Лабораторна робота № 4

Робота з узагальненнями та використання колекцій

Мета роботи: Вивчити технологію створення і використання узагальнених типів і методів. Ознайомитися з фреймворком колекцій Java.

1. Теоретичні відомості

Узагальнені типи в Java

Узагальнення в Java (англ. *Generics*) – це можливість узагальненого програмування, яка була додана в мову програмування Java в 2004 році як частина стандартної платформи J2SE 5.0. *Узагальнення* (*generics*) – це параметризовані типи, що дозволяють оголошувати класи, інтерфейси і методи, у яких тип даних, яким вони оперують, зазначений як параметр. Такі класи, інтерфейси і методи називають *узагальненими*. Узагальнення дають можливість створювати класи (інтерфейси), методи (конструктори) таким чином, щоб вони могли оперувати різними типами даних (тільки посилальних) без реалізації програмістом декількох різних класів (інтерфейсів), методів (конструкторів). Після появи узагальнень, більшість класів (інтерфейсів) платформи Java були переписані для їх використання.

До появи узагальненого програмування було необхідно вручну виконувати приведення типів і, в разі його відсутності, з'являлася помилка під час виконання програми. За допомогою узагальнень при створенні об'єкта узагальненого класу або виклику узагальненого методу вказується, який тип буде використовуватися, і помилка присікається ще на етапі компіляції. Наприклад, нехай нам необхідно створити клас `MyBox`, об'єкт якого повинен зберігати як змінну класу будь-який об'єкт (об'єкт будь-якого класу). З огляду на, що всі класи неявно успадковуються від java.lang.Object, код `MyBox` може бути таким:

```
public class MyBox {

    private Object obj;  //зберігає об'єкт будь-якого класу

    public Object get() {
        return obj;
    }

    public void setInt(Object obj) {    // розміщення об'єкта
        if (obj instanceof Integer) {   //з перевіркою типу
            this.obj = obj;
        }
    }
}
```

У методі розміщення об'єкта в об'єкт `MyBox` використовується перевірка типу розміщується об'єкта (в прикладі дозволяється тільки тип `Integer`). Якщо потрібно розміщувати інші типи, доведеться писати аналогічний метод з перевіркою на цей тип. Зрозуміло, що вимога розмістити будь-який об'єкт при

такому підході виконана бути не може. Є вихід – залучити для перевірки типів рефлексію, у цьому випадку метод розміщення об'єкта буде виглядати так:

```
public void set(Object obj, String type)
                            throws ClassNotFoundException {
    if (obj.getClass() == Class.forName(type)) {
        this.obj = obj;
    }
}
```

Таке рішення враховує можливість появи помилки (exception) у разі, якщо в параметрі `type` буде невірно вказано повне ім'я класу об'єкта, який розміщується. Використання вищеописаного класу демонструється у наступному фрагменті:

```
public static void main(String[] args) throws
                                        ClassNotFoundException {
    MyBox mb = new MyBox();
    mb.setInt(5);
    mb.set(10, "java.lang.Integer");
    /* Помилка java.lang.ClassCastException: java.lang.Integer
      can not be cast to java.lang.String */
    String s = (String) mb.get();
}
```

Незважаючи на те, що тип розміщуваних об'єктів перевіряється перед розміщенням в об'єкт класу `MyBox`, помилки можуть виникати, якщо після вилучення розміщеного об'єкта привласнити його змінній іншого типу. І передбачити цю помилку в коді неможливо, залишається покладатися тільки на уважність програміста, що використовує наш клас.

Елегантне рішення проблеми надає використання узагальнень. Наш клас в узагальненому вигляді буде виглядати (перевірок вже немає, точніше вони перекладені на компілятор):

```
public class MyBox<T> {

    private T t;

    public T get() {
        return t;
    }

    public void set(T t) {
        this.t = t;
    }
}
```

`<T>` в оголошенні класу позначає *ім'я параметра* (або *змінної*) *типу*. Це ім'я використовується як заповнювач, замість якого при використанні класу підставляється ім'я конкретного типу – званого *аргументом типу*, переданого класу при створенні об'єкта цього класу. Така ж сама змінна типу використовується в

оголошенні змінної класу і методах. Приклад коду методу, що використовує клас MyBox<T>:

```
public static void main(String[] args) {
    MyBox<Integer> mb = new MyBox<>();
    mb.set(15);
    /* Тепер перевірка здійснюється компілятором
      і код не буде запущений */
// String s = (String) mb.get();

    MyBox<String> mb1 = new MyBox<>();
    mb1.set("Hello!");
    /* Тепер компілятор помилку не показує, тому тип,
       що зберігається, збігається з типом змінної,
       якій присвоюється */
    String s = (String) mb1.get();
}
```

У першому операторі методу `main(String[] args)` створюється об'єкт класу `MyBox<T>` із зазначенням у кутових дужках аргументу типу. Починаючи з Java 7, в імені конструктора аргумент типу може бути опущений (кутові дужки залишаються і називаються *оператор diamond*). У разі невідповідності типу об'єкту, який зберігається в об'єкті `MyBox<T>` типу змінної, якій присвоюється об'єкт, помилка з'явиться ще на етапі компіляції (що добре, оскільки код без її виправлення не запуститься). У другій частині коду створюється ще одне сховище-об'єкт `MyBox<T>`, але вже призначене для зберігання типу `String`. Ніяких переробок класу `MyBox<T>` не потрібно. При цьому при присвоєнні отриманого об'єкта змінній компілятор також здійснює перевірку типу.

Відзначимо, що існує угода щодо використання букв латинського алфавіту для позначення змінних типу:

```
<T> - для позначення будь-якого класу;
<E> - для елемента колекції;
<K, V> - для ключа і значення пари "ключ-значення";
<N> - для числового типу.
```

У разі, якщо в узагальненому класі (інтерфейсі, методі) використовуються кілька змінних типу (вони вказуються у кутових дужках через кому), для другої і наступних зміниих типу рекомендується використовувати літери `S`, `U`, `V` і т.д.

Існує можливість використання узагальнених класів як звичайних, тобто без вказівки аргументу типу при оголошенні і створенні об'єкта. Такі типи називають *сирими* (або *простими*) (*raw*) типами узагальнень, вони введені для сумісності зі старим (до JDK 5) кодом. При використанні сирих типів, втрачається перевага безпеки типів, що надається узагальненнями, тому їх використання рекомендується тільки при необхідності забезпечення сумісності зі старим кодом. Нижче наведено приклад використання узагальненого класу `MyBox<T>` як сирого типу:

```
public static void main(String[] args) {
```

```
    MyBox mb = new MyBox();
    mb.set(15);          // може бути переданий будь-який обєкт
    mb.set("Hello!");    // може бути переданий будь-який обєкт
    /* Однак при отриманні об'єкта потрібно ручне
      відповідне приведення типу */
    String s = (String) mb.get();
}
```

Узагальнені методи в Java

Для розгляду узагальнених методів поставимо завдання написання методу, що отримує з масиву будь-якого типу елемент за індексом. Очевидно, що в цій ситуації тип масиву і його елементів повинен бути `Object`:

```
public class MyClass {

    public Object getElementByIndex(Object[] arr, int index) {
        return arr[index];
    }
}
```

Використання методу для масивів різних типів показано у наступному коді:

```
public static void main(String[] args) {
    MyClass mc = new MyClass();
    Integer[] intArr = {1, 2, 3, 4, 5};
    Character[] charArr = {'A', 'B', 'C', 'D', 'E'};
    /* Знадобиться приведення типу */
    Integer intRes = (Integer) mc.getElementByIndex(intArr, 3);
    Character charRes =
                (Character) mc.getElementByIndex(charArr, 3);
    System.out.println("intRes =" + intRes + ", charRes ="
                    + CharRes);
}
```

Видно, що при присвоюванні результату методу знадобиться ручне відповідне приведення типів, що може бути джерелом помилок. Узагальнення і у цьому випадку виручають і дозволяють писати лаконічний і безпечний код. Наведемо код узагальненого методу, вводячи змінну типу `T`:

```
public class MyClass {

    public<T> T getElementByIndex(T[] arr, int index) {
        return arr[index];
    }
}
```

Зауважимо, що узагальнений метод може розташовуватися як в узагальненому, так і у звичайному класі. Використання узагальненого методу проілюстровано у наступному коді:

```
public static void main(String[] args) {
    MyClass mc = new MyClass();
    Integer[] intArr = {1, 2, 3, 4, 5};
    Character[] charArr = {'A', 'B', 'C', 'D', 'E'};
```

```
    /* Приведення типу не потрібно */
    Integer intRes = mc.getElementByIndex(intArr, 3);
    /* Компілятор знаходить помилку невідповідності типів */
// String str = mc.getElementByIndex(charArr, 3);
}
```

Ручне приведення типів вже не потрібне, а у разі невідповідності типів, помилка виявляється ще на стадії компіляції.

Дамо тепер визначення для узагальненого методу. *Узагальненим методом* називається метод, який вводить змінні типу в оголошенні і в тілі методу. Змінна типу (або кілька змінних типу) в оголошенні методу вказується у кутових дужках перед результатом, що повертається. Видимість типів, оголошених у методі, обмежується областю методу (як і для звичайних методів). Узагальнений метод, зазвичай, є узагальненням алгоритму для набору типів.

Існує можливість використання узагальнених типів і в параметрах методу. Наведемо приклад узагальненого методу`<T> int count(T[]arr, T elem)`, який підраховує кількість елементів масиву довільного типу `T[]arr`, рівних заданому як параметр аргументу `T elem`:

```
public class GenericsMethArgs {

    public static void main(String[] args) {
        String[] arr = {"yes", "no", "yes"};
        Integer[] arr1 = {1, 3, 1, 5, 1};
        System.out.println("Count of \"yes \"is:"
                           + count(arr, "yes"));
        System.out.println("Count of \"yes \"is:"
                           + count(arr1, 1));
    }
    /* Метод підрахунку елементів масиву, рівних заданому
      як параметр */
    public static<T> int count(T[] arr, T elem) {
        int count = 0;
        for (T e: arr) {
            if (e.equals(elem)) {
                ++count;
            }
        }
        return count;
    }
}
```

Обмеження узагальнених типів

Існує можливість обмежити набір типів, які можуть бути використані в якості аргументів типів узагальнених класів (інтерфейсів) і методів. Це досягається зазначенням батьківського класу після ключового слова `extends` при оголошенні параметра типу. При цьому прийнятними аргументами типу буде тип батька і будь-які типи його спадкоємців. Слід підкреслити, що при оголошенні узагальнених інтерфейсів з обмеженим набором типів також використовується ключове

слово `extends`. Уявімо, що необхідно ввести в клас `MyBox<T>` метод, можливий для роботи тільки з числами. В цьому випадку слід обмежити аргументи типу класу, тільки числовими типами - спадкоємцями абстрактного класу java.lang.Number. Це робиться у такий спосіб:

```
public class MyBox<T extends Number> {

    private T t;

    public T get() {
        return t;
    }

    public void set(T t) {
        this.t = t;
    }

    /* Обмеження типу дозволяє вводити методи, загальні для
      всіх спадкоємців типу-обмеження – у даному випадку
      метод перевіряє парність числа */
    public boolean isEven() {
        return t.intValue()% 2 == 0;
    }
}
```

Зверніть увагу, що обмеження типу у даному випадку вказується в оголошенні класу, а використовується в усіх його членах. Приклад використання такого класу і узагальненого методу продемонстровано у наступному коді:

```
public static void main(String[] args) {
    MyBox<Integer> mbi = new MyBox<>();
    mbi.set(10);
    System.out.println("MyBox<Integer> mbi зберігає"
                       + Mbi.get());
/* Помилка компіляції String НЕ спадкоємець Number */
// MyBox<String> mbs = new MyBox<>();
    System.out.println("mbi - парне: " + mbi.isEven());
}
```

При спробі передати аргумент типу, який не є спадкоємцем типу-обмежувача, видається помилка на етапі компіляції.

