до природного порядку.
- **PriorityQueue(Collection<? extends E> c)** створює екземпляр PriorityQueue, який містить елементи колекції c.
  - Якщо c - екземпляр SortedSet або PriorityQueue, черга збереже впорядкування колекції;
  - Інакше ця черга з пріоритетами буде впорядкована згідно з природнім порядком елементів.
  - Конструктор викидає ClassCastException, коли елементи c неможливо порівняти один з одним відповідно до впорядкування черги з пріоритетами, а також NullPointerException, коли c або будь-який її елемент містить null reference.
- **PriorityQueue(int initialCapacity)** створює екземпляр PriorityQueue із заданою місткістю initialCapacity та природним впорядкуванням.
  - Конструктор викидає IllegalArgumentException, якщо initialCapacity < 1.

Інші конструктори

- PriorityQueue(int initialCapacity, Comparator<? super E> comparator) створює екземпляр PriorityQueue із заданим initialCapacity, який впорядковує свої елементи відповідно до вказаного компаратора.
  - Природний порядок використовується, коли компаратор містить null-посилання.
  - Конструктор викидає IllegalArgumentException, коли when initialCapacity < 1.
- PriorityQueue(PriorityQueue<? extends E> pq) створює екземпляр PriorityQueue, що містить елементи pq.
  - Ця черга з пріоритетами буде впорядкована так же, як і pq.
  - Конструктор викидає ClassCastException, коли елементи pq неможливо порівняти один з одним, NullPointerException, коли pq чи якийсь її елемент містить null-посилання.
- PriorityQueue(SortedSet<? extends E> ss) створює екземпляр PriorityQueue, що містить елементи ss.
  - Черга з пріоритетами буде впорядкована так же, як і ss.
  - Конструктор викидає ClassCastException, коли елементи ss неможливо порівняти один з одним according to ss's ordering, NullPointerException, коли ss чи якийсь її елемент містить null-посилання

Додавання випадкових цілих чисел до черги з пріоритетами

```
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo
{
   public static void main(String[] args)
   {
      Queue<Integer> qi = new PriorityQueue<Integer>();
      for (int i = 0; i < 15; i++)
         qi.add((int) (Math.random() * 100));
      while (!qi.isEmpty())
         System.out.print(qi.poll() + " ");
      System.out.println();
   }
}
```

```
30 43 53 61 61 66 66 67 76 78 80 83 87 90 97
```

- Після створення черги з пріоритетами головний потік додає в неї 15 випадкових цілих чисел (від 0 до 99).
  - Потім проходимо по priority queue, отримуючи наступний елемент та виводячи його.
  - Оскільки poll() повертає null, коли більше немає елементів, цикл можна записати так:

```
Integer i;
while ((i = qi.poll()) != null)
   System.out.print(i + " ");
```

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo
{
   final static int NELEM = 15;

   public static void main(String[] args)
   {
      Comparator<Integer> cmp;
      cmp = new Comparator<Integer>()
            {
               @Override
               public int compare(Integer e1, Integer e2)
               {
                  return e2 - e1;
               }
            };
      Queue<Integer> qi = new PriorityQueue<Integer>(NELEM, cmp);
      for (int i = 0; i < NELEM; i++)
         qi.add((int) (Math.random() * 100));
      while (!qi.isEmpty())
         System.out.print(qi.poll() + " ");
      System.out.println();
   }
}
```

Демонстрація

- Припустимо, потрібно змінити порядок виводу з попереднього прикладу на зворотній.
  - Зліва найбільший елемент, а справа- найменший.
- Спочатку оголосимо константу NELEM, щоб в одному місці задавати початкову місткість черги та кількість елементів, доданих в неї.
  - Потім інстанціюємо анонімний клас, який реалізує Comparator.
  - Його метод compareTo() віднімає елемент e1 від e2, щоб досягнути спадаючого впорядкування.
  - Компілятор виконує розпаковку e2 та e1, конвертуючи e2 - e1 в e2.intValue() - e1.intValue().
  - У кінці в конструктор PriorityQueue(int initialCapacity, Comparator<? super E> comparator) передається початкова місткість (NELEM елементів) та інстанційований компаратор.

```
97 72 70 70 67 64 56 43 36 22 9 5 3 2 1
```

Дек та методи роботи з ним

- A *deque* – двоголова черга, в якій вставка або видалення елементу відбувається в її голові/хвості. Деки можна використовувати як черги або стеки.
  - Дек (узагальнений тип Deque<E>) розширює Queue, унаслідуваний з якого метод add(E e) виконує вставку e у хвіст деку.

Метод	Опис
void addFirst(E e)	Виконує вставку елементу e в голову деку, за можливості, негайно без capacity restrictions. При використанні capacity-restricted деку краще використовувати метод offerFirst(). Виключення: IllegalStateException: e неможливо додати через capacity restrictions ClassCastException: клас e робить додавання в дек неможливим NullPointerException: e містить null-посилання, а дек їх не підтримує IllegalArgumentException: певна властивість e робить його додавання в дек неможливим
void addLast(E e)	Виконує вставку елементу e у хвіст деку, за можливості, негайно без capacity restrictions. При використанні capacity-restricted деку краще використовувати метод offerFirst(). Виключення аналогічні до addFirst()

`Iterator<E> descendingIterator()`	Returns an iterator over the elements in this deque in reverse sequential order. The elements will be returned in order from last (tail) to first (head). The inherited `Iterator<E> iterator()` method returns elements from the head to the tail.
`E element()`	Retrieves but doesn't remove the first element of this deque (at the head). This method differs from `peek()` only in that it throws `NoSuchElementException` when this deque is empty. This method is equivalent to `getFirst()`.
`E getFirst()`	Retrieves but doesn't remove the first element of this deque. This method differs from `peekFirst()` only in that it throws `NoSuchElementException` when this deque is empty.
`E getLast()`	Retrieves but doesn't remove the last element of this deque. This method differs from `peekLast()` only in that it throws `NoSuchElementException` when this deque is empty.
`boolean offer(E e)`	Inserts `e` at the tail of this deque if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false when no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the `add()` method, which can fail to insert an element only by throwing an exception. This method throws `ClassCastException` when `e`'s class prevents `e` from being added to this deque, `NullPointerException` when `e` contains the null reference and this deque doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this deque. This method is equivalent to `offerLast()`.

`boolean offerFirst(E e)`	Inserts `e` at the head of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the `addFirst()` method, which can fail to insert an element only by throwing an exception. This method throws `ClassCastException` when `e`'s class prevents `e` from being added to this deque, `NullPointerException` when `e` contains the null reference and this deque doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this deque.
`boolean offerLast(E e)`	Inserts `e` at the tail of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the `addLast()` method, which can fail to insert an element only by throwing an exception. This method throws `ClassCastException` when `e`'s class prevents `e` from being added to this deque, `NullPointerException` when `e` contains the null reference and this deque doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this deque.
`E peek()`	Retrieves but doesn't remove the first element of this deque (at the head), or returns null when this deque is empty. This method is equivalent to `peekFirst()`.
`E peekFirst()`	Retrieves but doesn't remove the first element of this deque (at the head), or returns null when this deque is empty.

`E peekLast()`	Retrieves but doesn't remove the last element of this deque (at the tail), or returns null when this deque is empty.
`E poll()`	Retrieves and removes the first element of this deque (at the head), or returns null when this deque is empty. This method is equivalent to `pollFirst()`.
`E pollFirst()`	Retrieves and removes the first element of this deque (at the head), or returns null when this deque is empty.
`E pollLast()`	Retrieves and removes the last element of this deque (at the tail), or returns null when this deque is empty.
`E pop()`	Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque. This method is equivalent to `removeFirst()`.
`void push(E e)`	Pushes `e` onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing `IllegalStateException` when no space is currently available. This method also throws `ClassCastException` when `e`'s class prevents `e` from being added to this deque, `NullPointerException` when `e` contains the null reference and this deque doesn't permit null elements to be added, and `IllegalArgumentException` when some property of `e` prevents it from being added to this deque. This method is equivalent to `addFirst()`.

`E remove()`	Retrieves and removes the first element of this deque (at the head). This method differs from `poll()` only in that it throws `NoSuchElementException` when this deque is empty. This method is equivalent to `removeFirst()`.
`E removeFirst()`	Retrieves and removes the first element of this deque. This method differs from `pollFirst()` only in that it throws `NoSuchElementException` when this deque is empty.
`boolean removeFirstOccurrence(Object o)`	Removes the first occurrence of `o` from this deque. If the deque doesn't contain `o`, it is unchanged. Returns true when this deque contained `o` (or equivalently, when this deque changed as a result of the call). This method throws `ClassCastException` when `o`'s class prevents `o` from being added to this deque and `NullPointerException` when `o` contains the null reference and this deque doesn't permit null elements to be added. The inherited `boolean remove(Object o)` method is equivalent to this method.
`E removeLast()`	Retrieves and removes the last element of this deque. This method differs from `pollLast()` only in that it throws `NoSuchElementException` when this deque is empty.
`boolean removeLastOccurrence(Object o)`	Removes the last occurrence of `o` from this deque. If the deque doesn't contain `o`, it is unchanged. Returns true when this deque contained `o` (or equivalently, when this deque changed as a result of the call). This method throws `ClassCastException` when `o`'s class prevents `o` from being added to this deque and `NullPointerException` when `o` contains the null reference and this deque doesn't permit null elements to be added.

Deque оголошує методи для доступу до елементів з обох боків деку

- Ці методи додають, видаляють та переглядають елемент.
  - Кожен з них існує в двох формах: одна викидає виключення, коли operation fails, а інша повертає спеціальне значення (null або false залежно від операції).
-

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Queue Method	Equivalent Deque Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

- Коли використовується дек, спостерігається FIFO-поведінка.
  - Елементи додаються в кінець деку та видаляються з початку.
- Деки можна використовувати в якості LIFO-стеків.
  - Коли дек використовується як стек, елементи pushed і popped на початку деку.
  - Метод стеку push(e) еквівалентний Deque.addFirst(e),
  - метод pop() - методу Deque.removeFirst(),
  - метод peek() – методу Deque.peekFirst().
  - Тому Deque оголошує орієнтовані на роботу зі стеком методи E peek(), E pop() та void push(E e)

ArrayDeque

- Клас ArrayDeque постачає реалізацію інтерфейсу Deque на основі масиву змінної довжини.
  - Забороняє додавати null-елементи в дек, а метод iterator() повертає fail-fast ітератори.
- Конструктори ArrayDeque:
  - ArrayDeque() створює порожній array deque на 16 елементів.
  - ArrayDeque(Collection<? extends E> c) створює array deque, який містить елементи колекції c в тому порядку, як вони повертаються ітератором колекції.
    - Перший елемент, повернений ітератором колекції c, стає першим елементом (front) деку.
    - NullPointerException викидається, коли колекція c містить null reference.
  - ArrayDeque(int numElements) створює порожній array deque з початковою місткістю, достатньою для зберігання numElements елементів.
    - Якщо numElements ≥ 0, виключення не викидаються.

Демонстрація ArrayDeque

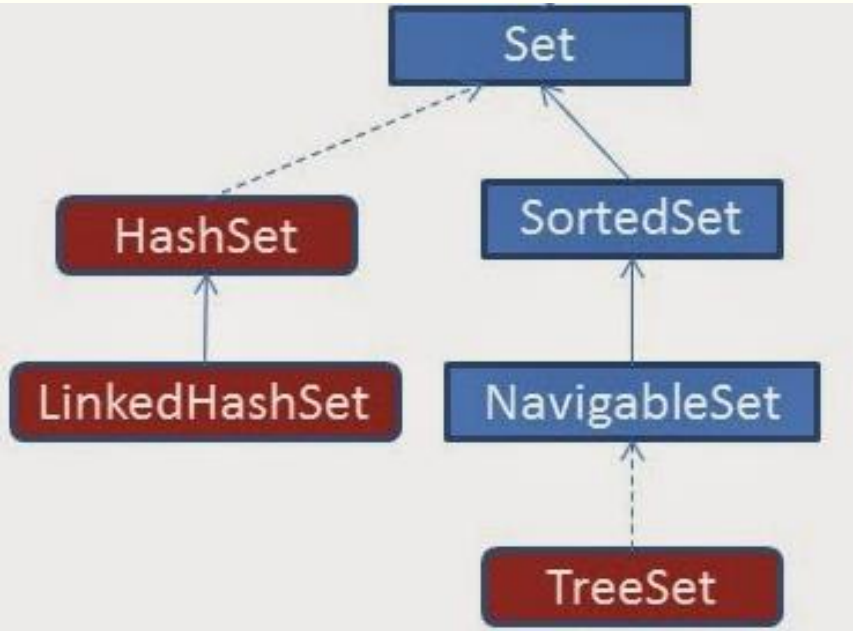
```
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDequeDemo
{
   public static void main(String[] args)
   {
      Deque<String> stack = new ArrayDeque<String>();
      String[] weekdays = { "Sunday", "Monday", "Tuesday", "Wednesday",
                            "Thursday", "Friday", "Saturday" };
      for (String weekday: weekdays)
         stack.push(weekday);
      while (stack.peek() != null)
         System.out.println(stack.pop());
   }
}
```

