

Java SE – Laboratorium nr 7

Temat: Kolekcje

1. Definicja i idea

Często w programach zachodzi potrzeba przechowywania w pamięci nieznanej z góry liczby elementów. Możliwość taką dostarczają tzw. kontenery (w Javie reprezentowane, oczywiście, przez klasy kontenerowe). Kontenery można podzielić na dwie grupy:

- kolekcje (ang. collection) – kontenery, których elementy podlegają ściśle określonym regułom (np. elementy listy są przechowywane w określonej kolejności),
- odwzorowania (ang. map) – kontenery przechowujące pary klucz-wartość ; można je traktować jako tablice, których elementami są zmienne typu wartość, zaś indeksowane są przez zmienne typu klucz.

Do przechodzenia przez elementy kontenera można również stosować iteratory. Są one obiektami, które mogą pracować na dowolnym kontenerze, my natomiast używamy ich wszędzie w ten sam sposób. Zapewnia to większą uniwersalność i elastyczność pisanego kodu.

Fragment hierarchii klas kontenerowych:

```
java.lang.Object
|
+--java.util.Collection
|   |
|   +--java.util.List
|       |
|       +--java.util.ArrayList
|       |
|       +--java.util.LinkedList
|
+--java.util.Map
    |
    +--java.util.HashMap
    |
    +--java.util.TreeMap
```

2. Typy generyczne

Od wersji 1.5 języka Java wprowadzono typy generyczne (parametryzowane, szablonowe). Mają one szczególne zastosowanie właśnie do kontenerów.

Do tej pory, aby stworzyć kolekcję obiektów jakiegoś typu musieliśmy pozwolić na przechowywanie w niej elementów typu Object. Wyjmując element z takiej kolekcji musieliśmy rzutować go w dół na odpowiedni typ, co było uciążliwe i mogło prowadzić do pomyłek możliwych do wykrycia dopiero na etapie wykonania programu.

Typy generyczne pozwalają sparametryzować typ kontenerowy tak, aby przechowywał on tylko obiekty określonego typu. Wówczas wyjmując obiekt wiemy już, jakiej jest klasy.

Deklarując obiekt typu sparametryzowanego musimy po nazwie typu podać w nawiasach kątowych (<>) parametr, czyli typ, na który typ generyczny ma pracować. Przykładowo:

```
ArrayList listaZwykla = new ArrayList();

// zwykła lista przechowująca Object'y

ArrayList<String> listaGeneryczna = new ArrayList<String>();

// lista przechowująca tylko String'i
```

W dalszym ciągu będziemy stosować kontenery generyczne wszędzie tam, gdzie to możliwe. Jedynie tam, gdzie interfejs (API) wymusza na nas stosowanie kontenerów typów niesparametryzowanych będziemy je stosować.

Listing 0: Przykład stosowania typów generycznych:

```
public class Box<T> { // T oznacza Typ
    private T t;

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

    public T get() {
        return t;
    }
}

public class Listing0 {
    public static void main(String[] args){
        Box<Integer> test = new Box<>();
        test.set(123);
        System.out.println(test.get());
    }
}
```

Komunikat w konsoli:

```
123
```

Przykład metody zwykłej:

```
public static int maksimum(int x1, int x2) {
    if (x1 > x2) return x1;
    else return x2;
}
```

Metoda generyczna:

```
public static <T extends Comparable> T maksimumGen(T x1, T x2) {  
    if (x1.compareTo(x2) > 0) return x1;  
    else return x2;  
}
```

3. Interfejs Collection

Interfejs Collection<Typ> posiada m.in. następujące metody:

- boolean add(Typ o) – umieszcza obiekt o w kolekcji, zwraca true, jeśli kolekcja uległa zmianie w wyniku tej operacji,
- boolean addAll(Collection<Typ> c) – dodaje do kolekcji wszystkie elementy z kolekcji c, zwraca true, jeśli kolekcja uległa zmianie w wyniku tej operacji,
- void clear() – usuwa wszystkie elementy z kolekcji,
- boolean contains(Typ o) – zwraca true jeśli element o jest w kolekcji,
- boolean containsAll(Collection<Typ> c) – zwraca true jeśli wszystkie elementy z kolekcji c jest w kolekcji,
- boolean isEmpty() – zwraca true jeśli kolekcja jest pusta,
- boolean remove(Typ o) – sprawdza, czy element o jest w kolekcji, jeśli tak – usuwa go (lub jeden z nich, jeśli jest więcej), zwraca true jeśli udało się usunąć,
- boolean removeAll(Collection<Typ> c) – usuwa z kontenera wszystkie elementy znajdujące się w kontenerze c (różnica zbiorów), zwraca true jeśli usunął chociaż jeden element,
- boolean retainAll(Collection<Typ> c) – usuwa z kontenera wszystkie elementy nie znajdujące się w kontenerze c (część wspólna zbiorów), zwraca true jeśli usunął chociaż jeden element,
- int size() – zwraca ilość elementów przechowywanych w kontenerze,
- Typ[] toArray() – zwraca elementy przechowywane w kontenerze w postaci tablicy,
- Iterator<Typ> iterator() – zwraca iterator do poruszania się po kolekcji.