Існує можливість використання в якості типу-обмежувача імені інтерфейсу, у цьому випадку дозволеними аргументами типу будуть всі класи, що реалізують даний інтерфейс. Уявімо, що необхідно переписати метод `public static<T> int count(T[] arr, T elem)` у класі `GenericsMethArgs` так, щоб він підраховував не рівні заданому як параметр елементи, а більше за нього. Очевидно, що у цьому випадку необхідно, щоб тип масиву і елемента був класом, який реалізує інтерфейс java.lang.Comparable. Тепер можна сказати, що насправді ім'я цього інтерфейсу java.lang.Comparable<T>, тобто він узагальнений і його метод

`int compareTo(T o)` зможе перевизначатися в реалізаціях без ручного приведення типу з `Object` до необхідного типу:

```
@Override
    public int compareTo(ComparableEmployee o) {
        return this.name.compareTo(o.name);
     }
```

Нижче наведено код класу з методом з обмеженою змінною типу і його використання:

```
public class BoundedTypeMethod {

    /* Метод, що підраховує кількість елементів масиву,
      значення яких більше заданого як аргумент значення */
    public static<T extends Comparable<T>> int countGtThan(T[]
anArray, T elem) {
        int count = 0;
        for (T e: anArray) {
            if (e.compareTo(elem) > 0) {
                ++count;
            }
        }
        return count;
    }

    public static void main(String[] args) {
        String[] strArr = {"abc", "if", "no", "yes"};
        Integer[] intArr = {1, 3, 2, 5, 4};
        System.out.println("Кількість елементів, більше "
                    + \"If\"= " + countGtThan(strArr, "if"));
        System.out.println("Кількість елементів, більше 2 = "
                          + countGtThan(intArr, 2));
        A a1 = new A(15);
        A a2 = new A(2);
        A a3 = new A(10);
        A[] objArr = {a1, a2, a3};
        /* Компілятор знаходить помилку, оскільки об'єкти A
          не можна порівняти - вони не реалізують Comparable<A> */
// System.out.println("Кількість елементів, більше a2"
// + "=" + countGreaterThan (objArr, a2));
    }
    /* Вкладений довільний клас */
    static class A {

        int a;

        public A(int a) {
            this.a = a;
        }
    }
}
```

Оскільки і клас `Integer`, і клас `String` реалізують інтерфейс `Comparable<T>`, помилок компіляції при виклику методу`<T extends Comparable<T>> int countGtThan(T[] anArray, T elem)` не відбувається. Однак, як тільки методу передаються масив і об'єкт класу, що не реалізує інтерфейс `Comparable<T>`, компілятор відразу ж видає помилку. Пропонуємо самостійно реалізувати цей інтерфейс вкладеним класом `static class A` і перевірити працездатність узагальненого методу з обмеженням змінної типу.

Можливо обмеження набору типів параметрів одним батьківським класом і декількома інтерфейсами. Таке обмеження називається множинним і записується, наприклад, таким чином:

```
<T extends A & B1 & B2>
```

Слід підкреслити, що першим в перерахуванні батьків повинен вказуватися клас (його може й не бути), а за ним через `&` - інтерфейси.

Насправді, компілятор Java не створює різні версії узагальненого класу для різних аргументів типу. Він видаляє усю інформацію про узагальнені типи, виконуючи необхідні операції приведення типів, щоб зробити поведінку прикладного програмного коду такою, як ніби створена конкретна версія узагальненого класу. Процес видалення компілятором інформації про узагальнені типи називається *стиранням типів* (*type erasure*). Таким чином, є тільки одна версія узагальненого класу, ув якій параметр типу є типом `Object` (або батьківським типом для обмежених типів) – це видно при дизасемблюванні коду узагальнених класів із використанням дизасемблера клас-файлів javap (Рис. 1).

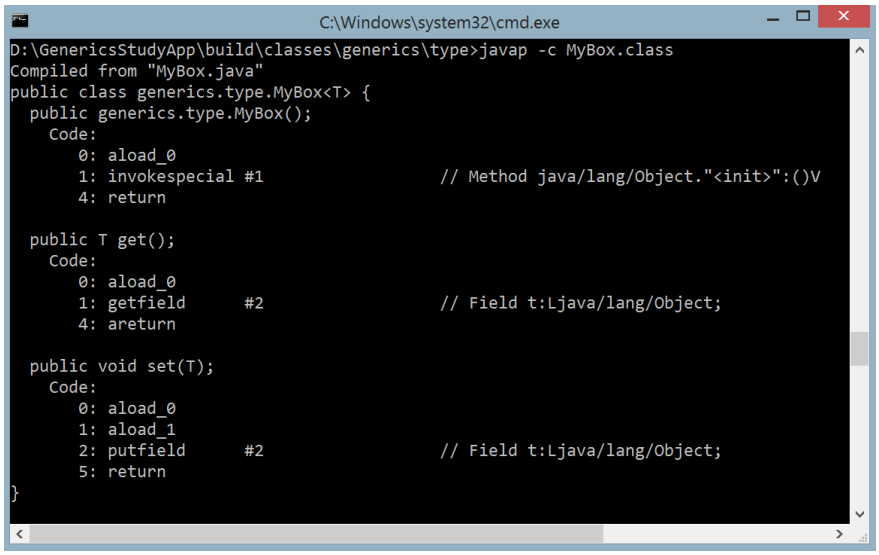


Рис. 1. Перегляд стирання типів в Object в дизассемблера

Використання метасимвольних змінних типу

Існує можливість використання метасімвольних змінних типу`<?>`. Як необмежених, так і обмежених зверху і знизу. Необмежені метасімвольние аргументи є корисними:

- в реалізаціях методів, які використовують функціональність класу `Object`;

- при використанні методів, що не залежать від параметра (змінної) типу (не посилається в тілі методу і не повертають змінну типу).

У наступному прикладі демонструється використання необмежених метасімвольних змінних типу:

```
public class UnboundedWildcards {

    public static void main(String[] args) {
        /* Застосування метасимволов в методах, які реалізують
          функціональність класу Object */
        Pair<?,?> Pair1 = new OrderedPair<>(1, 'a');
        Pair<?,?> Pair2 = new OrderedPair<>("one", "first");
        Pair<?,?> Pair3 = new OrderedPair<>(new int[] {1,2, 3},
                                            100L);
        printHash(pair1);
        printHash(pair2);
        printHash(pair3);

        /* Застосування метасимволов в методах, які використовують
           код, що не залежить від параметра (змінної) типу */
        List<String> strings = Arrays.asList("yes", "no");
        List<Integer> ints = Arrays.asList(1, 2, 3));
        printData(strings);
        printData(ints);

    /* Код методу реалізує функціональність класу Object */
    public static void printHash(Pair<?,?> pair) {
        System.out.println(pair.hashCode());
    }

    /* Код методу не залежить від параметра (змінної) типу */
    public static void printData(List<?> list) {
        for (Object object: list) {
            System.out.println(object + ";");
        }
    }
}
```

Існує можливість використання обмежених зверху метасімвольних аргументів зазначенням в якості типу<? extends Number>, наведемо приклад:

```
public static void main (String[] args) {
    List<Integer> ints = Arrays.asList(1, 3, 5);
    List<Double> doubles = Arrays.asList(2.2, 4.4, 6.6);

    System.out.println(sumNumberA(ints));
    System.out.println(sumNumberA(doubles));

    System.out.println(sumNumberB(ints));
    System.out.println(sumNumberB(doubles));
}
```

```
/* Метод, який приймає Number або будь-який його підтип як
  значення аргументу типу для списку. У методі не використовується
  змінна типу */
public static double sumNumberA(List<? extends Number> list) {
    double sum = 0;
    for (int i = 0; i<list.size(); i++) {
        sum += list.get(i).doubleValue();
    }
    return sum;
}

/* Метод, який приймає Number або будь-який його підтип як
  значення аргументу типу для списку. У методі використовується
  змінна типу */
public static<T extends Number> double sumNumberB(List<T>
                                                        list) {
    double sum = 0;
    for (T n: list) {// Використання змінної типу
        sum += n.doubleValue();
    }
    return sum;
}
```

У наведеному фрагменті коду демонструється рівнозначність використання обмежених зверху звичайних і метасімвольних змінних типу. Різниця полягає лише в тому, що використання у тілі методу метасімвольної змінної типу неможливе.

Іноді потрібно забезпечити можливість використання типів по ієрархії успадкування з обмеженням знизу, це також можна здійснити за допомогою обмежених метасімвольних змінних типів. У наведеному далі прикладі метод `addIntegers(List<? Super Integer> list, Object obj)` буде працювати з типами від `Integer` до `Object` вгору по ієрархії успадкування: `List<Integer>`, `List<Number>` та `List<Object>`:

```
public static void addIntegers(List<? super Integer> list,
                                                  Object obj) {
    list.add(new Integer((int) obj));
}

public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    List<Number> numList = new ArrayList<>();
    List objList = new ArrayList();
    /* Додамо до списків об'єкти типу параметра і батьківських
       для нього типів */
    int a = 5;
    addIntegers(intList, a);
    addIntegers(numList, a);
    addIntegers(objList, a);
    Number num = 20;
    /* Вбудованим методом списку */
```

```
// intList.add(num); // Помилка компіляції
    addIntegers(intList, num);
    addIntegers(numList, num);
    addIntegers(objList, num);
    Object obj = 30;
    /* Вбудованим методом списку */
// intList.add(obj); // Помилка компіляції
    addIntegers(intList, obj);
    addIntegers(numList, obj);
    addIntegers(objList, obj);

    System.out.println(intList);
    System.out.println(numList);
    System.out.println(objList);
}
```

Відзначимо, що використовувати ключове слово `super` для обмеження знизу звичайних (НЕ метасімвольних) змінних типу, наприклад,`<T super Integer> void addIntegersB(List<T> list, Object obj)` неможливо. Це пояснюється тим, що граничний тип для обмежених зверху змінних типу при стиранні типів замінює в байт-коді клас `Object`. А при обмеженні знизу `Object` і є граничним єдиним обмеженням.

Для узагальнених типів існують такі обмеження:

- аргументи типу не можуть бути примітивними типами;
- не можна створити об'єкти параметрів типу;
- не можна оголошувати поля з параметрами типів як типів як статичні;
- не можна використовувати приведення типів і оператор `instanceof` з узагальненими типами;
- не можна створювати масиви об'єктів узагальнених типів;
- не можна створювати, перехоплювати і викидати об'єкти узагальнених типів як виключення;
- не можна перевантажувати методи з узагальненими типами в якості аргументів, тому що при стирання типів сигнатура може виявитися однаковою.