```
Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
Sunday
```

- Створює дек для його використання в якості стеку та масиву назв днів.
  - Потім ці назви додаються в стек і вибираються з нього, показуючи зворотний порядок.

Інтерфейс Set (множини, сети)



- Множина – колекція, що не містить повторів.
  - Іншими словами, в сетах немає таких пар елементів *e1 та e2, щоб* *e1*.equals(*e2*) повертав true.
  - Крім того, сет може містити максимум один null-елемент.
- Множини описуються за допомогою інтерфейсу Set (узагальнений тип Set<E>).
- Інтерфейс Set розширює інтерфейс Collection та переоголошує успадковані методи для зручності та з метою додавання умов до контракту.
  - Додаються умови для методів add(), equals(), hashCode().
- Документація по інтерфейсу Set стверджує, що всі конструктори класів реалізації мають створювати сети, що не містять повторних елементів.
  - Нових методів інтерфейс Set не представляє.

Клас TreeSet забезпечує реалізацію сету на основі дерева

- Елементи зберігаються відсортованими.
  - Проте доступ до них дещо повільніший, ніж для інших, невідсортованих сетів, оскільки потрібно виконувати обхід по зв'язках.
- Конструктори TreeSet:
  - **TreeSet()** створює новий, порожній tree set, відсортований у природному порядку його елементів. Всі елементи, додані в сет, мають реалізовувати інтерфейс Comparable.
  - **TreeSet(Collection<? extends E> c)** створює новий tree set, що містить елементи колекції c, відсортовані в природному порядку. Всі додані в новий сет елементи повинні реалізовувати інтерфейс Comparable.
    - Конструктор викидає ClassCastException, коли елементи колекції c не реалізують Comparable або не є взаємно порівнюваними
    - NullPointerException, коли c містить null reference.
  - **TreeSet(Comparator<? super E> comparator)** створює новий, порожній tree set, відсортований в порядку, заданому компаратором. Передача null компаратору передбачає використання природного порядку.
  - **TreeSet(SortedSet<E> ss)** створює новий tree set, що містить ті ж елементи і в тому ж порядку, що і ss.
    - Конструктор викидає NullPointerException, коли ss містить null reference.

Демонстрація Tree Set з рядковими елементами, відсортованими в природному порядку

```
import java.util.Set;
import java.util.TreeSet;

public class TreeSetDemo
{
   public static void main(String[] args)
   {
      Set<String> ss = new TreeSet<String>();
      String[] fruits = {"apples", "pears", "grapes", "bananas", "kiwis"};
      for (String fruit: fruits)
         ss.add(fruit);
      dump("ss:", ss);
   }

   static void dump(String title, Set<String> ss)
   {
      System.out.print(title + " ");
      for (String s: ss)
         System.out.print(s + " ");
      System.out.println();
   }
}
```

- TreeSetDemo створює tree set та масив назв фруктів.
- Потім сет заповнюється цими назвами та виводиться в консоль.

Оскільки String реалізує Comparable, для цього додатку вміст масиву fruits коректно додається в tree set, створений з конструктора TreeSet().

```
ss: apples bananas grapes kiwis pears
```

HashSet

- Клас HashSet забезпечує реалізацію, що підтримується хешованими структурами даних
  - Реалізується як екземпляр HashMap, який постачає швидкий спосіб визначення, чи елемент уже присутній у цій структурі.
  - Хоч клас не гарантує впорядкованості елементів, HashSet набагато швидший за TreeSet.
  - Крім того, HashSet дозволяє зберігати null-посилання в своїх екземплярах.
- Конструктори HashSet:
  - **HashSet()** створює новий, порожній hashset, де підтримуваний екземпляр HashMap має місткість (initial capacity) 16 та коефіцієнт завантаження 0.75.
  - **HashSet(Collection<? extends E> c)** створює новий hashset з елементами колекції c, де підтримуваний екземпляр HashMap має відповідну колекції місткість та коефіцієнт завантаження 0.75. Конструктор викидає NullPointerException, коли c містить null-посилання.
  - **HashSet(int initialCapacity)** створює новий, порожній hashset, де підтримуваний екземпляр HashMap має задану місткість (initialCapacity) та коефіцієнт заповнення 0.75.
    - Конструктор викидає IllegalArgumentException, коли значення initialCapacity менше за 0.
  - **HashSet(int initialCapacity, float loadFactor)** створює новий, порожній hashset, де підтримуваний екземпляр HashMap має задану місткість (initialCapacity) та коефіцієнт завантаження (loadFactor).
    - Конструктор викидає IllegalArgumentException, коли initialCapacity < 0 або loadFactor <=0.

Демонстрація Hashset з невпорядкованими рядковими елементами

```
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo
{
   public static void main(String[] args)
   {
      Set<String> ss = new HashSet<String>();
      String[] fruits = {"apples", "pears", "grapes", "bananas", "kiwis",
                         "pears", null};
      for (String fruit: fruits)
         ss.add(fruit);
      dump("ss:", ss);
   }

   static void dump(String title, Set<String> ss)
   {
      System.out.print(title + " ");
      for (String s: ss)
         System.out.print(s + " ");
      System.out.println();
   }
}
```

- HashSetDemo створює hashset та масив назв фруктів.
- Потім сет заповнюється цими назвами та виводиться в консоль.

На відміну від TreeSet, HashSet дозволяє додавати null (NullPointerException не викидається), тому вивід включає null.