Oczywiście wszystkie te metody są w interfejsie jedynie zadeklarowane. Ich realizacja należy do konkretnych klas implementujących ten interfejs.

4. Interfejs List

Interfejs List jest rozszerzeniem interfejsu Collection. Deklaruje on dodatkowo m.in. następujące metody (dla interfejsu sparametryzowanego List<Typ>):

- void add(int indeks, Typ o) – dodaje obiekt o do listy na pozycji indeks,
- Typ get(int indeks) – zwraca element znajdujący się w liście na pozycji indeks,
- int indexOf(Typ o) – zwraca indeks pierwszego wystąpienia na liście obiektu o, -1 jeśli taki obiekt nie występuje,
- int lastIndexOf(Typ o) – zwraca indeks ostatniego wystąpienia na liście obiektu o, -1 jeśli taki obiekt nie występuje,
- Typ remove(int indeks) – usuwa z listy element znajdujący się na pozycji indeks a następnie go zwraca,
- Object set(int indeks, Typ o) – zastępuje element znajdujący się na pozycji indeks obiektem o, zwraca zastąpiony obiekt (starą wartość z pozycji indeks),
- List<Typ> subList(int pocz, int kon) – zwraca listę utworzoną z elementów listy wyjściowej o indeksach od pocz do kon - 1, a w przeciwnym wypadku listę pustą (gdy pocz=kon) lub wyjątek IndexOutOfBoundsException,
- ListIterator<Typ> listIterator() – zwraca iterator umożliwiający poruszanie się po liście.

Klasa ArrayList jest implementacją interfejsu List w oparciu o tablicę, której rozmiar jest w razie potrzeby zwiększany. Dzięki temu dostęp do elementów jest szybki, ale dodawanie i usuwanie elementów w innym miejscu niż na końcu jest wolne.

Listing 1: Przykład użycia ArrayList:

```
public class Listing1 {
    public static void main(String[] args) {
        Random rand = new Random();
        List<Integer> lista = new ArrayList<>();

        //dodawanie elementów do listy
        for(int x=0;x<10;x++) lista.add(x);

        System.out.println(lista);

        //czyszczenie listy i dodanie innych elementów
        lista.clear();

        for(int x=0;x<10;x++) lista.add(rand.nextInt(100));

        System.out.println(lista);
    }
}
```

Komunikat w konsoli:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 30, 32, 57, 43, 44, 74, 8, 6, 85]
```

Klasa `LinkedList` również - tak jak `ArrayList` - implementuje podstawowy interfejs `List` jak i `Queue`. Dostęp do poszczególnych elementów listy jest tu wolniejszy, ale za to operacje wstawiania i usuwania elementów z dowolnego miejsca są szybsze (zwłaszcza na początku i na końcu). Jest za to mniej wydajna w realizacji swobodnego dostępu do elementów listy.

Klasa `LinkedList<Typ>` posiada dodatkowo m.in. metody:

- `void addFirst(Typ o)` – dodaje element na początek listy,
- `void addLast(Typ o)` – dodaje element na koniec listy,
- `Typ getFirst()` – zwraca pierwszy element listy,
- `Typ getLast()` – zwraca ostatni element listy,
- `Typ removeFirst()` – usuwa pierwszy element listy, a następnie go zwraca,
- `Typ removeLast()` – usuwa ostatni element listy, a następnie go zwraca.

Te metody pozwalają na wykorzystanie jej w roli stosu, kolejki albo kolejki dwukierunkowej.

Listing 2: Przykład użycia `LinkedList`:

```
public class Listing2 {
    public static void main(String[] args) {
        Random rand = new Random();
        List<Integer> lista = new LinkedList<>();

        //dodawanie elementów do listy
        for(int x=0;x<10;x++) lista.add(x);

        System.out.println(lista);

        //czyszczenie listy i dodanie innych elementów
        lista.clear();

        for(int x=0;x<10;x++) lista.add(rand.nextInt(100));

        System.out.println(lista);
    }
}
```

Komunikat w konsoli:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[39, 83, 77, 58, 66, 59, 73, 30, 72, 8]
```

5. Interfejs Set

Kontenery implementujące interfejs Set, czyli zbiory służą do przechowywania nie powtarzających się elementów. W stosunku do interfejsu Collection interfejs Set nie implementuje żadnych nowych metod.

Istnieją dwie klasyczne implementacje interfejsu Set:

- **HashSet** – implementacja w oparciu o tablicę haszową, umożliwia bardzo szybkie zlokalizowanie elementu nawet w dużym zbiorze, obiekty przechowywane w tym zbiorze powinny implementować funkcję `hashCode()` z klasy `Object`,
- **TreeSet** – implementacja na bazie drzewa, zapewnia uporządkowanie elementów w zbiorze.