Загальна характеристика фреймворка колекцій в Java

При написанні програми дуже часто виникає потреба зберігати набір будь-яких об'єктів в універсальних контейнерах, які перевіряють тип об'єкта при приміщенні в контейнер і вилученні з нього. Масиви (виконують таке завдання) не підходять для випадків, коли заздалегідь невідома точна кількість елементів. Колекції в Java можна уявити як "динамічні масиви", які автоматично збільшують свій розмір при розміщенні елементів і зменшують при вилученні. *Java Collection Framework* (*JCF*) являє собою ієрархію інтерфейсів і їх реалізацій, яка є частиною JDK і дозволяє розробнику користуватися великою кількістю зручних структур даних і алгоритмів роботи з ними. На вершині ієрархії в JCF розташовуються два інтерфейси: `Collection<E>` і `Map<E>` (Рис. 2). Ці інтерфейси

поділяють всі колекції за типом зберігання даних на дві частини: послідовні набори елементів і набори пар ключ-значення (словники).

Засоби по роботі з динамічними масивами об'єктів (колекціями) з'явилися в Java ще у версії JDK 1.0 (23.01.1996) - класи java.util.Vector і java.util.HashTable. В JDK 1.2 (1998 рік) Джошуа Блох розробив архітектуру і основні реалізації фреймворка колекцій. В JDK 5 (2004 рік) фреймворк був переписаний для введення узагальнень (generics). В JDK 6 (2006 рік) додані інтерфейси `Deque<E>`, `NavigableSet<E>`, `NavigableMap<E>`. В JDK 8 (2014 рік) фреймворк був адаптований до використання лямбда-виразів, потоків і потокових операцій. В JDK 9 (2017 рік) додані зручні фабричні методи для створення екземплярів колекцій.

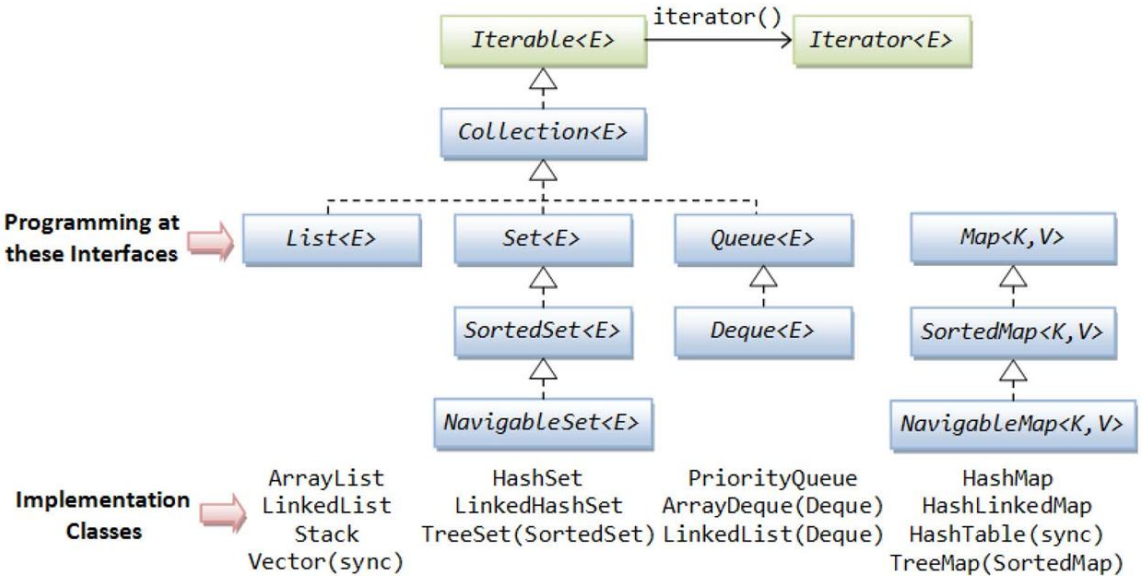


Рис. 2. Ієрархія інтерфейсів Java Collection Framework та їх популярних реалізацій

Інтерфейс `Collection<E>` містить абстрактні методи, загальні для всіх колекцій, такі, як додавання та видалення елементів, очищення колекції, отримання ітератора для колекції (об'єкта, що організує обхід колекції і отримання її елементів) і створення масиву з елементів колекції. Використовується, коли необхідно обробляти елементи колекцій різних типів.

Інтерфейс `Collection<E>` розширює інтерфейс `Iterable<E>`, що дозволяє елементам колекції бути «перебраними» і містить абстрактний метод `iterator()`, який повертає об'єкт-ітератор, що реалізує інтерфейс `Iterator<E>` з методами `hasNext()` і `next()` (будуть розглянуті далі).

Інтерфейс `List<E>` описує колекцію упорядкованих елементів (наприклад, по черговості додавання в колекцію), доступ до яких здійснюється за індексом (позиції в колекції). Реалізує математичну абстракцію списку, може зберігати однакові елементи. Використовується, коли потрібен певний порядок зберігання елементів з можливістю вставки і видалення елементів за індексом.

Інтерфейс `Set<E>` – це колекція, що не дозволяє дублювання елементів, реалізує математичну абстракцію множини. Використовується для зберігання унікальних елементів, не гарантує зберігання елементів у черговості додавання.

Інтерфейс `Queue<E>` – також, як і `List<E>` – колекція упорядкованих елементів, але доступ до них можливий тільки з початку та/або з кінця колекції. Дозволяє підтримувати правила черг "перший зайшов – перший вийшов" (FIFO) та "останній зайшов – перший вийшов" (LIFO).

Інтерфейс `Map<E>` – це колекція наборів пар "ключ-значення", при цьому ключі є унікальними. Використовується для зберігання складних даних, пошук і отримання яких здійснюється за ключами. Не є прямим спадкоємцем `Collection<E>`.

Інтерфейси `Collection<E>` і `Iterator<E>`

Методи інтерфейсу `Collection<E>` можуть бути умовно поділені на: основні методи, методи групової обробки і методи перетворення до масиву. У наведеному нижче фрагменті коду демонструється використання методів інтерфейсу `Collection<E>` на прикладі реалізації-списку `ArrayList<String>`:

```
Collection<String> myColl = new ArrayList<>();
myColl.add("aaa");
myColl.add("bbbb");
myColl.add("ccccc");
System.out.println(myColl); //[aaa, bbbb, ccccc]
System.out.println("Collection myColl is empty:"
                   + MyColl.isEmpty()); // false
System.out.println("Size of the myColl is:"
                   + MyColl.size()); // 3
System.out.println("myColl contains element \"aaa\"is:"
                   + MyColl.contains("aaa")); // true
/* Створення ідентичною колекції */
Collection<String> anotherColl = new ArrayList<>(myColl);
System.out.println("myColl and anotherColl refer same object:"
                  + (MyColl == anotherColl)); // false
System.out.println("myColl and anotherColl are equal:"
                   + MyColl.equals(anotherColl)); // true
System.out.println("Видалення наявного елемента:"
                   + MyColl.remove("aaa")); // true
System.out.println("Видалення відсутнього елемента:"
                  + MyColl.remove("zzz")); // false
/* При використанні методу toArray () цільової масив може
   бути тільки типу Object */
Object[] MyArrObj = myColl.toArray();
System.out.println("MyArrObj:"
                   + Arrays.toString(MyArrObj)); //[bbbb, ccccc]
/* При необхідності створення масиву конкретного типу слід
  використовувати метод toArray(T[] a) */
String[] strings = new String[2];
String[] MyArrStr = myColl.toArray(strings); //[bbbb, ccccc]
System.out.println("MyArrStr:"
                   + Arrays.toString(MyArrStr));
System.out.println("strings:"
                   + Arrays.toString(strings)); //[bbbb, ccccc]
/* Групові операції */
```

```
myColl = new ArrayList<>(Arrays.asList("aaa"));
anotherColl = new ArrayList<>(Arrays.asList("bbbb", "ccccc"));
myColl.addAll(anotherColl);
System.out.println(myColl); //[aaa, bbbb, ccccc]
System.out.println("myColl contains all elements of AnotherColl "
                  + "is: MyColl.containsAll(anotherColl)); // true
myColl.retainAll(anotherColl));
System.out.println(myColl); //[bbbb, ccccc]
myColl.add("aaa");
System.out.println(myColl); //[bbbb, ccccc, aaa]
myColl.removeAll(anotherColl));
System.out.println(myColl); //[aaa]
anotherColl.clear();
System.out.println("Collection anotherColl is empty:"
                   + AnotherColl.isEmpty()); // true
```

Оголошення `myColl` як об'єкта інтерфейсу `Collection<E>` дозволить при необхідності перетворити колекцію з одного типу (`ArrayList<E>` у даному випадку) в будь-який інший. Однак слід врахувати, що набір методів для об'єкта типу `Collection<E>` буде обмежуватися тільки методами інтерфейсу `Collection<E>`, а не класу, використаного у конструкторі при створенні об'єкта (в наведеному прикладі `ArrayList<E>`). Метод `add(E element)` повертає `true` у випадку, якщо колекція змінюється і `false` – якщо залишається незмінною (деякі реалізації не дозволяють повторне додавання одних і тих же об'єктів).

Метод `remove(E element)` повертає `true` у випадку, якщо колекція змінюється (видаляється елемент колекції) і `false` – якщо залишається незмінною (елемента, зазначеного для видалення в колекції немає).

Метод `addAll(Collection<? Extends E> c)` додає до поточної колекції елементи колекції-параметра (реалізує об'єднання колекцій). Метод `retainAll(Collection<?> c)` залишає у поточній колекції тільки елементи, наявні в колекції-параметрі (реалізує перетин колекцій).

При використанні методу `toArray()` цільової масив може бути тільки типу `Object`, а приведення його до якогось типу викликає виключення java.lang.ClassCastException. Тому для отримання з колекції масиву певного типу (посилального), слід використовувати метод `<T> T[] toArray(T[] a)`, який заповнює масив-параметр `T[] a` елементами колекції, якщо розмір цього масиву достатній для розміщення всіх елементів колекції, в іншому випадку масив заповнюється посиланнями `null`. При надмірному розмірі масиву-параметра в ньому зберігаються елементи колекції, доповнені посиланнями `null` для надлишкових позицій.

Для колекції будь-якого типу (окрім реалізацій `Map<E>` за допомогою методу `iterator()` може бути отриманий об'єкт-реалізація інтерфейсу `Iterator<E>`, що дозволяє виконувати обхід колекції з отриманням або видаленням її елементів.

```
Collection<String> myColl =
```

```
            new ArrayList<>(Arrays.asList("aa", "bbb", "cccc"));
/* Отримуємо ітератор колекції */
Iterator<String> iterator = myColl.iterator();
/* Виконуємо обхід колекції за допомогою ітератора */
while (iterator.hasNext()) {
    /* Витягує елемент колекції, на який вказує
       ітератор, і перевстановлює ітератор на наступний
       елемент */
    String element = iterator.next();
    System.out.print(element + " "); // aa bbb cccc
}
System.out.println();
iterator = myColl.iterator();
while (iterator.hasNext()) {
    iterator.next();
    /* Видаляє елемент колекції, на який вказує
       ітератор після останнього отримання методом next() */
    iterator.remove();
}
System.out.println("myColl:" + myColl); //[]
```

При отриманні з колекції методом `iterator()` об'єкта-ітератора, він вказує на перший елемент колекції. Викликом з ітератора методу `next()` отримуємо елемент колекції, на який вказує ітератор, після чого ітератор вказує на наступний елемент. Метод ітератора `hasNext()` служить для переходу до наступного елемента при обході колекції та для виходу з циклу при досягненні останнього елемента. Якщо спробувати його проігнорувати і викликати вручну метод `next()` після вилучення останнього елемента, виникне помилка java.util.NoSuchElementException. У будь-який момент обходу колекції ітератор може бути отриманий повторно, при цьому він знову буде вказувати на перший елемент колекції. Якщо відразу ж після отримання ітератора без виклику з нього методу `next()` викликати метод `remove()`, виникає помилка java.lang.IllegalStateException.