```
ss: null grapes bananas kiwis pears apples
```

Власний клас Planet, який не переозначає метод hashCode()

```
import java.util.HashSet;
import java.util.Set;

public class CustomClassAndHashSet
{
   public static void main(String[] args)
   {
      Set<Planet> sp = new HashSet<Planet>();
      sp.add(new Planet("Mercury"));
      sp.add(new Planet("Venus"));
      sp.add(new Planet("Earth"));
      sp.add(new Planet("Mars"));
      sp.add(new Planet("Jupiter"));
      sp.add(new Planet("Saturn"));
      sp.add(new Planet("Uranus"));
      sp.add(new Planet("Neptune"));
      sp.add(new Planet("Fomalhaut b"));
      Planet p1 = new Planet("51 Pegasi b");
      sp.add(p1);
      Planet p2 = new Planet("51 Pegasi b");
      sp.add(p2);
      System.out.println(p1.equals(p2));
      System.out.println(sp);
   }
}
```

```
class Planet
{
   private String name;

   Planet(String name)
   {
      this.name = name;
   }

   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Planet))
         return false;
      Planet p = (Planet) o;
      return p.name.equals(name);
   }

   String getName()
   {
      return name;
   }

   @Override
   public String toString()
   {
      return name;
   }
}
```

Припустимо, Вам потрібно додати екземпляри класів у hashset.

- Як і для String, Ваші класи мають переозначувати equals() та hashCode();
- інакше повторні екземпляри класів зможуть зберігатись у hashset.

Клас Planet оголошує одне поле name типу String.

- Створювати окремо клас Planet для роботи з простими рядковими даними здається зайвим, але можна додати нові поля в Planet в майбутньому.

Результат запуску:

```
true
Neptune, Mars, Mercury, Fomalhaut b, Venus, 51 Pegasi b, 51 Pegasi b, Jupiter, Saturn, Earth, Uranus]
```

```
class Planet
{
   private String name;

   Planet(String name)
   {
      this.name = name;
   }

   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Planet))
         return false;
      Planet p = (Planet) o;
      return p.name.equals(name);
   }

   String getName()
   {
      return name;
   }

   @Override
   public int hashCode()
   {
      return name.hashCode();
   }

   @Override
   public String toString()
   {
      return name;
   }
}
```

Власний клас Planet, який не переозначає метод hashCode()

- Вивід:
  ```
  true
  [Saturn, Earth, Uranus, Fomalhaut b, 51
  Pegasi b, Venus, Jupiter, Mercury, Mars,
  Neptune]
  ```
- **Зауважте!** Клас LinkedHashSet є підкласом HashSet, який використовує зв'язний список для зберігання елементів.
  - У результаті ітератор класу LinkedHashSet повертає елементи в порядку їх вставки.
  - Якщо в демонстрації HashSet замінити рядок на

    `Set<String> ss = new LinkedHashSet<String>();` ,

    вивід додатку буде ss: apples pears grapes bananas kiwis null.
  - LinkedHashSet працює повільніше за HashSet, проте швидше за TreeSet.

Поняття перелічення (Enum)

	Data Structure	Sorting	Iterator	Nulls?
HashSet	Hash table	No	Fail-fast	Yes
Linked HashSet	Hash table + linked list	Insertion Order	Fail-fast	Yes
EnumSet	Bit vector	Natural Order	Weakly consistent	No
TreeSet	Red-black tree	Sorted	Fail-fast	Depends
CopyOnWrite ArraySet	Array	No	Snapshot	Yes
Concurrent SkipListSet	Skip list	Sorted	Weakly consistent	No

- Перелічення (enumerated type) – це тип, який задає іменовану послідовність пов’язаних констант в якості їх значень.
  - Місяці в календарі, монети у валюті, дні в тижні – приклади перерахувань.
- Java-розробники традиційно використовують набори іменованих цілочисельних констант для представлення перерахувань.
  - Оскільки така форма представлення показує свою проблематичність, у Java 5 представили альтернативу.

Проблема з традиційними перелічуваними типами

```
class Coin
{
   final static int PENNY = 0;
   final static int NICKEL = 1;
   final static int DIME = 2;
   final static int QUARTER = 3;
}
```

```
class Weekday
{
   final static int SUNDAY = 0;
   final static int MONDAY = 1;
   final static int TUESDAY = 2;
   final static int WEDNESDAY = 3;
   final static int THURSDAY = 4;
   final static int FRIDAY = 5;
   final static int SATURDAY = 6;
}
```

- Найбільша проблема перелічень (код зліва) - нестача compile-time безпеки типів.
  - Наприклад, можна передати coin в метод, який потребує weekday, а компілятор не зверне на це уваги.
  - Можна також порівнювати монети та дні тижня: Coin.NICKEL == Weekday.MONDAY, навіть задавати беззмістовні вирази, на зразок Coin.DIME + Weekday.FRIDAY - 1/Coin.QUARTER.
  - Компілятор не помітить цього, оскільки він бачить лише цілі числа.
- Додатки, що залежать від перелічень, крихкі.
  - Оскільки константи типу компілюються в class-файли, зміна constant's int value вимагає перекомпілювання залежних застосунків та підвищує ризики помилкової поведінки.
- Інша проблема перелічень – цілочисельні константи неможливо перетворити у змістовні рядкові описи.
  - Наприклад, що означає цифра 4 при налагоджуванні збійного додатку?
  - Було б добре бачити THURSDAY замість 4.

Альтернатива переліченням

- Java 5 представила enums у якості кращої альтернативи переліченням.

```
enum Coin { PENNY, NICKEL, DIME, QUARTER }
enum Weekday { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

- Незважаючи на схожість синтаксису з int-based перерахуваннями в C++ та інших мовах, enum є класами.
  - Кожна константа є public static final полем, яке представляє екземпляр її класу перелічення (enum).
  - Оскільки конструктор для нових констант викликати неможливо, а константи незмінні, можна безпечно використовувати оператор == для їх ефективного порівняння (на відміну від рядків).
  - Наприклад, c == Coin.NICKEL.
- Enums забезпечують compile-time безпеку типів, забороняючи порівнювати константи різних типів.
  - Компілятор повідомить про помилку при Coin.PENNY == Weekday.SUNDAY.
  - А також при передачі константи з невідповідного enum у метод.
- Додатки, що залежать від enums не крихкі, оскільки константи з enum не компілюються в class-файл.
  - Також enum постачає метод toString() для повернення більш корисного опису значення константи.
  - Враховуючи корисність enum, у Java 5 удосконалено оператор switch для їх підтримки.

Демонстрація роботи оператора switch з enum

```
public class EnhancedSwitch
{
   enum Coin { PENNY, NICKEL, DIME, QUARTER }

   public static void main(String[] args)
   {
      Coin coin = Coin.NICKEL;
      switch (coin)
      {
         case PENNY  : System.out.println("1 cent"); break;
         case NICKEL : System.out.println("5 cents"); break;
         case DIME   : System.out.println("10 cents"); break;
         case QUARTER: System.out.println("25 cents"); break;
         default     : assert false;
      }
   }
}
```

- Удосконалений оператор switch дозволяє задавати назву константи в якості мітки для case.
  - Якщо дописувати префікс, наприклад, Coin.DIME, компілятор повідомить про помилку.

Удосконалення Enum – можете додавати поля, конструктори та методи в enum, можна навіть, щоб enum реалізовував інтерфейси

```
enum Coin
{
   PENNY(1),
   NICKEL(5),
   DIME(10),
   QUARTER(25);

   private final int denomValue;

   Coin(int denomValue)
   {
      this.denomValue = denomValue;
   }

   int denomValue()
   {
      return denomValue;
   }

   int toDenomination(int numPennies)
   {
      return numPennies / denomValue;
   }
}
```

-

```
public class Coins
{
   public static void main(String[] args)
   {
      if (args.length == 1)
      {
         int numPennies = Integer.parseInt(args[0]);
         System.out.println(numPennies + " pennies is equivalent to:");
         int numQuarters = Coin.QUARTER.toDenomination(numPennies);
         System.out.println(numQuarters + " " + Coin.QUARTER.toString() +
                            (numQuarters != 1 ? "s," : ","));
         numPennies -= numQuarters * Coin.QUARTER.denomValue();
         int numDimes = Coin.DIME.toDenomination(numPennies);
         System.out.println(numDimes + " " + Coin.DIME.toString() +
                            (numDimes != 1 ? "s, " : ","));
         numPennies -= numDimes * Coin.DIME.denomValue();
         int numNickels = Coin.NICKEL.toDenomination(numPennies);
         System.out.println(numNickels + " " + Coin.NICKEL.toString() +
                            (numNickels != 1 ? "s, " : ", and"));
         numPennies -= numNickels*Coin.NICKEL.denomValue();
         System.out.println(numPennies + " " + Coin.PENNY.toString() +
                            (numPennies != 1 ? "s" : ""));
      }
      System.out.println();
      System.out.println("Denomination values:");
      for (int i = 0; i < Coin.values().length; i++)
         System.out.println(Coin.values()[i].denomValue());
   }
}
```

Вивід програми

```
119 pennies is equivalent to:
4 QUARTERs,
1 DIME,
1 NICKEL, and
4 PENNYs

Denomination values:
1
5
10
25
```

- Програма конвертує введену кількість пенні в кількість потрібних для видачі монет та їх номіналів
  - Також додаток передбачає метод toString() для виводу монетного представлення.
- Ще викликався метод values(), який повертає масив усіх констант Coin, оголошених в enum
  - Тип, що повертає value(), у даному випадку, - Coin[] .
  - Масив корисний, коли потрібно ітерувати по константах.
- Вивід показує, що toString() повертає назву константи.
  - Інколи корисно переозначити цей метод повертав більш змістовний результат.
  - Наприклад, метод виділяє токени (іменовані послідовності символів) з рядка, використовуючи Token enum для списку назв токенів та за допомогою переозначення методу toString().

Переозначення toString() для повернення значення Token-константи

```
public enum Token
{
   IDENTIFIER("ID"),
   INTEGER("INT"),
   LPAREN("("),
   RPAREN(")"),
   COMMA(",");

   private final String tokValue;

   Token(String tokValue)
   {
      this.tokValue = tokValue;
   }

   @Override
   public String toString()
   {
      return tokValue;
   }

   public static void main(String[] args)
   {
      System.out.println("Token values:");
      for (int i = 0; i < Token.values().length; i++)
         System.out.println(Token.values()[i].name() + " = " +
                            Token.values()[i]);
   }
}
```

- Метод main() викликає values() для повернення масиву Token-констант.
- Для кожної константи він
  - викликає метод name() константи для того, щоб повернути назву константи,
  - неявно викликає toString(), щоб повернути constant's value

```
Token values:
IDENTIFIER = ID
INTEGER = INT
LPAREN = (
RPAREN = )
COMMA = ,
```

```
public enum TempConversion
{
   C2F("Celsius to Fahrenheit")
   {
      @Override
      double convert(double value)
      {
         return value * 9.0 / 5.0 + 32.0;
      }
   },

   F2C("Fahrenheit to Celsius")
   {
      @Override
      double convert(double value)
      {
         return (value - 32.0) * 5.0 / 9.0;
      }
   };

   TempConversion(String desc)
   {
      this.desc = desc;
   }

   private String desc;

   @Override
   public String toString()
   {
      return desc;
   }
```

Використання анонімних підкласів для зміни поведінки Enum Constants

Інший спосіб покращити enum – присвоїти різну поведінку кожній константі.