Інтерфейс `List<E>` та його реалізації

Інтерфейс `List<E>` представляє колекцію упорядкованих елементів (наприклад, по черговості додавання до колекції), доступ до яких здійснюється за індексом (за позицією у колекції). Реалізує математичну абстракцію списку, може зберігати однакові елементи. Використовується, коли потрібен певний порядок зберігання елементів з можливістю вставки та видалення елементів за індексом. Найбільш часто використовувані реалізації: `ArrayList<E>`, `LinkedList<E>`, `Vector<E>`, `Stack<E>` (останні дві підтримують багатопотокове використання) (Рис. 3).

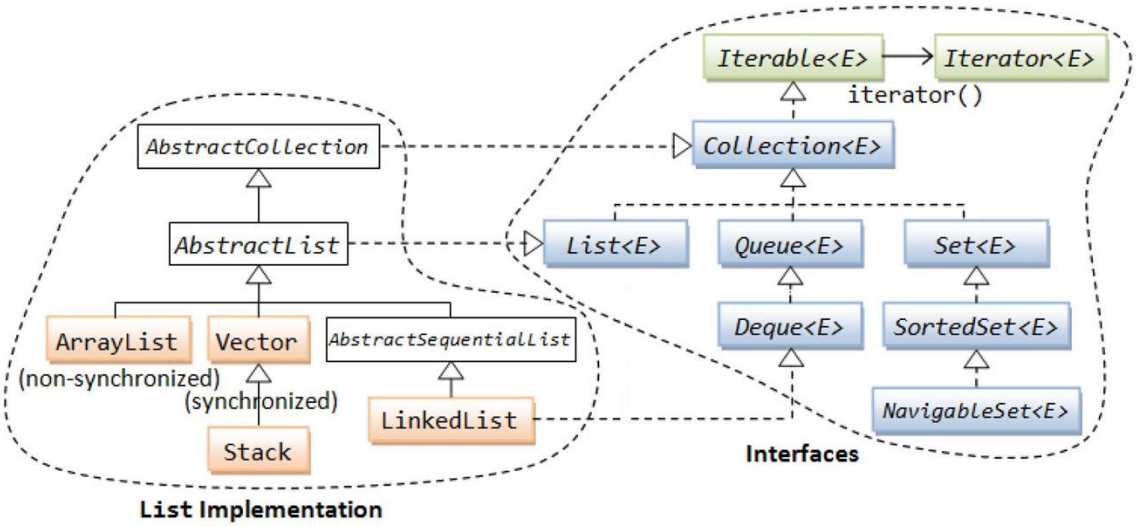


Рис. 3. Реалізації інтерфейсу List<E>

На додаток до методів, успадкованих від `Collection<E>`, інтерфейс `List<E>` включає наступні методи:

- позиційного доступу – заснованого на позиції елемента у списку (індексі):
  ```
  void add(int index, E element)
  boolean addAll(int index, Collection<? extends E> c)
  E get(int index)
  E remove(int index)
  E set(int index, E element)
  ```
- пошуку - виконують пошук зазначеного елемента в списку і повертають його індекс:
  ```
  int indexOf(Object o)
  int lastIndexOf(Object o)
  ```
- ітерації - розширюють семантику ітератора для отримання переваг послідовного розташування елементів:
  ```
  ListIterator<E> listIterator()
  ListIterator<E> listIterator(int index)
  ```
- виділяє піддіапазони елементів списку:
  ```
  List<E> subList(int fromIndex, int toIndex)
  ```
- сортування з використанням компаратора:
  ```
  void sort(Comparator<? super E> c)
  ```
- заміни елементів відповідно до результату роботи реалізації функціонального інтерфейсу `UnaryOperator<E>`:
  ```
  void replaceAll(UnaryOperator<E> operator)
  ```

Наступний фрагмент коду демонструє використання методів інтерфейсу `List<String>` на прикладі реалізації `ArrayList<String>`:

```
List<String> list =
        new ArrayList<>(Arrays.asList("aaa", "bbbb", "ccccc"));
/* Демонстрація роботи методів позиційного доступу */
list.add(1, "a1a1a1");
System.out.println(list); //[aaa, a1a1a1, bbbb, ccccc]
/* Додавання (вставка) колекції за вказаною індексу */
List<String> anotherList =
```

```
                    new ArrayList<>(Arrays.asList("eeee", "fff"));
list.addAll(4, anotherList);
System.out.println(list); //[aaa, a1a1a1, bbbb, ccccc, eeee,
                          // fff]
String s1 = list.get(3);
System.out.println(s1); // ccccc
list.remove(1);
System.out.println(list); //[aaa, bbbb, ccccc, eeee, fff]
list.set(3, "bbbb");
System.out.println(list); //[aaa, bbbb, ccccc, bbbb, fff]
/* Демонстрація роботи методів пошуку */
int pos1 = list.indexOf("bbbb");
System.out.println(pos1); // 1
int pos2 = list.lastIndexOf("bbbb");
System.out.println(pos2); // 3
/* Демонстрація методу отримання частини списку*/
List<String> sublist = list.subList (1, 3);
System.out.println(sublist); //[bbbb, ccccc]
/* Сортування списку у природному порядку - за алфавітом
(Елементи списку повинні реалізовувати інтерфейс Comparable) */
list.sort (null);
System.out.println(list); //[aaa, bbbb, bbbb, ccccc, fff]
/* Сортування списку за ознакою, визначеною компаратором */
list.sort (new StringLengthComparator ());
System.out.println(list); //[aaa, fff, bbbb, bbbb, ccccc]
/*Обхід списку у зворотному порядку за допомогою listIterator*/
ListIterator<String> listIterator =
                                list.listIterator(list.size());
while (listIterator.hasPrevious()) {
    System.out.print(listIterator.previous()
                     + ""); // 4eeee4 4bbbb4 ccccc fff aaa
}
/* Компаратор за довжиною рядків */
static class StringLengthComparator
                                 implements Comparator<String> {
    @Override
    public int compare String o1, String o2) {
        return o1.length() - o2.length();
    }
}
```

Як видно з прикладу, індекс списку аналогічний індексу масиву. Метод додавання за індексом відпрацьовує при додаванні за наступним за останнім індексом, однак при спробі додавання за межі масиву, виникає помилка java.lang.IndexOutOfBoundsException. Методи інтерфейсу `Collection<E>`, що виконують додавання і видалення елементів (без індексу), додають елемент в кінець і видаляють – з початку списку. Слід зазначити, що при вставці елементів в початок і середину вже заповненого списку відбувається копіювання частини масиву, що зберігає елементи списку (Рис. 4).

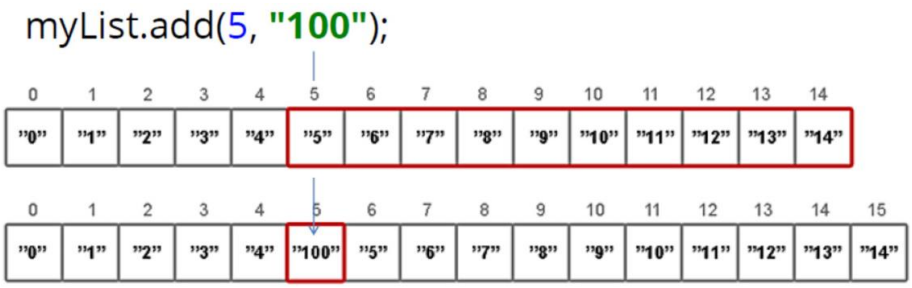


Рис. 4. Копіювання частини масиву при вставці елементів в ArrayList<E>

Методи пошуку працюють аналогічно однойменним методам класу java.lang.String. Аргументи методу `List<E> subList(int fromIndex, int toIndex)` вказують, з якого індексу включно починається частина списку к і до якого індексу (не включаючи його) – простягається. Сортування елементів списку, що є об'єктами класів, які реалізують інтерфейс java.lang.Comparable<E>, використовує правило сортування, зазначене в компараторі.

Об'єкт `ListIterator<E>` розширює функціональність звичайного ітератора, доступного для всіх колекцій. Зокрема, він додатково дозволяє обхід списку у зворотному напрямку, а також додавання і заміну елементів списку.

У попередніх прикладах для демонстрації функціоналу колекцій і списків ми використовували найбільш популярну реалізацію списку – клас java.util.ArrayList<E>, що використовує для зберігання елементів списку масив, розмір якого автоматично "підбирається" під необхідну кількість елементів. За допомогою методу `void ensureCapacity(int minCapacity)` цього класу можна вручну задавати розмір масиву (це запобігає операції копіювання масиву для завдання більшого розміру і підвищує продуктивність виконання програми). Також за допомогою методу `void trimToSize()` можлива підгонка розміру масиву під фактичну кількість елементів, що зберігаються в ньому.

Ще однією популярною реалізацією інтерфейсу `List<E>` є клас java.util.LinkedList<E>, який зберігає свої елементи у структурі даних *двобічно зв'язаний список*, у якому кожен елемент містить покажчики на попередній і наступний елементи. Ітератор підтримує обхід в обидві сторони. Реалізує методи отримання, видалення і вставки в початок, середину і кінець списку. Дозволяє додавати будь-які елементи, в тому числі і `null`.

Об'єкт `LinkedList<E>` організований з об'єктів-вузлів `Node<E>`, що зберігають посилання об'єкт, який в них міститься, а також посилання на наступний і попередній вузли списку. Об'єкт `header` - це псевдо-елемент списку, його значення завжди дорівнює `null`, а властивості `next` і `prev` завжди вказують на перший та останній елемент списку, відповідно. Для порожнього списку властивості `next` і `prev` вказують самі на себе (тобто на елемент `header`). Кожен раз при додаванні нового елемента до списку створюється новий екземпляр класу `Node<E>` і перевизначаються покажчики на попередній і наступний елементи (Рис. 5).

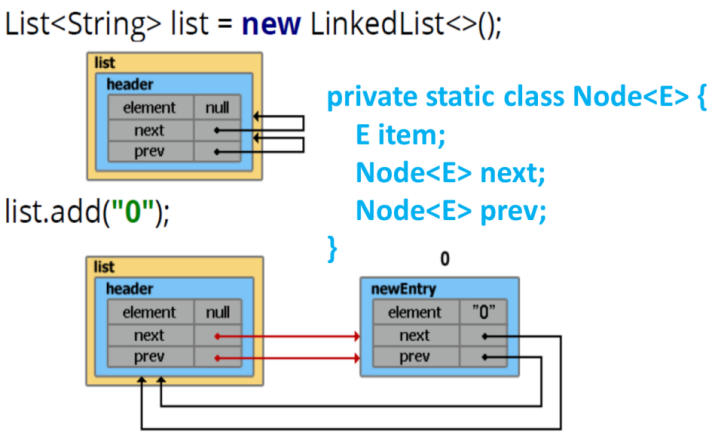


Рис. 5. Додавання елемента в LinkedList<E>

Очевидно, що при вставці і видаленні елементів в `LinkedList<E>`, тільки перевизначаються значення посилань `next` і `prev` сусідніх елементів, що має бути швидше, ніж копіювання частини масиву в `ArrayList<E>`, особливо, якщо вставка відбувається до початку вже заповненого списку. Ця особливість `LinkedList<E>` пояснює його застосування у випадках, коли необхідно забезпечувати вставку і видалення елементів з початку і з кінця списку. Ці операції є типовими для структур даних "черга" і "стек", тому додатково до методів інтерфейсу `List<E>` клас `LinkedList<E>` реалізує методи інтерфейсу `Deque<E>` і його батьківського класу `Queue<E>`, що забезпечують такі операції (див. Рис. 3). Оскільки різнойменних методів, які виконують однакові операції, багато, в наведеній нижче таблиці вказуються їх особливості.

Табл. 1. Особливості реалізацій методів інтерфейсів `Deque<E>` і `Queue<E>` класом `LinkedList<E>` (фіолетовим кольором виділені методи `Queue<E>`)

<table>
<tr><th></th><th>Додавання елемента до початку списку</th><th>Додавання елемента в кінець списку</th><th>Отримання елемента з початку списку</th><th>Отримання елемента з кінця списку</th></tr>
<tr><td rowspan="8">Елемент залишається у списку</td><td colspan="2"></td><td colspan="2">повертають для порожнього списку
<code>NoSuchElementException</code></td></tr>
<tr><td><code>void addFirst(E e)</code></td><td><code>void addLast(E e)</code></td><td><code>E element()</code>
<code>E getFirst()</code></td><td><code>E getLast()</code></td></tr>
<tr><td colspan="2"></td><td colspan="2">повертають <code>null</code> для порожнього списку</td></tr>
<tr><td><code>boolean offerFirst(E e)</code>
<code>void push(E e)</code></td><td><code>boolean offer(E e)</code>
<code>boolean offerLast(E e)</code></td><td><code>E peek()</code>
<code>E peekFirst()</code></td><td><code>E peekLast()</code></td></tr>
<tr><td rowspan="8">Об'єкт видалений зі списку</td><td colspan="2" rowspan="8"></td><td colspan="2">повертають для порожнього списку
<code>NoSuchElementException</code></td></tr>
<tr><td><code>E removeFirst()</code>
<code>E pop()</code></td><td><code>E removeLast()</code></td></tr>
<tr><td colspan="2">повертають null для порожнього списку</td></tr>
<tr><td>E poll()
E pollFirst()</td><td>E pollLast()</td></tr>
</table>

У наступному фрагменті коду наводяться приклади реалізації черги і стека за допомогою LinkedList<E>:

```
Deque<Integer> queue = new LinkedList<>();
/*Реалізація черзі з дисципліною доступу до елементів
  "Перший прийшов - першим вийшов" (FIFO, First In-First Out)*/
int i = 0;
while (i < 5) {
    queue.addFirst(i);
    i ++;
}
System.out.println(queue); //[4, 3, 2, 1, 0]
while (! queue.isEmpty()) {
    System.out.print(queue.removeLast() + " "); // 0 1 2 3 4
}
System.out.println("\n" + queue); //[]

/*Реалізація стека з дисципліною доступу до елементів "останній
  прийшов - першим вийшов "(LIFO, Last In - First Out)*/
i = 0;
while (i<5) {
    queue.push(i);
    i ++;
}
System.out.println(queue); //[4, 3, 2, 1, 0]
while (! queue.isEmpty()) {
    System.out.print(queue.pop() + " "); // 4 3 2 1 0
}
System.out.println("\n" + queue); //[]
```

Інтерфейс `Set<E>` та його реалізації

Інтерфейс Set<E> дозволяє організовувати структури даних в інформатиці, які є реалізаціями математичної множини. На відміну від списків, множина зберігає елементи-об'єкти певного типу без певного порядку. Повторення елементів множини не допускається.

Найбільш популярною реалізацією інтерфейсу Set<E> є клас java.util.HashSet<E>, який є найбільш популярною реалізацією. HashSet<E> використовує для зберігання елементів поле:

```
private transient HashMap<E, Object> map
```

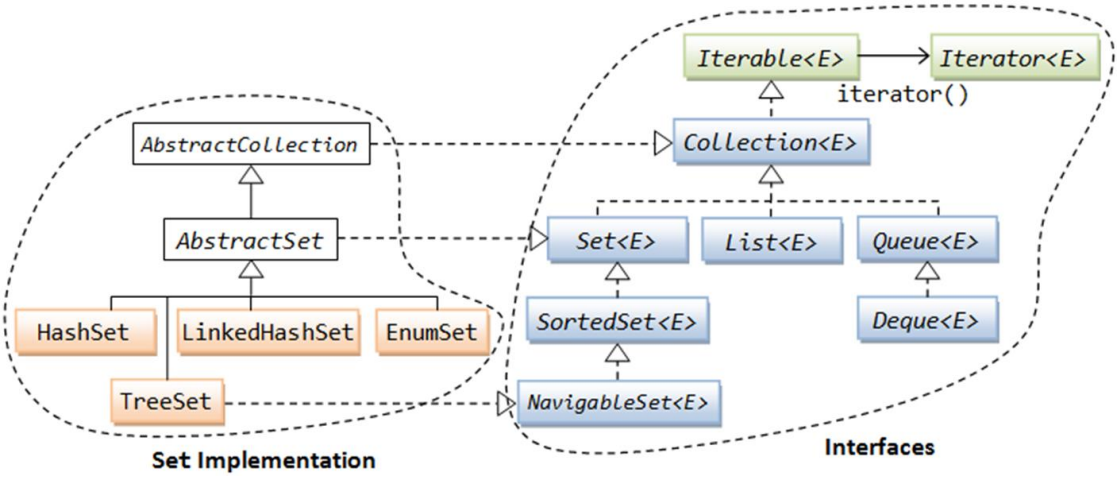


Рис. 6. Реалізації інтерфейсу Set<E>

При розміщенні елемента в `HashSet<E>`, він фактично міститься в java.util.HashMap<E, Object>:

```
public boolean add(E e) {
    return map.put(e, PRESENT) == null;
}
```

зі значенням

```
private static final Object PRESENT = new Object();
```

Оскільки ключі колекції `HashMap<E, Object>` зберігаються в `HashSet<E>` (опис колекції `HashMap<K, V>` наведено далі), то при доданні елементів до колекції їх ключі розміщуються в масиві елементів колекції `HashSet<E>` за індексами, що залежать від хеш-коду елемента-об'єкта, тобто без порядку зберігання. Якщо порядок зберігання важливий, використовують множину `TreeSet<E>`, у якій елементи зберігаються відсортованими по зростанню у природному порядку або порядку, визначеному відповідним компаратором. Клас `LinkedHashSet<E>` дозволяє створювати множину зі зберіганням об'єктів-елементів у порядку їх додавання до множини. У наведеному фрагменті коду створюються екземпляри множин описаних вище типів, і демонструється порядок зберігання в них елементів після додавання:

```
Set<String> mySet =
          new HashSet<>(Arrays.asList("aaa", "bbbb", "ccccc"));
System.out.println(mySet); //[aaa, ccccc, bbbb]
/* Дублювання елементів запобігається */
mySet.add("aaa");
System.out.println(mySet); //[aaa, ccccc, bbbb]
mySet = new TreeSet<>(Arrays.asList("aaa", "bbbb", "ccccc"));
System.out.println(mySet); //[aaa, bbbb, ccccc]
mySet =
    new LinkedHashSet<>(Arrays.asList("ccccc", "bbbb", "aaa"));
System.out.println(mySet); //[ccccc, bbbb, aaa]
```

Слід зазначити, що для користувацьких класів, об'єкти яких розміщуються в `HashSet<E>` повинні бути перевизначені методи `int hashCode()` та `boolean`

`equals(Object obj)` класу java.lang.Object. Пояснимо на прикладі. Нехай необхідно додати в `HashSet<E>` об'єкти наступного класу:

```
public class Student {
    String name;
    int age;

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

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

таким кодом:

```
Set<Student> students = new HashSet<>();
students.add(new Student("Ivan", 20));
students.add(new Student("Mike", 22));
students.add(new Student("Mike", 25));
students.add(new Student("Ivan", 20));
for (Student student: students) {
    System.out.print(student + "-" + student.hashCode() + "");
   }
```

Хоча, як зазначено вище, HashSet<Student> мав би запобігти приміщення в колекцію дублюючого елемента, цього не відбувається, висновок на консоль:

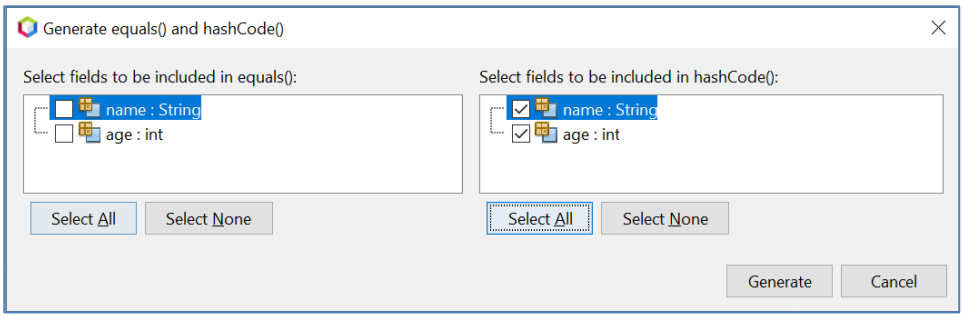
```
Student {name = Mike, age = 25}-2018699554 Student {name = Ivan, age = 20}-366712642 Student {name = Mike, age = 22}-1829164700 Student {name = Ivan, age = 20}-1311053135
```

Оскільки реалізація методу `int hashCode()` за замовчуванням для кожного об'єкта обчислює незалежне від властивостей об'єкта випадкове ціле значення, для `HashSet<Student>` всі об'єкти є різними, тому що мають різний хеш-код.

Додамо до класу `Student` перевизначення методу `int hashCode()`. В IDE NetBeans для цього можна викликати команду *Source-Insert Code...* (*Alt + Insert*) і в меню вибрати команду equals() і hashCode(). Відкриється вікно *Generate equals() and hashCode()*, у якому для початку зазначимо тільки властивості класу, які повинен враховувати метод `int hashCode()` (Мал. 7).

В клас буде доданий метод:

```
@Override
public int hashCode() {
    int hash = 3;
    hash = 19 * hash + Objects.hashCode(this.name);
    hash = 19 * hash + this.age;
    return hash;
}
```



Мал. 7. Генерування методів equals() і hashCode() в IDE NetBeans

Повторний запуск програми забезпечує виведення:

```
Student {name = Mike, age = 22}-45568919 Student {name = Ivan, age = 20}-43536449 Student {name = Ivan, age = 20}-43536449 Student {name = Mike, age = 25}-45568922
```

Тепер хеш-код однакових об'єктів однаковий, проте все одно виводяться дублі. Це відбувається тому, що не перевизначений метод `boolean equals(Object obj)` класу java.lang.Object, який в реалізації за замовчуванням перевіряє рівність адрес об'єктів в пам'яті (і тому вважає два однакових об'єкта різними):

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

За допомогою команди *Source-Insert Code...* (*Alt + Insert*) можна сгенерувати для класу `Student` додатково метод `boolean equals(Object obj)`, який тепер враховує властивості об'єкта класу `Student`:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass ()! = obj.getClass()) {
        return false;
    }
    final Student other = (Student) obj;
    if (this.age! = other.age) {
        return false;
    }
    if (! Objects.equals(this.name, other.name)) {
        return false;
    }
    return true;
}
```

Тепер запуск програми покаже тільки три елементи у колекції, функціональність запобігання дублів підтримується:

```
Student {name = Mike, age = 22}-45568919 Student {name = Ivan, age = 20}-43536449 Student {name = Mike, age = 25}-45568922
```

Таким чином, у разі розміщення об'єктів користувацьких класів у `HashSet<E>` необхідно в таких класах перевизначати методи `int hashCode()` та `boolean equals(Object obj)` класу java.lang.Object.