- Цю задачу можна виконати, представивши абстрактний метод в enum та переозначивши його в анонімному підкласі константи.

```
Celsius to Fahrenheit for 100.0 degrees = 212.0
Fahrenheit to Celsius for 98.6 degrees = 37.0
```

```
   abstract double convert(double value);

   public static void main(String[] args)
   {
      System.out.println(C2F + " for 100.0 degrees = " + C2F.convert(100.0));
      System.out.println(F2C + " for 98.6 degrees = " + F2C.convert(98.6));
   }
}
```

Клас Enum

- Компілятор вважає enum синтаксичним цукром.
  - Коли зустрічає оголошення на зразок enum Coin {}, він генерує клас (Coin), який субкласує абстрактний клас Enum (з пакету java.lang).
- Документація класу Enum говорить, що в ньому переозначаються методи clone(), equals(), finalize(), hashCode(), toString() класу Object.
  - clone() переозначається, щоб запобігти клонуванню констант, щоб їх можна було порівнювати за допомогою оператору ==.
  - equals() переозначається, щоб порівнювати константи за посиланнями.
  - finalize() переозначається, щоб забезпечити неможливість фіналізації констант.
  - hashCode() – переозначається в парі з equals().
  - toString() переозначається, щоб повертати назву константи.
- Крім toString(), всі переозначені методи оголошені як final, тому не можуть переозначуватись в підкласі.

Методи класу Enum

- compareTo() порівнює поточну константу з переданою в якості аргументу методу константою, щоб переглянути, яка константа передує іншій в enum і повертає значення, що вказує на їх порядок.
  - Цей метод дає можливість відсортувати масив констант.
- getDeclaringClass() повертає об’єкт типу Class, що відповідає enum поточної константи.
  - Наприклад, об’єкт типу Coin повертається після виклику Coin.PENNY.getDeclaringClass() для enum Coin { PENNY, NICKEL, DIME, QUARTER}.
  - Також повертає TempConversion при виклику TempConversion.C2F.getDeclaringClass() для TempConversion enum.
  - Метод compareTo() використовує методи Class.getClass() та Enum.getDeclaringClass() порівняння лише констант з одного enum.
    - Інакше викидається ClassCastException.
- name() повертає назву константи. Якщо toString() не перезначувати, метод теж повертає constant’s name.
- ordinal() повертає zero-based *порядок* – ціле число, яке визначає позицію константи в enum.
  - compareTo() порівнює порядки (ordinals).

Клас Enum

- Enum також постачає метод
  - public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
- для повернення константи enum із заданого enum із заданою назвою.
  - enumType ідентифікує об’єкт enum класу Class, з якого повертається константа.
  - name вказує назву константи для повернення. Наприклад,
    Coin penny = Enum.valueOf(Coin.class, "PENNY"); присвоює константу типу Coin з назвою PENNY об’єкту penny.
  - У Java-документації класу Enum Ви не знайдете метод values(), оскільки компілятор *synthesizes* (manufactures) цей метод під час генерування класу.

Розширення класу Enum

```
public final class Coin extends Enum<Coin>
{
   public static final Coin PENNY = new Coin("PENNY", 0);
   public static final Coin NICKEL = new Coin("NICKEL", 1);
   public static final Coin DIME = new Coin("DIME", 2);
   public static final Coin QUARTER = new Coin("QUARTER", 3);
   private static final Coin[] $VALUES = { PENNY, NICKEL, DIME, QUARTER };

   public static Coin[] values()
   {
      return Coin.$VALUES.clone();
   }

   public static Coin valueOf(String name)
   {
      return Enum.valueOf(Coin.class, "Coin");
   }

   private Coin(String name, int ordinal)
   {
      super(name, ordinal);
   }
}
```

- Узагальнений тип Enum має вигляд Enum<E extends Enum<E>>.

Приклад традиційного перелічення

```
static final int SUNDAY = 1;
static final int MONDAY = 2;
static final int TUESDAY = 4;
static final int WEDNESDAY = 8;
static final int THURSDAY = 16;
static final int FRIDAY = 32;
static final int SATURDAY = 64;
```

- Код виглядатиме краще при комбінуванні констант, на зразок static final int DAYS_OFF = SUNDAY | MONDAY;
  - DAYS_OFF – приклад цілочисельної бітової множини (*bitset*) фіксованого розміру – набору бітів, у якому кожен біт дорівнює 1, якщо пов’язаний з ним елемент відноситься до сету, і 0, якщо елемент у сеті відсутній.
  - Цілочисельний бітсет не може містити більше 32 членів, оскільки int містить всього 32 біти.
  - Аналогічно, бітсет чисел типу long не може містити більше 64 members.

У зв’язці з колекціями

```
import java.util.Set;
import java.util.TreeSet;

enum Weekday
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class DaysOff
{
   public static void main(String[] args)
   {
      Set<Weekday> daysOff = new TreeSet<Weekday>();
      daysOff.add(Weekday.SUNDAY);
      daysOff.add(Weekday.MONDAY);
      System.out.println(daysOff);
   }
}
```

```
[SUNDAY, MONDAY]
```

Використання сетів (інтерфейс Set) для представлення bitset більш затратне по пам’яті та часу. Тому було представлено EnumSet.

- Клас EnumSet забезпечує реалізацію Set, що базується на бітсетах.
  - Його елементи є константами, що мають надходити з того enum, який задано при створенні enum set.
  - Null-елементи заборонені; інакше викидає NullPointerException.

Створення EnumSet, еквівалентного DAYS_OFF

```
import java.util.EnumSet;
import java.util.Iterator;
import java.util.Set;

enum Weekday
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumSetDemo
{
   public static void main(String[] args)
   {
      Set<Weekday> daysOff = EnumSet.of(Weekday.SUNDAY, Weekday.MONDAY);
      Iterator<Weekday> iter = daysOff.iterator();
      while (iter.hasNext())
         System.out.println(iter.next());
   }
}
```

```
SUNDAY
MONDAY
```

- EnumSetDemo використовує переваги того, що EnumSet (узагальнений тип EnumSet<E extends Enum<E>>) постачає різні методи класу для конструювання доречних перелічуваних множин.
- <E extends Enum<E>> EnumSet<E> **of**(E e1, E e2) повертає екземпляр EnumSet, що складається з елементів e1 та e2.
  - У цьому прикладі - це Weekday.SUNDAY та Weekday.MONDAY
- Також EnumSet постачає інші методи для створення enum sets.
  - Наприклад, метод allOf() повертає екземпляр EnumSet, що містить всі константи в enum. Єдиним аргументом є літерал класу (назва_класу.class)

```
Set<Weekday> allWeekDays = EnumSet.allOf(Weekday.class);
```

- Аналогічно, метод range() повертає екземпляр EnumSet, що містить діапазон елементів enum (границі діапазону – параметри методу)

```
for (WeekDay wd: EnumSet.range(WeekDay.MONDAY, WeekDay.FRIDAY))
   System.out.println(wd);
```

Відсортовані множини. TreeSet

- підтримує свої елементи у зростаючому порядку (відповідно до природного впорядкування або компаратора, що постачається при створенні відсортованого сету).
  - Відсортовані множини описуються інтерфейсом SortedSet (узагальнений тип SortedSet<E>), який розширює інтерфейс Set.
- За двома винятками, успадковані від Set методи поводять себе так же, як і для будь-яких сетів:
  - Екземпляр Iterator повертається методом iterator(), який обходить відсортовану мережу в порядку зростання елементів.
  - Масив, що повертається методом toArray(), містить впорядковані елементи відсортованого сету.

Методи SortedSet

Method	Description
`Comparator<? super E> comparator()`	Returns the comparator used to order the elements in this set or null when this set uses the natural ordering of its elements.
`E first()`	Returns the first (lowest) element currently in this set, or throws a `NoSuchElementException` instance when this set is empty.
`SortedSet<E> headSet(E toElement)`	Returns a view of that portion of this set whose elements are strictly less than `toElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `toElement` is not compatible with this set's comparator (or, when the set has no comparator, when `toElement` doesn't implement `Comparable`), `NullPointerException` when `toElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when this set has a restricted range and `toElement` lies outside of this range's bounds.
`E last()`	Returns the last (highest) element currently in this set, or throws a `NoSuchElementException` instance when this set is empty.

Method	Description
`SortedSet<E> subSet(E fromElement, E toElement)`	Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive. (When `fromElement` and `toElement` are equal, the returned set is empty.) The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `fromElement` and `toElement` cannot be compared to one another using this set's comparator (or, when the set has no comparator, using natural ordering), `NullPointerException` when `fromElement` or `toElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when `fromElement` is greater than `toElement` or when this set has a restricted range and `fromElement` or `toElement` lies outside of this range's bounds.
`SortedSet<E> tailSet(E fromElement)`	Returns a view of that portion of this set whose elements are greater than or equal to `fromElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `fromElement` is not compatible with this set's comparator (or, when the set has no comparator, when `fromElement` doesn't implement `Comparable`), `NullPointerException` when `fromElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when this set has a restricted range and `fromElement` lies outside of the range's bounds.

Відсортована множина назв фруктів та овочів

```
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo
{
   public static void main(String[] args)
   {
      SortedSet<String> sss = new TreeSet<String>();
      String[] fruitAndVeg =
      {
         "apple", "potato", "turnip", "banana", "corn", "carrot", "cherry",
         "pear", "mango", "strawberry", "cucumber", "grape", "banana",
         "kiwi", "radish", "blueberry", "tomato", "onion", "raspberry",
         "lemon", "pepper", "squash", "melon", "zucchini", "peach", "plum",
         "turnip", "onion", "nectarine"
      };
      System.out.println("Array size = " + fruitAndVeg.length);
      for (String fruitVeg: fruitAndVeg)
         sss.add(fruitVeg);
      dump("sss:", sss);
      System.out.println("Sorted set size = " + sss.size());
      System.out.println("First element = " + sss.first());
      System.out.println("Last element = " + sss.last());
      System.out.println("Comparator = " + sss.comparator());
      dump("hs:", sss.headSet("n"));
      dump("ts:", sss.tailSet("n"));
      System.out.println("Count of p-named fruits & vegetables = " +
                         sss.subSet("p", "q").size());
      System.out.println("Incorrect count of c-named fruits & vegetables = " +
                         sss.subSet("carrot", "cucumber").size());
      System.out.println("Correct count of c-named fruits & vegetables = " +
                         sss.subSet("carrot", "cucumber\0").size());
   }
```

- SortedSetDemo створює відсортований сет та масив назв фруктів та овочів, з якого буде заповнюватись сет.
  - Далі – вивід інформації про сет, включаючи head і tail.
- Метод comparator() повертає null, тому що відсортований сет було створено без компаратора.
  - Замість цього відсортований сет покладається на природне впорядкування елементів типу String.

```
   static void dump(String title, SortedSet<String> sss)
   {
      System.out.print(title + " ");
      for (String s: sss)
         System.out.print(s + " ");
      System.out.println();
   }
}
```

Вивід

- Вивід показує, що розмір відсортованого сету менше за розмір масиву, оскільки сет не може містити дублікати (banana, turnip, onion).

```
Array size = 29
sss: apple banana blueberry carrot cherry corn cucumber grape kiwi lemon mango melon nectarine
onion peach pear pepper plum potato radish raspberry squash strawberry tomato turnip zucchini
Sorted set size = 26
First element = apple
Last element = zucchini
Comparator = null
hs: apple banana blueberry carrot cherry corn cucumber grape kiwi lemon mango melon
ts: nectarine onion peach pear pepper plum potato radish raspberry squash strawberry tomato
turnip zucchini
Count of p-named fruits & vegetables = 5
Incorrect count of c-named fruits & vegetables = 3
Correct count of c-named fruits & vegetables = 4
```

Власний клас Employee, що не реалізує Comparable

```
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
   public static void main(String[] args)
   {
      SortedSet<Employee> sse = new TreeSet<Employee>();
      sse.add(new Employee("Sally Doe"));
      sse.add(new Employee("Bob Doe")); // ClassCastException
      sse.add(new Employee("John Doe"));
      System.out.println(sse);
   }
}

class Employee
{
   private String name;

   Employee(String name)
   {
      this.name = name;
   }

   @Override
   public String toString()
   {
      return name;
   }
}
```

- Викидається екземпляр ClassCastException під час другого виклику методу add(), оскільки екземпляр TreeSet – не може викликати метод compareTo() для елементів другого об'єкту Employee.
  - Пов'язано з відсутністю реалізації інтерфейсу Comparable в Employee.

```
Exception in thread "main" java.lang.ClassCastException: Employee cannot be cast to
java.lang.Comparable
        at java.util.TreeMap.compare(Unknown Source)
        at java.util.TreeMap.put(Unknown Source)
        at java.util.TreeSet.add(Unknown Source)
        at CustomClassAndSortedSet.main(CustomClassAndSortedSet.java:9)
```

Власний клас Employee, що реалізує Comparable

- Вирішення проблеми – мати реалізацію Comparable у вигляді класу.

```
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
   public static void main(String[] args)
   {
      SortedSet<Employee> sse = new TreeSet<Employee>();
      sse.add(new Employee("Sally Doe"));
      sse.add(new Employee("Bob Doe"));
      Employee e1 = new Employee("John Doe");
      Employee e2 = new Employee("John Doe");
      sse.add(e1);
      sse.add(e2);
      System.out.println(sse);
      System.out.println(e1.equals(e2));
   }
}
```

```
class Employee implements Comparable<Employee>
{
   private String name;

   Employee(String name)
   {
      this.name = name;
   }

   @Override
   public int compareTo(Employee e)
   {
      return name.compareTo(e.name);
   }

   @Override
   public String toString()
   {
      return name;
   }
}
```

Вивід додатку

```
Bob Doe, John Doe, Sally Doe]
false
```

- Показує лише один "John Doe" у відсортованому сеті.
- Проте значення false (результат порівняння equals()) також вказує на те, що природний порядок відсортованого сету неузгоджений з equals(), що порушує контракт SortedSet:
  - Порядок, що підтримується в sorted set (незалежно від постачання явного компаратора) має узгоджуватись з equals(), якщо відсортований сет коректно реалізує інтерфейс Set.
  - Оскільки інтерфейс Set визначено в термінах операції equals(), проте відсортований сет виконує всі порівняння елементів через метод compareTo() (або compare()).

Клас Employee, що відповідає контракту

```
import java.util.SortedSet;
import java.util.TreeSet;

public class CustomClassAndSortedSet
{
   public static void main(String[] args)
   {
      SortedSet<Employee> sse = new TreeSet<Employee>();
      sse.add(new Employee("Sally Doe"));
      sse.add(new Employee("Bob Doe"));
      Employee e1 = new Employee("John Doe");
      Employee e2 = new Employee("John Doe");
      sse.add(e1);
      sse.add(e2);
      System.out.println(sse);
      System.out.println(e1.equals(e2));
   }
}
```

```
class Employee implements Comparable<Employee>
{
   private String name;

   Employee(String name)
   {
      this.name = name;
   }

   @Override
   public int compareTo(Employee e)
   {
      return name.compareTo(e.name);
   }
   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Employee))
         return false;
      Employee e = (Employee) o;
      return e.name.equals(name);
   }

   @Override
   public String toString()
   {
      return name;
   }
}
```

Виправляє контракт, переозначуючи метод equals(), проте hashCode() ігнорується в tree-based sorted sets.

```
[Bob Doe, John Doe, Sally Doe]
true
```

ОГЛЯД НАВІГАЦІЙНИХ СЕТІВ (NAVIGABLE SETS)

Навіговані (Navigable) множини

- TreeSet – приклад *navigable set*, відсортованої множини, прохід по якій можливий як у зростаючому, так і в спадному порядку. Також вона може повідомляти про найближчі співпадіння з результатами пошуку.
- Navigable sets описуються інтерфейсом NavigableSet (узагальнений тип NavigableSet<E>), який розширює SortedSet.

Method	Description
`E ceiling(E e)`	Returns the least element in this set greater than or equal to `e`, or null when there is no such element. This method throws `ClassCastException` when `e` cannot be compared with the elements currently in the set and `NullPointerException` when `e` is `null` and this set doesn't permit null elements.
`Iterator<E> descendingIterator()`	Returns an iterator over the elements in this set, in descending order. Equivalent in effect to `descendingSet().iterator()`.
`NavigableSet<E> descendingSet()`	Returns a reverse order view of the elements contained in this set. The descending set is backed by this set, so changes to the set are reflected in the descending set and vice versa. If either set is modified (except through the iterator's own `remove()` operation) while iterating over the set, the results of the iteration are undefined.

Method	Description
`E floor(E e)`	Returns the greatest element in this set less than or equal to `e` or null when there is no such element. This method throws `ClassCastException` when `e` cannot be compared with the elements currently in the set and `NullPointerException` when `e` is `null` and this set doesn't permit null elements.
`NavigableSet<E> headSet(E toElement, boolean inclusive)`	Returns a view of the portion of this set whose elements are less than (or equal to, when `inclusive` is `true`) `toElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `toElement` is not compatible with this set's comparator (or, when the set has no comparator, when `toElement` doesn't implement `Comparable`), `NullPointerException` when `toElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when this set has a restricted range and `toElement` lies outside of this range's bounds.
`E higher(E e)`	Returns the least element in this set strictly greater than the given element or null when there is no such element. This method throws `ClassCastException` when `e` cannot be compared with the elements currently in the set and `NullPointerException` when `e` is `null` and this set doesn't permit null elements.
`E lower(E e)`	Returns the greatest element in this set strictly less than the given element or null when there is no such element. This method throws `ClassCastException` when `e` cannot be compared with the elements currently in the set and `NullPointerException` when `e` is `null` and this set doesn't permit null elements.
`E pollFirst()`	Returns and removes the first (lowest) element from this set, or returns null when this set is empty.
`E pollLast()`	Returns and removes the last (highest) element from this set, or returns null when this set is empty.

`NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)`	Returns a view of the portion of this set whose elements range from `fromElement` to `toElement`. (When `fromElement` and `toElement` are equal, the returned set is empty unless `fromInclusive` and `toInclusive` are both `true`.) The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `fromElement` and `toElement` cannot be compared to one another using this set's comparator (or, when the set has no comparator, using natural ordering), `NullPointerException` when `fromElement` or `toElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when `fromElement` is greater than `toElement` or when this set has a restricted range and `fromElement` or `toElement` lies outside of this range's bounds.