Аналогічним чином, для об'єктів користувацьких класів, які розміщені у `TreeSet<E>`, необхідно реалізовувати метод порівняння інтерфейсу java.lang.Comparable<E> або інтерфейсу java.util.Comparator<E>. Наприклад, для класу `Student implements Comparable<Student>` з порівнянням за властивістю `name`, а потім по властивістю `age` метод буде виглядати:

```
@Override
public int compareTo(Student o) {
    int i = this.name.compareTo(o.name);
    if (i! = 0) {
        return i;
    }
    return this.age - o.age;
}
```

І виведення до консолі раніше створених об'єктів:

```
students = new TreeSet<>();
students.add(ivan1);
students.add(mike1);
students.add(mike2);
students.add(ivan2);
System.out.println(students);
```

буде впорядкованим:

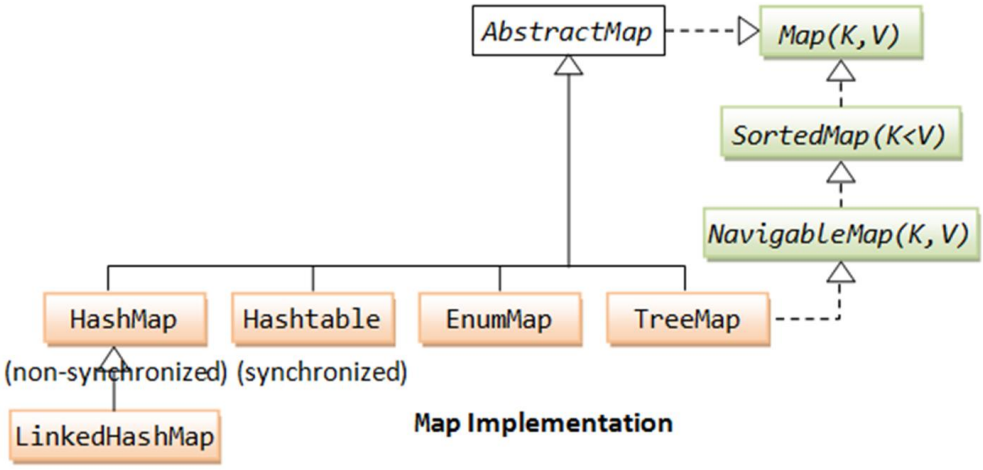
```
[Student {name = Ivan, age = 20}, Student {name = Mike, age = 22}, Student {name = Mike, age = 25}]
```

Спроба додати в `TreeSet<E>` користувацький клас з невизначеним методом порівняння призводить до виникнення помилки java.lang.ClassCastException.

Інтерфейс `Map<K, V>` та його реалізації

`Map<K, V>` називають *відображенням* (*асоціативним масивом, словником, картою*) – це абстрактний тип даних, що дозволяє зберігати пари "ключ-значення" і підтримує операції додавання пари, а також пошуку і видалення пари за ключем. Передбачається, що відображення не може зберігати дві пари з однаковими ключами. `Інтерфейс Map<K, V>` не є спадкоємцем інтерфейсу `Collection<E>` (Рис. 2).

Стандартний JDK включає ряд реалізацій цього інтерфейсу: `HashMap<K, V>`, `LinkedHashMap<K, V>`, `Hashtable<K, V>`, `EnumMap<K extends Enum<K>, V>`, `TreeMap<K, V>` та інші (Мал. 8).



Мал. 8. Реалізації інтерфейсу Map<K, V>

Найбільш популярною реалізацією інтерфейсу `Map<K, V>` є клас `HashMap<K, V>`. Пари "ключ-значення" у цьому класі зберігаються в елементах двубічно зв'язаних списків `Node<K, V>`, які містяться в масиві `Node<K, V>[] table`. Елементи зв'язного списку визначаються статичним вкладеним класом:

```
static class Node<K, V> implements Map.Entry<K, V> {
    final int hash; \\ хеш-код ключа
    final K key; \\ ключ
    V value; \\ значення
    Node<K, V> next; \\ двубічно зв'язаний елемент списку
...}
```

Для кожного ключа розраховується хеш-код і для його унікального значення виділяється елемент масиву, до якого вноситься відповідний об'єкт `Node<K, V>`. У разі колізії хеш-коду ключа, відповідна пара "ключ-значення" розміщується у наступному елементі зв'язного списку, розташованого за тим ж індексом масиву.

Крім зазначеного масиву, `HashMap<K, V>` характеризується такими властивостями:

`size` – кількість збережених в даний момент пар "ключ-значення";

`capacity` – максимальна можлива кількість доданих пар "ключ-значення" до перебудови масиву (початкове значення за замовчуванням `capacity = 16`, максимальне – приблизно `1 млрд` (`1073741824`));

`loadFactor` – коефіцієнт заповнення хеш-таблиці – число збережених пар "ключ-значення", поділене на розмір масиву `H` (число можливих значень хеш-функції), є важливим параметром, від якого залежить середній час виконання операцій. Значення за замовчуванням `0,75` є хорошим компромісом між часом доступу і обсягом даних, що зберігаються;

`threshold` – гранична кількість елементів, при досягненні якої розмір хеш-таблиці збільшується вдвічі, розраховується за формулою `capacity * loadFactor` (за замовчуванням `16 * 0,75 = 12`);

Існують варіанти конструктора з `capacity` за замовчуванням і з `capacity` і `threshold` в якості параметрів:

```
Map<String, String> hashmap1 = new HashMap<String, String>(20);
Map<String, String> hashmap2 =
                          new HashMap<String, String>(20, 0.8f);
```

При додаванні елемента, послідовність кроків наступна:

1. Перевіряється, чи не перевищить додавання нового елемента значення `threshold`. Якщо перевищує, то `capacity` збільшується та елемент додається до хешмепу.

2. При додаванні нової пари "ключ-значення" обчислюється хеш-код ключа з використанням методу `int hashCode()` класу java.lang.Object:

```
static final int hash(Object key) {
    int h;
    return (key == null)? 0
                         : (H = key.hashCode()) ^ (h >>> 16);
}
```

3. За хеш-кодом ключа знаходиться індекс елемента масиву записів, у який розміщується нове значення: `h & (length - 1)`, де `length` – довжина масиву на момент додавання елемента, наприклад, при `h = 51` і `length = 16` індекс масиву дорівнює `3` (Рис. 9).

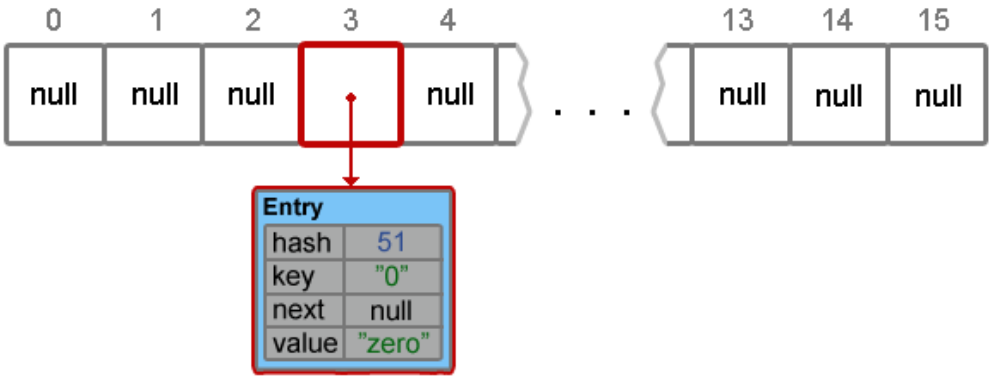


Рис. 9. Додавання пари "ключ-значення" до HashMap<String, String>

4. Якщо в цьому елементі вже розташований об’єкт, то існує два варіанти:

- якщо збігаються ключі елемента, що зберігається, та елемента, що додається, тоді змінюється значення для збереженого елемента на значення елемента, що додається;
- якщо ключі елемента, що зберігається, та елемента, що додається, не збігаються, (наприклад, `h & (length - 1)` для `h = 101603` і `length = 16` також дасть індекс масиву `3`), тоді елементом цього масиву в початок зв'язаного списку встановлюється елемент `Node<K, V>`, у якого поле `next`, посилається на елемент, який перебував тут раніше (Рис. 10).

5. У разі додавання пари "ключ-значення" зі значенням ключа `null`, обчислення хеш-коду не відбувається, і елемент додається завжди за індексом масиву `= 0` (Рис. 10).

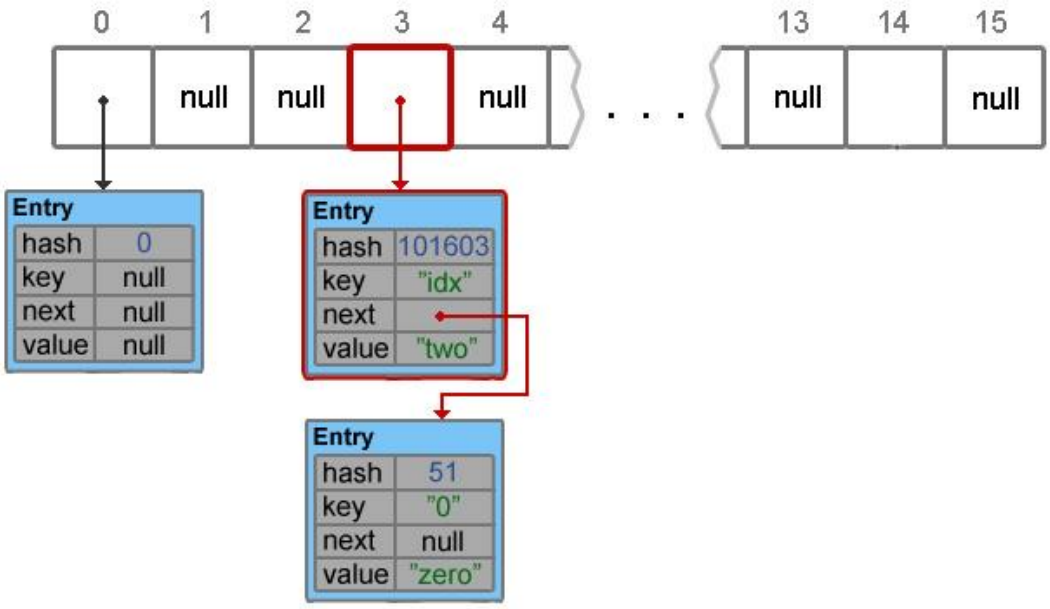


Рис. 10. Додавання пари "ключ-значення" в HashMap<String, String> при колізії хеш-коду ключа