Method	Description
`NavigableSet<E> tailSet(E fromElement, boolean inclusive)`	Returns a view of the portion of this set whose elements are greater than (or equal to, when `inclusive` is `true`) `fromElement`. The returned set is backed by this set, so changes in the returned set are reflected in this set and vice versa. The returned set supports all optional set operations that this set supports. This method throws `ClassCastException` when `fromElement` is not compatible with this set's comparator (or, when the set has no comparator, when `fromElement` doesn't implement `Comparable`), `NullPointerException` when `fromElement` is `null` and this set doesn't permit null elements, and `IllegalArgumentException` when this set has a restricted range and `fromElement` lies outside of this range's bounds.

```
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.TreeSet;

public class NavigableSetDemo
{
   public static void main(String[] args)
   {
      NavigableSet<Integer> ns = new TreeSet<Integer>();
      int[] ints = { 82, -13, 4, 0, 11, -6, 9 };
      for (int i: ints)
         ns.add(i);
      System.out.print("Ascending order: ");
      Iterator iter = ns.iterator();
      while (iter.hasNext())
         System.out.print(iter.next() + " ");
      System.out.println();
      System.out.print("Descending order: ");
      iter = ns.descendingIterator();
      while (iter.hasNext())
         System.out.print(iter.next() + " ");
      System.out.println("\n");
      outputClosestMatches(ns, 4);
      outputClosestMatches(ns.descendingSet(), 12);
   }

   static void outputClosestMatches(NavigableSet<Integer> ns, int i)
   {
      System.out.println("Element < " + i + " is " + ns.lower(i));
      System.out.println("Element <= " + i + " is " + ns.floor(i));
      System.out.println("Element > " + i + " is " + ns.higher(i));
      System.out.println("Element >= " + i +" is " + ns.ceiling(i));
      System.out.println();
   }
}
```

Навігація у множині цілих чисел

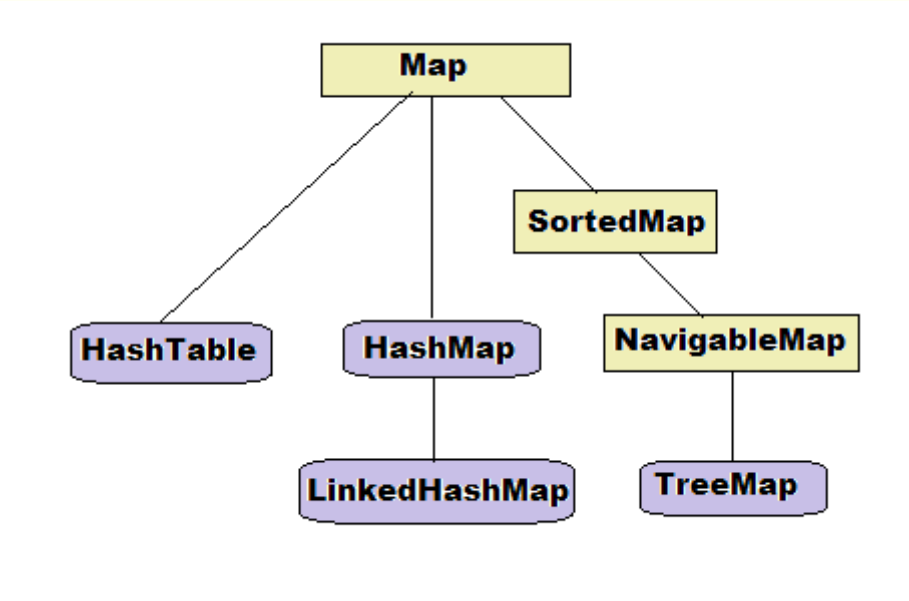
- Лістинг створює navigable set елементів типу Integer.
  - Використовуються переваги автоупаковки, щоб забезпечувати конвертацію int в Integer.
  - Вивід програми:

```
Ascending order: -13 -6 0 4 9 11 82
Descending order: 82 11 9 4 0 -6 -13

Element < 4 is 0
Element <= 4 is 4
Element > 4 is 9
Element >= 4 is 4

Element < 12 is 82
Element <= 12 is 82
Element > 12 is 11
Element >= 12 is 11
```

Мепи (карти, Maps)



- Меп (карта) - група пар «ключ-значення» (відомі як *entries*).
  - Оскільки ключ ідентифікує entry, меп не може мати дублікати ключів.
  - Більше того, кожен ключ може відображати максимум одне значення.
  - Мепи описуються інтерфейсом Map (узагальнений тип Map<K,V>), який не має батьківського інтерфейсу.

Method	Description
`void clear()`	Removes all elements from this map, leaving it empty. This method throws `UnsupportedOperationException` when `clear()` is not supported.
`boolean containsKey(Object key)`	Returns true when this map contains an entry for the specified `key`; otherwise, returns false. This method throws `ClassCastException` when `key` is of an inappropriate type for this map and `NullPointerException` when `key` contains the null reference and this map doesn't permit null keys.
`boolean containsValue(Object value)`	Returns true when this map maps one or more keys to `value`. This method throws `ClassCastException` when `value` is of an inappropriate type for this map and `NullPointerException` when `value` contains the null reference and this map doesn't permit null values.
`Set<Map.Entry<K,V>> entrySet()`	Returns a `Set` view of the entries contained in this map. Because this map backs the view, changes that are made to the map are reflected in the set and vice versa.
`boolean equals(Object o)`	Compares `o` with this map for equality. Returns true when `o` is also a map and the two maps represent the same entries; otherwise, returns false.
`V get(Object key)`	Returns the value to which `key` is mapped or null when this map contains no entry for `key`. If this map permits null values, then a return value of null doesn't necessarily indicate that the map contains no entry for `key`; it is also possible that the map explicitly maps `key` to the null reference. The `containsKey()` method may be used to distinguish between these two cases. This method throws `ClassCastException` when `key` is of an inappropriate type for this map and `NullPointerException` when `key` contains the null reference and this map doesn't permit null keys.

`int hashCode()`	Returns the hash code for this map. A map's hash code is defined to be the sum of the hash codes for the entries in the map's `entrySet()` view.
`boolean isEmpty()`	Returns true when this map contains no entries; otherwise, returns false.
`Set<K> keySet()`	Returns a `Set` view of the keys contained in this map. Because this map backs the view, changes that are made to the map are reflected in the set and vice versa.

Method	Description
`V put(K key,V value)`	Associates `value` with `key` in this map. If the map previously contained an entry for `key`, the old value is replaced by `value`. This method returns the previous value associated with `key` or null when there was no entry for `key`. (The null return value can also indicate that the map previously associated the null reference with `key`, if the implementation supports null values.) This method throws `UnsupportedOperationException` when `put()` is not supported, `ClassCastException` when `key`'s or `value`'s class is not appropriate for this map, `IllegalArgumentException` when some property of `key` or `value` prevents it from being stored in this map, and `NullPointerException` when `key` or `value` contains the null reference and this map doesn't permit null keys or values.
`void putAll(Map<? extends K,? extends V> m)`	Copies all entries from map `m` to this map. The effect of this call is equivalent to that of calling `put(k, v)` on this map once for each mapping from key `k` to value `v` in map `m`. This method throws `UnsupportedOperationException` when `putAll()` is not supported, `ClassCastException` when the class of a key or value in map `m` is not appropriate for this map, `IllegalArgumentException` when some property of a key or value in map `m` prevents it from being stored in this map, and `NullPointerException` when `m` contains the null reference or when `m` contains null keys or values and this map doesn't permit null keys or values.

`V remove(Object key)`	Removes key's entry from this map when it is present. This method returns the value to which this map previously associated with key or null when the map contained no mapping for key. If this map permits null values, then a return value of null doesn't necessarily indicate that the map contained no entry for key; it is also possible that the map explicitly mapped key to null. This map will not contain an entry for key once the call returns. This method throws `UnsupportedOperationException` when `remove()` is not supported, `ClassCastException` when the class of key is not appropriate for this map, and `NullPointerException` when key contains the null reference and this map doesn't permit null keys.
`int size()`	Returns the number of key/value entries in this map. If the map contains more than `Integer.MAX_VALUE` entries, this method returns `Integer.MAX_VALUE`.
`Collection<V> values()`	Returns a `Collection` view of the values contained in this map. Because this map backs the view, changes that are made to the map are reflected in the collection and vice versa.

- На відміну від List, Set та Queue, інтерфейс Map не розширює Collection.
  - Проте існує можливість розглядати меп як екземпляр Collection, викликавши методи keySet(), values() та entrySet() інтерфейсу Map, які повертають сет ключів, колекцію значень та сет key/value pair entries відповідно.
- **Зауважте!** Метод values() поветрає Collection замість Set, оскільки кілька ключів може відображатись на одне значення, що призведе до повернення від values() кількох копій того ж значення.

```
enum Color
{
   RED(255, 0, 0),
   GREEN(0, 255, 0),
   BLUE(0, 0, 255);

   private int r, g, b;

   private Color(int r, int g, int b)
   {
      this.r = r;
      this.g = g;
      this.b = b;
   }

   @Override
   public String toString()
   {
      return "r = " + r + ", g = " + g + ", b = " + b;
   }
}
```

- У прикладі оголошується map of String keys та Color values, додається кілька entries у map, а потім йде ітерування по ключах та значеннях.
- Вивід буде подібним до такого:

```
red
blue
green
RED
r = 255, g = 0, b = 0
r = 0, g = 0, b = 255
r = 0, g = 255, b = 0
r = 255, g = 0, b = 0
```

```
Map<String, Color> colorMap = ...; // ... represents the creatior
colorMap.put("red", Color.RED);
colorMap.put("blue", Color.BLUE);
colorMap.put("green", Color.GREEN);
colorMap.put("RED", Color.RED);
for (String colorKey: colorMap.keySet())
   System.out.println(colorKey);
Collection<Color> colorValues = colorMap.values();
for (Iterator<Color> it = colorValues.iterator(); it.hasNext();)
   System.out.println(it.next());
```

Методи *Map.Entry*

Method	Description
`boolean equals(Object o)`	Compares `o` with this entry for equality. Returns true when `o` is also a map entry and the two entries have the same key and value.
`K getKey()`	Returns this entry's key. This method optionally throws `IllegalStateException` when this entry has previously been removed from the backing map.
`V getValue()`	Returns this entry's value. This method optionally throws `IllegalStateException` when this entry has previously been removed from the backing map.
`int hashCode()`	Returns this entry's hash code.
`V setValue(V value)`	Replaces this entry's value with `value`. The backing map is updated with the new `value`. This method throws `UnsupportedOperationException` when `setValue()` is not supported, `ClassCastException` when `value`'s class prevents it from being stored in the backing map, `NullPointerException` when `value` contains the null reference and the backing map doesn't permit null, `IllegalArgumentException` when some property of `value` prevents it from being stored in the backing map, and (optionally) `IllegalStateException` when this entry has previously been removed from the backing map.

- Приклад ітерування по map entries:

```
for (Map.Entry<String, Color> colorEntry: colorMap.entrySet())
   System.out.println(colorEntry.getKey() + ": " + colorEntry.getValue());
```

- Приклад виводу:

```
red: r = 255, g = 0, b = 0
blue: r = 0, g = 0, b = 255
green: r = 0, g = 255, b = 0
RED: r = 255, g = 0, b = 0
```

Клас TreeMap

- Клас TreeMap забезпечує реалізацію мепу, що базується на червоно-чорних деревах.
  - Всі entries відсортовані по ключах.
  - Проте обхід дещо повільніший, ніж для інших реалізацій Map.
- Конструктори TreeMap
  - TreeMap() створює новий, порожній tree map, який зберігається згідно з природним порядком його ключів.
    - Всі додані в меп ключі мають реалізовувати інтерфейс Comparable.
  - TreeMap(Comparator<? super K> comparator) створює новий, порожній tree map, який зберігається згідно із заданим компаратором.
    - Передача null в компаратор передбачає використання природного порядку.
  - TreeMap(Map<? extends K, ? extends V> m) створює новий tree map, що містить m's entries, відсортовані відповідно до природного порядку його ключів.
    - Всі додані в меп ключі мають реалізовувати інтерфейс Comparable.
    - Конструктор викидає ClassCastException, коли ключі m не реалізують Comparable або їх неможливо порівняти, а також NullPointerException, якщо m містить null reference.
  - TreeMap(SortedMap<K, ? extends V> sm) створює новий tree map, що містить ті ж entries та використовує той же порядок, що й sm.
    - Конструктор викидає NullPointerException, коли sm містить null-посилання.

Сортування Map’s Entries відповідно до природного порядку їх String-Based ключів

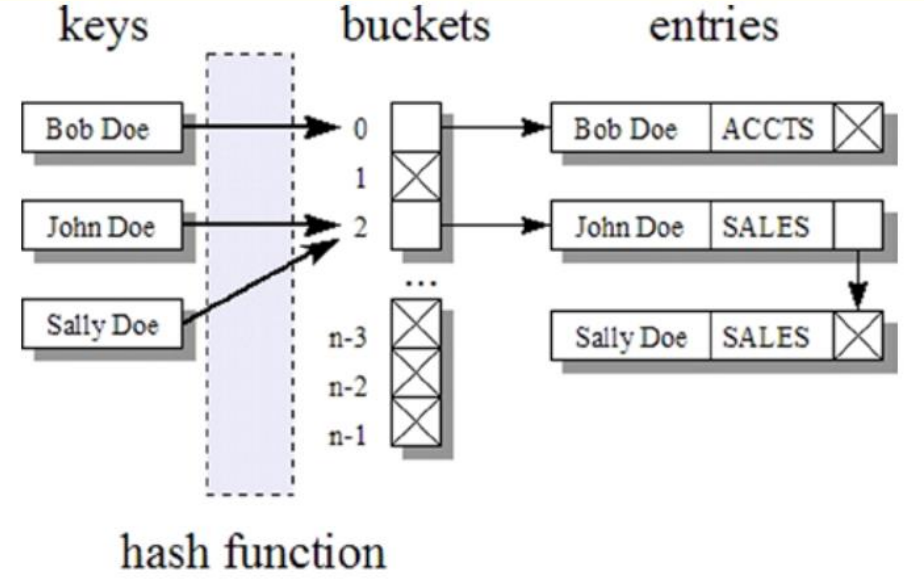
```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapDemo
{
   public static void main(String[] args)
   {
      Map<String, Integer> msi = new TreeMap<String, Integer>();
      String[] fruits = {"apples", "pears", "grapes", "bananas", "kiwis"};
      int[] quantities = {10, 15, 8, 17, 30};
      for (int i = 0; i < fruits.length; i++)
         msi.put(fruits[i], quantities[i]);
      for (Map.Entry<String, Integer> entry: msi.entrySet())
         System.out.println(entry.getKey() + ": " + entry.getValue());
   }
}
```

```
apples: 10
bananas: 17
grapes: 8
kiwis: 30
pears: 15
```

- TreeMapDemo створює tree map та масив з назвами фруктів.
  - Потім меп заповнюється цими назвами та виводить свої елементи у стандартний вивід.

Клас HashMap



- Постачає реалізацію map, яка базується на хеш-таблицях.
  - Ця реалізація підтримує всі операції з Map та дозволяє null keys та null values.
  - Гарантій впорядкованості Елементів немає.
- Хеш-функція хешує Bob Doe в 0, що визначає перший bucket.
  - Він містить ACCTS, тип яких – Bob Doe’s employee.
  - Хеш-функція також хешує John Doe та Sally Doe в 1 та 2 відповідно (buckets contain SALES).
  - В ідеалі, хеш-функція має хешувати кожен ключ в унікальне цілочисельне значення.
  - На практиці деякі ключі будуть хешуватись в одне цілочисельне значення. Це колізія.
  - Для вирішення колізій більшість хеш-таблиць асоціюють з коміркою (bucket) зв’язний список значень.
  - Комірка містить адресу першого вузла і зв’язному списку, а кожен вузол містить одне з колізійних значень.

Хеш-таблиці та коефіцієнт заповнення

- При збереженні значення в хеш-таблиці (hashtable) використовується хеш-функція, яка хешує ключ у хеш-код, а потім шукає підходящий зв'язний список, щоб перевірити наявність у ньому відповідного ключа.
  - Якщо таке входження (entry) є, значення оновлюється.
  - Інакше створюється новий вузол з даними ключем/значенням та додається в список.
- Аналогічно отримується значення з хеш-таблиці: хешований ключ дозволяє знайти відповідний список входжень та перевірити існування значення з відповідним ключем.
  - Якщо входження присутнє, повертається його значення.
  - Інакше хеш-таблиця може повернути спеціальне значення, що вказує на відсутність входження, чи викинути виключення.
- Кількість комірок (buckets) називають ***місткістю (capacity) хеш-таблиці***.
- Відношення кількості збережених entries до кількості комірок називають ***коефіцієнтом заповнення (load factor) хеш-таблиці***.
  - Вибір коефіцієнту заповнення важливий для балансування продуктивності та використання пам'яті.
  - При наближенні до 1 ймовірність колізій та «вартість» їх обробки (пошук у довгому списку) зростає.
  - При наближенні до 0, розмір хеш-таблиці (з точки зору кількості комірок) зростає з невеликим зниженням «вартості» пошуку.
  - Для багатьох хеш-таблиць коефіцієнт заповнення встановлюють на рівні 0.75, близькому до оптимального.
    - Значення за замовчуванням у реалізації хеш-таблиці в рамках HashMap.

Конструктори HashMap

- **`HashMap()`** створює новий, порожній hashmap з початковою місткістю (capacity) – 16 та коефіцієнтом заповнення – 0.75.
- **`HashMap(int initialCapacity)`** створює новий, порожній hashmap з початковою місткістю initialCapacity та коефіцієнтом заповнення – 0.75.
  - Викидає виключення IllegalArgumentException, якщо initialCapacity < 0.
- **`HashMap(int initialCapacity, float loadFactor)`** creates a new, empty hashmap with a capacity specified by initialCapacity and a load factor specified by loadFactor.
  - Викидає виключення IllegalArgumentException, коли initialCapacity < 0 або loadFactor <= 0.
- **`HashMap(Map<? extends K, ? extends V> m)`** створює новий hashmap, що містить входження m.
  - Викидає NullPointerException, якщо m містить null-посилання.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo
{
   public static void main(String[] args)
   {
      Map<String, Integer> argMap = new HashMap<String, Integer>();
      for (String arg: args)
      {
         Integer count = argMap.get(arg);
         argMap.put(arg, (count == null) ? 1 : count + 1);
      }
      System.out.println(argMap);
      System.out.println("Number of distinct arguments = " + argMap.size());
   }
}
```

- HashMapDemo створює хешмеп ключів типу String та цілочисельних значень.
  - Кожен ключ – один з аргументів командного рядка, а значення – кількість появ аргументу в командному рядку.
- Запуск:
  - **`java HashMapDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood`**
    {wood=2, could=2, how=1, if=1, chuck=2, a=2, woodchuck=2, much=1}
    Number of distinct arguments = 8

Переозначення hashCode()

- Оскільки клас String переозначає equals() та hashCode(), String-об’єкти можна використовувати як ключі хешмепу.
  - Коли створюється клас, чиї екземпляри будуть використовуватись як ключі, необхідно забезпечити перевизначення обох методів.
- Частіше за все, класи оголошують кілька полів, тому потрібно продумати кращу методу реалізацію hashCode().
  - Реалізація має генерувати хеш-коди, що мінімізують колізії.
  - Правила вибору для кращої реалізації hashCode() та різні алгоритми (recipes for accomplishing tasks) have been created.
  - Один з ефективних алгоритмів показано в *Effective Java, Second Edition*, by Joshua Bloch (Addison-Wesley, 2008; ISBN: 0321356683).

Приклад алгоритму, схожого на алгоритм Блоха (для довільного класу X)

- 1. Ініціалізувати змінну int hashCode (назва довільна) деяким ненульовим цілим значенням, наприклад, 19.
  - Мета: переконатись, що беруться до уваги будь-які initial fields, чиї хеш-коди дорівнюють 0.
  - Якщо ініціалізувати hashCode = 0, на final hash code не будуть впливати такі поля, тому виникає небезпека зростання кількості колізій.
- 2. Для кожного поля f, яке також використовується в методі equals() класу X, обчислити хеш-код та присвоїти його цілочисельній змінній hc:
  - Якщо f має тип Boolean, обчислюйте hc = f ? 1 : 0.
  - Якщо f має тип byte integer, character, integer чи short integer type, обчислити hc = (int) f. Integer-значення є хеш-кодом.
  - Якщо f має тип long integer, обчислюйте hc = (int) (f ^ (f >>> 32)).
  - Якщо f має тип floating-point, обчислюйте hc = Float.floatToIntBits(f). Метод враховує +infinity, -infinity та NaN.
  - Якщо f має тип double precision floating-point, обчислюйте long l = Double.doubleToLongBits(f);
    hc = (int) (l ^ (l >>> 32)).
  - Якщо f є посилковим полем з null-посиланням, обчислюйте hc = 0.
  - Якщо f є посилковим полем з не-null посиланням, а метод equals() класу X порівнює поле, рекурсивно викликаючи equals(), обчислюйте hc = f.hashCode(). Проте якщо equals() виконує складніше порівняння, створіть *канонічне* (найпростіше можливе) представлення поля і викличте hashCode() для цього представлення.
  - Якщо f – масив, розглядайте кожен елемент як окреме поле, застосовуючи цей алгоритм рекурсивно та комбінуючи значення hc, як показано на наступному кроці.

- 3. Комбінуйте hc з hashCode так: hashCode = hashCode * 31 + hc.
  - Множення hashCode на 31 робить утворене значення хешу залежним від порядку появи полів у класі, що покращує значення хешу, коли клас містить кілька схожих полів (кілька int-ів, наприклад).
  - Вибрано 31 для того, щоб бути consistent з методом hashCode() класу String.
- 4. Повертайте hashCode з hashCode().