Нижче наведено фрагмент коду, який ілюструє використання `HashMap<K, V>`. Перевагою цієї реалізації інтерфейсу `HashMap<K, V>` є константне час доступу до збережених значень.

```
Map<String, Integer> hm = new HashMap<>();
/*Додавання пар ключ-значення, метод повертає значення,
  що додається */
hm.put("one", 1);
hm.put("two", 2);
hm.put("three", 3);

/* Повторно вже наявна пара ключ-значення не додається */
hm.put("one", 1);
System.out.println(hm); // {one = 1, two = 2, three = 3}
/* При додаванні нового значення з вже наявним ключем,
  значення елемента замінюється новим */
hm.put("one", 4);
System.out.println(hm); // {one = 4, two = 2, three = 3}
/* У разі якщо ключ, щододається, вже є, додавання
  не виконується, метод повертає значення, що додається */
hm.putIfAbsent("one", 1);
System.out.println(hm); // {one = 4, two = 2, three = 3}
/* Отримання значення по ключу */
int x = hm.get("two");
System.out.println(x); // 2
/* Перевірка наявності ключа і значення */
System.out.println(hm.containsKey("one")); // true
System.out.println(hm.containsValue(2)); // true
/* Друк ключів - метод keySet() інтерфейсу Map,
  реалізований в AbstractMap повертає множину з ключами */
for (String key: hm.keySet()) {
    System.out.print(key + " "); // one two three
```

```
}
System.out.println();
/* Заміна значення по ключу */
System.out.println(hm.replace ("two", 22)); // 2
/* Друк значень - метод values () інтерфейсу Map, реалізований
   у AbstractMap, повертає колекцію зі значеннями */
for (int value: hm.values()) {
    System.out.print(value + " "); // 4 22 3
}
System.out.println();
/* Видалення пари ключ-значення */
System.out.println(hm.remove("two")); // 22
/*Друк всіх ключів і значень - метод entrySet() інтерфейсу Map,
  реалізований в HashMap, повертає множину Set<Map.Entry<K, V >>*/
for (Entry<String, Integer> entry: hm.entrySet()) {
    System.out.print(entry + " "); // one = 4 three = 3
}
System.out.println();

/* Друк всіх ключів і значень через ітератор */
Iterator<Map.Entry<String, Integer>> iterator =
                                        hm.entrySet().iterator();
while (iterator.hasNext()) {
    Map.Entry<String, Integer> entry = iterator.next();
    /* Key = one, Value = 4 Key = three, Value = 3 */
    System.out.println("Key =" + entry.getKey () + ", Value ="
                      + Entry.getValue());
}
```

Реалізація `TreeMap<K, V>` зберігає впорядковані за деякими властивостями ключа пари "ключ-значення", при цьому для об'єктів класів, що реалізують інтерфейс java.lang.Comparable<E>, впорядкування відбувається у натуральному порядку, а для об'єктів інших класів – в порядку, передбаченими відповідним компаратором . Нижче наводиться фрагмент коду, який ілюструє особливості `TreeMap<K, V>`:

```
   /* Порядок елементів - натуральний - за алфавітом */
   Map<String, Integer> tm = new TreeMap<>();
   tm.put("aaa", 3);
   tm.put("aa", 2);
   tm.put("b", 1);
   System.out.println(tm); // {aa = 2, aaa = 3, b = 1}
   /* Порядок елементів - за кількістю символів у ключі */
   Map<String, Integer> tmsl =
                        new TreeMap<>(new StrLengthComparator());
   tmsl.put("aaa", 3);
   tmsl.put("aa", 2);
   tmsl.put("b", 1);
   System.out.println(tmsl); // {b = 1, aa = 2, aaa = 3}
   }
```

Використовуваний компаратор `StrLengthComparator` визначається класом:

```
class StrLengthComparator implements Comparator<String> {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
}
```

Реалізація `LinkedHashMap<K, V>` забезпечує порядок зберігання елементів або в порядку їх вставки, або в порядку доступу до них. Наступний фрагмент коду демонструє властивості `LinkedHashMap<K, V>`:

```
HashMap<Integer, String> lhmap =
                            new LinkedHashMap<Integer, String>();
/* Додавання пар "ключ-значення" - LinkedHashMap підтримує
  порядок вставки, метод повертає значення, що додається або
  null, якщо ключа, шо вставляється, не було у відображенні */
System.out.println(lhmap.put(12, "Вася")); // null
System.out.println(lhmap.put(2, "Петя")); // null
System.out.println(lhmap.put(7, "Маша")); // null
System.out.println(lhmap.put(49, "Вова")); // null
System.out.println(lhmap.put(3, "Іра")); // null
System.out.println(lhmap.put(2, "Оля")); // Петя
System.out.println(lhmap.put(49, "Вова")); // Вова
System.out.println(lhmap); // {12 = Вася, 2 = Оля, 7 = Маша,
                              // 49 = Вова, 3 = Іра}
/* У разі, якщо ключ, що додається, вже є у відображенні,
  додавання не виконується, метод повертає значення, що
    додається, або відповідне ключу, якщо він є у відображенні */
System.out.println(lhmap.putIfAbsent (2, "Вася")); //Оля

/* Відображення вмісту, використовуючи ітератор */
Set set = lhmap.entrySet();
Iterator iterator = set.iterator();
while (iterator.hasNext()) {
    Map.Entry mentry = (Map.Entry) iterator.next();
    System.out.print("ключ =" + mentry.getKey ()
                    + ", Значення =");
    System.out.println(mentry.getValue());
}
/* Ключ = 12, значення = Вася
   ключ = 2, значення = Оля
   ключ = 7, значення = Маша
   ключ = 49, значення = Вова
   ключ = 3, значення = Іра */


/* Отримання елемента по ключу */
int index = 2;
System.out.println("Значення за індексом" + index + "="
```

```
                    + Lhmap.get(index)); //Оля

/* Видалення елемента по ключу */
System.out.println(lhmap.remove(3)); // Іра
System.out.println(lhmap); // {12 = Вася, 2 = Оля, 7 = Маша,
                                  49 = Вова}

/* Перевірка наявності ключа і значення */
System.out.println(lhmap.containsKey (5)); // false
System.out.println(lhmap.containsValue ("Маша")); // true

/* Друк ключів */
for (int key: lhmap.keySet()) {
    System.out.print(key + " ");
}
System.out.println();
/* Заміна значення за ключем */
System.out.println(lhmap.replace (49, "Сергій")); // Вова
/* Друк значень */
for (String value: lhmap.values()) {
    System.out.print(value + " "); // Вася Оля Маша Сергій
}
System.out.println();
/* Видалення пари ключ-значення */
System.out.println(lhmap.remove(12)); // Вася
/* Друк всіх ключів і значень */
for (Map.Entry<Integer, String> entry: lhmap.entrySet()) {
    System.out.print(entry + " "); // 2 = Оля 7 = Маша 49 = Сергій
}
    System.out.println();

/* Встановлюємо accessOrder в true - зберігає елемент у порядку
  доступу: останній елемент, до якого був доступ,
  переміщується у кінець відображення */
Map<Integer, String> hm1 =
                            new LinkedHashMap<>(10, 0.75f, true);
hm1.put(1, "aaa");
hm1.put(2, "bbbb");
hm1.put(3, "ccccc");
System.out.println(hm1); // {1 = aaa, 2 = bbbb, 3 = ccccc}
System.out.println(hm1.get(1)); // aaa
System.out.println(hm1); // {2 = bbbb, 3 = ccccc, 1 = aaa}
System.out.println(hm1.get(3)); // ccccc
System.out.println(hm1); // {2 = bbbb, 1 = aaa, 3 = ccccc}
System.out.println(hm1.put(4, "dddddd")); // null
System.out.println(hm1); // {2 = bbbb, 1 = aaa, 3 = ccccc,
                         // 4 = dddddd}
```

Утилітарний клас `Collections`

Клас java.util.Collections містить статичні методи, що реалізують популярні способи роботи з колекціями: пошуку, заміни, сортування, копіювання елементів і інші. Використання найбільш популярних методів класу java.util.Collections показано у наступному фрагменті:

```
ArrayList<Integer> list =
                    new ArrayList<>(Arrays.asList(4, 1, 3, 2));
System.out.println(list); //[4, 1, 3, 2]
/* Повертає індекс елемента, який шукається */
System.out.println(Collections.binarySearch(list, 3)); // 2
/* При передачі null замість об'єкта компаратора, сортування
  виконується в природному порядку */
Collections.sort(list, null);
System.out.println(list); //[1, 2, 3, 4]
/* Сортування з використанням компаратора –
  у зворотному природному порядку */
Collections.sort(list, Collections.reverseOrder());
System.out.println(list); //[4, 3, 2, 1]
List<String> strList =
        new ArrayList<>(Arrays.asList("aaa", "bbbb", "ccccc"));
/* Змінює порядок елементів в колекції на зворотний */
Collections.reverse(strList);
System.out.println(strList); //[ccccc, bbbb, aaa]
/* Здійснює кільцеве зміщення елементів на вказане число
   позицій */
Collections.rotate (strList, 2);
System.out.println(strList); //[bbbb, aaa, ccccc]
List<String> original =
        new ArrayList<>(Arrays.asList("aaa", "bbbb", "ccccc"));
/* Отримання незмінних колекцій */
List<String> unmodifiedlist =
                        Collections.unmodifiableList(original);
/* Список заборонено модифікувати - виникає
  java.lang.UnsupportedOperationException */
// unmodifiedlist.add("dddddd");
/* Створення незмінного списку зі вказаною кількістю
   зазначеного елемента */
List<String> str = Collections.nCopies(3, "abc");
System.out.println(str); //[abc, abc, abc]
/* Додавання одного списку до іншого */
original.addAll(str);
System.out.println(original); //[aaa, bbbb, ccccc, abc, abc,
                                // abc]
/* Перемішування елементів списку */
Collections.shuffle(original);
System.out.println(original); //[aaa, ccccc, abc, abc, bbbb,
                                 // abc]
/* Підрахунок частоти входження зазначеного елемента */
System.out.println(Collections.frequency(original, "abc")); // 3

/* Заміна всіх входжень одного елемента іншим */
```

```
Collections.replaceAll(original, "abc", "cba");
System.out.println(original); //[aaa, ccccc, cba, cba, bbbb, cba]
List<Integer> numbers = new ArrayList<>();
Random rnd = new Random();
for (int i = 0; i < 10; i ++) {
    numbers.add(rnd.nextInt(100) +1);
}
System.out.println(numbers);
/* Пошук максимальних і мінімальних елементів */
System.out.println("min =" + Collections.min(numbers)
                + ", Max =" + Collections.max(numbers));
```

2. Завдання

1. Розробіть програмний код відповідно до завдання у наведеній нижче таблиці (Номер варіанта обирайте за формулою V = (№ mod 16) +1, де № - Ваш порядковий номер в журналі академгрупи).