```
import java.util.HashMap;
import java.util.Map;

public class Point
{
   private int x, y;

   Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   int getX()
   {
      return x;
   }

   int getY()
   {
      return y;
   }

   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Point))
         return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
   }
```

Переозначення hashCode() для повернення доречних хеш-кодів для об’єктів Point

- Внесемо правки в код класу Point з теми 3, оголосивши метод hashCode().
  - Метод використовує попередній алгоритм, щоб переконатись у логічній еквівалентності об’єктів Point.
  - Якщо переозначити equals(), але не hashCode(), Ви порушите другий пункт контракту: хеш-коди однакових об’єктів повинні бути однаковими.
  - Серйозні наслідки:

```
java.util.Map<Point, String> map = new java.util.HashMap<Point, String>();
map.put(p1, "first point");
System.out.println(map.get(p1)); // Output: first point
System.out.println(map.get(new Point(10, 20))); // Output: null
```

```
@Override
public int hashCode()
{
   int hashCode = 19;
   int hc = x;
   hashCode = hashCode * 31 + hc;
   hc = y;
   hashCode = hashCode * 31 + hc;
   return hashCode;
}
```

Метод main()

```
public static void main(String[] args)
{
    Point p1 = new Point(10, 20);
    Point p2 = new Point(20, 30);
    Point p3 = new Point(10, 20);
    // Test reflexivity
    System.out.println(p1.equals(p1));              // Output: true
    // Test symmetry
    System.out.println(p1.equals(p2));              // Output: false
    System.out.println(p2.equals(p1));              // Output: false
    // Test transitivity
    System.out.println(p2.equals(p3));              // Output: false
    System.out.println(p1.equals(p3));              // Output: true
    // Test nullability
    System.out.println(p1.equals(null));            // Output: false
    // Extra test to further prove the instanceof operator's usefulness.
    System.out.println(p1.equals("abc"));           // Output: false
    Map<Point, String> map = new HashMap<Point, String>();
    map.put(p1, "first point");
    System.out.println(map.get(p1));                // Output: first point
    System.out.println(map.get(new Point(10, 20))); // Output: first point
}
}
```

- При запуску додатку найбільш цікаві два останніх рядки виводу.
  - Замість представлення first point followed by null on two separate lines, тепер додаток коректно представляє першу точку після першої точки в цих рядках.

ДЯКУЮ ЗА УВАГУ!

IdentityHashMap

- Клас IdentityHashMap забезпечує реалізацію Map, яка використовує еквівалентність посилань (==) замість еквівалентності об’єктів (equals()) при порівнянні ключів та значень.
  - Це цілеспрямоване порушення загального контракту для Map, який санкціонує використання методу equals() при порівнянні елементів.
- IdentityHashMap отримує хеш-коди за допомогою методу int identityHashCode(Object x) класу System замість методу hashCode() кожного ключа.
  - identityHashCode() повертає той же хеш-код для x, що і метод Object.hashCode(), незалежно від того, чи був переозначений метод hashCode().
  - Хеш-код для null-посилання дорівнюватиме 0.
  - Через це IdentityHashMap працює швидше інших реалізацій Map.
- Також IdentityHashMap підтримує *змінювані* (*mutable*) *ключі* (об’єкти, що використовуються як ключі та чиї хеш-коди змінюються при зміні значень полів у мепі).

Відмінності між IdentityHashMap і HashMap у контексті змінюваних ключів

```
class Employee
{
   private String name;
   private int age;

   Employee(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

   @Override
   public boolean equals(Object o)
   {
      if (!(o instanceof Employee))
         return false;
      Employee e = (Employee) o;
      return e.name.equals(name) && e.age == age;
   }

   @Override
   public int hashCode()
   {
      int hashCode = 19;
      hashCode = hashCode * 31 + name.hashCode();
      hashCode = hashCode * 31 + age;
      return hashCode;
   }

   void setAge(int age)
   {
      this.age = age;
   }

   void setName(String name)
   {
      this.name = name;
   }

   @Override
   public String toString()
   {
      return name + " " + age;
   }
}
```

Contrasting IdentityHashMap with HashMap in a Mutable Key Context

```
import java.util.IdentityHashMap;
import java.util.HashMap;
import java.util.Map;

public class IdentityHashMapDemo
{
   public static void main(String[] args)
   {
      Map<Employee, String> map1 = new IdentityHashMap<Employee, String>();
      Map<Employee, String> map2 = new HashMap<Employee, String>();
      Employee e1 = new Employee("John Doe", 28);
      map1.put(e1, "SALES");
      System.out.println(map1);
      Employee e2 = new Employee("Jane Doe", 26);
      map2.put(e2, "MGMT");
      System.out.println(map2);
      System.out.println("map1 contains key e1 = " + map1.containsKey(e1));
      System.out.println("map2 contains key e2 = " + map2.containsKey(e2));
      e1.setAge(29);
      e2.setAge(27);
      System.out.println(map1);
      System.out.println(map2);
      System.out.println("map1 contains key e1 = " + map1.containsKey(e1));
      System.out.println("map2 contains key e2 = " + map2.containsKey(e2));
   }
}
```

- Метод main() створює екземпляри IdentityHashMap та HashMap, that each store an entry consisting of an Employee key and a String value.
- Оскільки екземпляри Employee є змінюваними (setAge() і setName() присутні), main() змінює їх вік, у той час, як ці ключі збережено у відповідних мепах.

Результати виводу

```
{John Doe 28=SALES}
{Jane Doe 26=MGMT}
map1 contains key e1 = true
map2 contains key e2 = true
{John Doe 29=SALES}
{Jane Doe 27=MGMT}
map1 contains key e1 = true
map2 contains key e2 = false
```

- Останні 4 рядка показують, що змінені входження залишаються у своїх мепах.
  - Проте метод containsKey() з map2 повідомляє, що цей екземпляр HashMap більше не містить ключа Employee (має бути Jane Doe 27),
  - containsKey() для map1 показує, що цей екземпляр IdentityHashMap все ще містить its Employee key (John Doe 29).

ENUMMAP

Клас EnumMap постачає реалізацію інтерфейсу Map, чиї ключі є членами одного enum.
Null-ключі заборонені; будь-яка спроба зберегти null key призводить до викидання NullPointerException.
Оскільки EnumMap внутрішньо представляється як масив, має передбачуваний з точки зору продуктивності поводиться як масив.

Конструктори EnumMap

- EnumMap(Class<K> keyType) створює порожній меп-перелічення із заданим keyType.
  - Викидає NullPointerException, коли keyType містить null-посилання.
- EnumMap(EnumMap<K,? extends V> m) створює меп-перелічення з same key type as m, and with m's entries.
  - Викидає NullPointerException, коли m містить null-посилання.
- EnumMap(Map<K,? extends V> m) створює меп-перелічення, що ініціалізується за допомогою входжень з m.
  - Якщо m – екземпляр EnumMap, конструктор поводиться як попередній.
  - Інакше m повинна містити принаймні одне входження, щоб визначити тип ключа для нового мепу-перелічення.
  - Викидає NullPointerException, коли m містить null-посилання, а також IllegalArgumentException, якщо m не є екземпляром EnumMap або порожній.

Меп-перелічення констант для монет

```
import java.util.EnumMap;
import java.util.Map;

enum Coin
{
    PENNY, NICKEL, DIME, QUARTER
}
```

```
public class EnumMapDemo
{
   public static void main(String[] args)
   {
      Map<Coin, Integer> map = new EnumMap<Coin, Integer>(Coin.class);
      map.put(Coin.PENNY, 1);
      map.put(Coin.NICKEL, 5);
      map.put(Coin.DIME, 10);
      map.put(Coin.QUARTER, 25);
      System.out.println(map);
      Map<Coin,Integer> mapCopy = new EnumMap<Coin, Integer>(map);
      System.out.println(mapCopy);
   }
}
```

- EnumMapDemo створює меп: ключі типу Coin, значення – цілочисельні.
- Далі в цей меп вставляється кілька екземплярів Coin, потім меп виводиться.
- У кінці створюємо копію мепу та виводимо його копію.

```
{PENNY=1, NICKEL=5, DIME=10, QUARTER=25}
{PENNY=1, NICKEL=5, DIME=10, QUARTER=25}
```

Самостійне опрацювання (приховані слайди)

- WeakHashMap
- Sorted Maps
- Navigable Maps
- Arrays and Collections Utility APIs
- Legacy Collection APIs