V	завдання	V	завдання
1	Для класуPerson додайте узагальнений метод, що обчислює кількість років, місяців і днів, прожитих персоною.	9	Розробіть узагальнений інтерфейс ArithmeticNumberз абстрактними методами додавання і віднімання об'єктів-реалізацій інтерфейсу. Розробіть класи простого дробу і комплексного числа, що реалізують узагальнений інтерфейс
2	Розробіть узагальнений клас, що інкапсулює масив довільного числового типу, з узагальненим методом перемішування випадковим чином елементів масиву	10	Розробіть узагальнений клас EmployeeUtil, що інкапсулює об'єкт працівника компанії, з узагальненим методом, який порівнює зарплату поточного інкапсульованого працівника з інкапсульованим працівником-параметром
3	Розробіть узагальнений інтерфейс Pair з абстрактними методами отримання першого і другого значення пари. Розробіть реалізацію інтерфейсу з методом упорядкування пар по першому значенню	11	Для класуAccount додайте узагальнений метод, що обчислює залишок на рахунку після оплати покупки довільного товару
4	Розробіть узагальнений клас,що інкапсулює масив довільного типу, з методом пошуку максимального елемента масиву	12	Розробіть узагальнений клас, що інкапсулює масив довільного числового типу, з методом обчислення суми значень елементів масиву і методом порівняння суми значень елементів інкапсульованого масиву поточного об'єкта і об'єкта-параметра
5	Для класуPizza додайте узагальнений метод, що обчислює сумарну вартість інгредієнтів, використаних при її виготовленні	13	Для класуOrder додайте узагальнений метод, що обчислює кількість і сумарну вартість товарів, що входять в замовлення

V	завдання	V	завдання
6	Розробіть узагальнений клас, що інкапсулює масив довільного числового типу, з методом обчислювання середнього арифметичного значень елементів масиву і методом порівняння середньоарифметичного значення інкапсульованого масиву поточного об'єкта і об'єкта-параметра	14	Для класуZoo додайте узагальнений метод, що обчислює кількість тварин довільного виду в зоопарку
7	Для узагальненого класуProduct додайте узагальнений метод, що обчислює знижку вартості продукту	15	Розробіть узагальнений клас, що інкапсулює масив довільного числового типу, з методом повернення суми елементів масиву
8	Розробіть узагальнений клас, що інкапсулює масив довільного числового типу, з методом повернення масиву з позитивними елементами інкапсульованого масиву	16	Для класуGarage додайте узагальнений метод, що знаходить автомобіль з максимальним об'ємом двигуна

2. Розробіть програмний код відповідно до завдання у наведеній нижче таблиці (Номер варіанта обирайте за формулою V = (№ mod 16) +1, де № – Ваш порядковий номер в журналі академгрупи).

V	завдання	V	завдання
1	Створіть два об'єкти стека за допомогою LinkedList<Integer>, занесіть у ці стеки по 10 випадковим чином обраних значень. Потім поміняйте місцями інформацію в стеках	9	Для множини студентів, кожен з яких має підсумкові оцінки з дисциплін у вигляді колекції HashMap<String, Double>, визначте список стипендіатів, за умови проходження 43% студентів за середнім балом
2	Використовуючи колекції, розробіть програму, що моделює роботу N-місцевої автозаправки, вважаючи постійним час заправки і випадковим час прибуття автомобілів	10	У колі стоять N людей, при веденні рахунку по колу з нього видаляється кожна друга людина, поки не залишиться одна. Розробіть програмну модель задачі, використовуючи для реалізації ArrayList та LinkedList, і порівняйте час виконання програми для цих реалізацій
3	Для довільного фрагмента HTML-коду перевірте парність тегів, використовуючи стек	11	Для колекції випадкових цілих чисел у заданому користувачем програми діапазоні реалізуйте метод перетину колекцій і метод об'єднання колекцій. Продемонструйте роботу методів для таких реалізацій як ArrayList<Integer> і LinkedList<Integer> і порівняйте час виконання проргами для цих реалізацій.

V	завдання	V	завдання
4	Для довільного рядка, що представляє текст англійською мовою, виділіть всі унікальні слова і занесіть їх до HashMap<String, Integer>. Підрахуйте кількість входжень до рядка кожного з слів (враховуйте регістр слів)	12	Реалізуйте циклічну чергу для зберігання N цілих чисел з методами додавання і вилучення елементів з черги, а також методами перевірки чи порожня черга,чи повністю заповнена черга і методом, що повертає кількість вільних місць у черзі
5	Для двох списків ArrayList<Double>, що містять результати N-вимірювань струму і напруги (Табл. 2), розрахуйте значення опору методом найменших квадратів	13	Складіть два многочлени заданого ступеня, якщо коефіцієнти многочленів зберігаються в об'єктах HashMap<Integer, Integer>
6	Розробіть клас, що описує повідомлення блогу з полями автор, тема, вміст і дата. Виконайте моделювання колекції повідомлень, що підтримує можливість сортування за автором, темою, датою, а також – пошук за підрядком у темі	14	Розробіть клас автомобіля з властивостями швидкість та початкове положення на гоночній трасі. Для списку автомобілів знайдіть кількість обгонів через час t після старту
7	Створіть параметризований клас Node<T> для зберігання елемента файлової структури (каталогу, підкаталогу, файлу). Використовуйте HashSet<Node<T>> для зберігання посилань на дочірні елементи каталогу. Передбачте методи додавання і видалення елементів, а також метод обходу дерева, починаючи з зазначеного елемента, з виведенням назв елементів	15	Помножте два многочлени заданого ступеня (необов'язково однакового), якщо коефіцієнти многочленів зберігаються у списках ArrayList<Integer>
8	Розробіть клас кошика для Інтернет-магазину, що підтримує додавання і видалення товарів, зміну їх кількості та виведення ціни товару, вартості обраної кількості товару та сумарну вартість товарів у кошику	16	Для списку випадкових цілих значень і заданого числа X, не використовуючи допоміжних об'єктів і не змінюючи розмір списку, виконайте перестановку елементів списку так, щоб спочатку у списку перебували елементи, які не більше X, а потім – більше X (про сортування мова не йде)

Табл. 2 . Дані до завдання 5.

I, A	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1
U, B	0,27	0,56	0,9	1,18	1,49	1,79	2,05	2,42	2,68	3,01
I, A	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	2
U, B	3,35	3,56	3,85	4,18	4,48	4,79	5,12	5,45	5,68	5,9

3. За допомогою методу `System.currentTimeMillis()` або `System.nanoTime()` виміряйте час виконання програми.

4. Наведіть до звіту вихідний код розроблених програм, а також скріншоти з результатами їх роботи.

3. Контрольні питання

1. Дайте визначення узагальнень Java. З якої версії Java стала доступною ця технологія?
2. Наведіть приклад класу-універсального контейнера без використання узагальнень. Які проблеми використання такого класу спостерігаються? Як вони вирішуються із застосуванням технології узагальнень?
3. Наведіть приклад класу в узагальненому вигляді і фрагмент коду його використання. Вкажіть в них ім'я параметра типу і аргумент типу. Поясніть правила використання оператора<> (diamond).
4. Назвіть правила угод по використанню букв латинського алфавіту для позначення змінних типу.
5. Які класи називають сирими (простими) типами, які недоліки порівняно з узагальненими класами вони мають? Коли доцільно використовувати такі типи?
6. Що називають узагальненим методом, коли зручно проєктувати узагальнені методи? Наведіть приклад оголошення і використання узагальненого методу.
7. Для чого використовують обмежені типи? Наведіть приклад оголошення і використання узагальненого класу з обмеженим типом.
8. Що називають множинним обмеженням типів? Наведіть приклад оголошення типу з множинним обмеженням і поясніть правила для класів і інтерфейсів, які обмежують тип.
9. Що називають стиранням типів. Поясніть на прикладі, як працює ця технологія.
10. В яких випадках доцільно використання необмежених метасимвольних змінних типу? Наведіть приклад їх використання.
11. Наведіть приклади використання обмежених зверху і обмежених знизу метасимвольних змінних типу.
12. Перерахуйте обмеження для використання узагальнених типів.
13. Що називають колекціями Java? Що являє собою Java Collection Framework? Опишіть історію розвитку цього фреймворка за версіями JDK.
14. Дайте характеристику базовим інтерфейсам Java Collection Framework.
15. Опишіть умовне угруповання методів інтерфейсу Collection<E> і охарактеризуйте ці методи для реалізації – списку ArrayList<String>.
16. Що являє собою об'єкт-ітератор? Дайте характеристику методам інтерфейсу Iterator<E> на прикладі ArrayList<String>.
17. Назвіть призначення інтерфейсу List<E> і охарактеризуйте його популярні реалізації. Опишіть на прикладах додаткові до методів інтерфейсу Collection<E> методи інтерфейсу List<E>.
18. Охарактеризуйте реалізацію ArrayList<E> інтерфейсу List<E>. Опишіть внутрішню структуру об'єктів ArrayList<E>. Наведіть приклади використання додаткових до методів інтерфейсу List<E> методів класу ArrayList<E>.

19. Охарактеризуйте реалізацію LinkedList<E> інтерфейсу List<E>. Опишіть внутрішню структуру об'єктів LinkedList<E>? Наведіть приклади використання додаткових до методів інтерфейсу List<E> методів класу ArrayList<E>.
20. Наведіть приклад організації черги FIFO типу Deque<E> з реалізацією LinkedList<E>.
21. Наведіть приклад організації стека LIFO типу Deque<E> з реалізацією LinkedList<E>.
22. Назвіть призначення інтерфейсу Set<E> і дайте коротку характеристику його популярним реалізаціям.
23. Опишіть внутрішню структуру зберігання елементів у реалізації інтерфейсу Set<E> HashSet<E>. Наведіть приклад коду, який демонструє особливості HashSet<E>.
24. Поясніть необхідність перевизначення методів `int hashCode()` і `boolean equals(Object obj)` класу java.lang.Object для класів, елементи яких містяться в HashSet<E>.
25. Наведіть приклад коду, який демонструє особливості TreeSet<E>.
26. Поясніть необхідність реалізації методу порівняння інтерфейсу Comparable<E> або інтерфейсу Comparator<E>для класів, елементи яких містяться в TreeSet<E>.
27. Наведіть приклад коду, який демонструє особливості LinkedHashSet<E>.
28. Назвіть призначення інтерфейсу Map<K, V> і дайте коротку характеристику його популярним реалізаціям.
29. Опишіть внутрішню організацію HashMap<K, V> і дайте визначення властивостям `size`, `capacity`, `loadFactor` і `threshold`.
30. Опишіть внутрішню роботу HashMap<K, V> при додаванні до нього пари "ключ-значення".
31. Вкажіть призначення TreeMap<K, V> і наведіть приклад, який демонструє його роботу.
32. Вкажіть призначення LinkedHashMap<K, V> і наведіть приклад, який демонструє його роботу.
33. Назвіть призначення класу java.util.Collections і наведіть приклади використання його методів.

4. Література

1. Васильєв О. Програмування мовою Java. Тернопіль: Навчальна книга - Богдан, 2020. 696 с.
2. Horstmann C. S. Core Java, Volume I: Fundamentals. 12-th Ed. "Addison-Wesley", 2022. 1197 p.
3. Naftalin M., Wadler P. Generics and Collections. -O'Reilly Media, 2007. -286 p.
4. Kozubek A, Java Generics: extends, super and wildcards explained. Website OneWebSQL. 06 June 2013.http://onewebsql.com/blog/generics-extends-super
5. Learning the Java Language. The Java™ Tutorials. Oracle Java documentation site. URL: https://docs.oracle.com/javase/tutorial/java/TOC.html (дата звернення 08.08.2023).

6. IntelliJ IDEA IDE URL: https://www.jetbrains.com/idea/ (дата звернення 08.18.2023).
7. Структури даних в картинках. ArrayList. Хабрахабр. https://habr.com/post/128269/
8. Структури даних в картинках. LinkedList. Хабрахабр. https://habr.com/post/127864/
9. Структури даних в картинках. HashMap. Хабрахабр. https://habr.com/post/128017/