Otrzymujemy również klasę `LinkedHashSet` dziedziczącą z `HashSet`. Ten Set na zewnątrz prezentuje porządek zgodny z kolejnością wstawiania, udając najzwyklejszą listę.

Listing 3: Prezentacja kolekcji Set:

```
public class Listing3 {  
    public static void main(String[] args) {  
        Random rand = new Random();  
        Set<Integer> set1 = new HashSet<>();  
        Set<Integer> set2 = new LinkedHashSet<>();  
        Set<Integer> set3 = new TreeSet<>();  
  
        for(int x=0;x<10000;x++){  
            int var = rand.nextInt(30);  
            set1.add(var);  
            set2.add(var);  
            set3.add(var);  
        }  
  
        System.out.println("HashSet:\n" + set1);  
        System.out.println("LinkedHashSet:\n" + set2);  
        System.out.println("TreeSet:\n" + set3);  
    }  
}
```

Komunikat w konsoli:

```
HashSet:  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 16, 19, 18, 21,  
20, 23, 22, 25, 24, 27, 26, 29, 28]  
LinkedHashSet:  
[22, 24, 3, 14, 28, 4, 20, 17, 2, 25, 26, 6, 9, 7, 11, 16, 13, 5, 29, 12,  
23, 10, 27, 15, 0, 21, 8, 18, 19, 1]  
TreeSet:  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,  
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

6. Interfejs Queue

Kolejka to kontener typu *first-in, first-out* (FIFO). Znaczy to, że elementy dodaje się na koniec, a pobiera z początku. Kolejki są zwykle wykorzystywane w roli pewnego mechanizmu transferu obiektów pomiędzy różnymi obszarami programu. Są one szczególnie przydatne w programowaniu współbieżnym, bowiem pozwalają bezpiecznie przekazywać obiekty pomiędzy zadaniami.

LinkedList zawiera metody odpowiadające zachowaniu kolejki i implementuje interfejs Queue, więc można go użyć w roli implementacji kolejki. Klasa LinkedList<Typ> posiada dodatkowo m.in. metody:

- Typ element() – zwraca, ale nie usuwa, pierwszy element kolejki. Gdy kolejka jest pusta zwracany jest wyjątek NoSuchElementException,
- void offer(Typ o) – dodaje element na koniec kolejki,
- Typ peek() – zwraca, ale nie usuwa, pierwszy element kolejki. Gdy kolejka jest pusta zwracana jest wartość null,
- Typ poll() – zwraca i usuwa pierwszy element kolejki. Gdy kolejka jest pusta zwracana jest wartość null,
- Typ remove() – zwraca i usuwa pierwszy element kolejki. Gdy kolejka jest pusta zwracany jest wyjątek NoSuchElementException.

Listing 4: Przykład użycia LinkedList jako kolejki:

```
public class Listing4 {
    public static void main(String[] args) {
        Random rand = new Random();
        Queue<Integer> kolejka = new LinkedList<>();

        //dodawanie elementów do kolejki
        for(int x=0;x<10;x++) kolejka.offer(x);

        System.out.println(kolejka + "\n");

        //czyszczenie kolejki i wyświetlanie elementów
        int rozmiar = kolejka.size();
        for(int x=0;x<rozmiar;x++) System.out.print(kolejka.poll()+" ");

        //dodawanie nowych elementów
        for(int x=0;x<10;x++) kolejka.offer(rand.nextInt(100));

        System.out.println("\n\n" + kolejka);
    }
}
```


Komunikat w konsoli:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[23, 11, 2, 22, 37, 45, 31, 38, 51, 23]
```

Klasa `PriorityQueue` (kolejka priorytetowa) przewiduje, że następnym elementem wydobytym z kolejki będzie element o najwyższym priorytecie. Kiedy za pomocą metody `offer()` umieszczamy obiekt w tej kolejce to jest on wstawiany zgodnie z priorytetem. Domyślnie obiekty są układane według tak zwanego porządku naturalnego, który można zmieniać udostępniając własny komparator. Kolejka priorytetowa zapewnia, że wywołanie `peek()`, `poll()` bądź `remove()` zwróci element o najwyższym priorytecie. Typy wbudowane, takie jak `Integer`, `String`, `Character` są przystosowane do kolejki priorytetowej.

Listing 5: Przykład użycia `PriorityQueue`:

```
public class Listing5 {
    public static String losujString(){
        char[] chars = "abcdefghijklmnopqrstuvwxyz".toCharArray();
        StringBuilder sb = new StringBuilder();
        Random random = new Random();
        for (int i = 0; i < 5; i++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        Random rand = new Random();
        Queue<Integer> kolejka1 = new PriorityQueue<>();
        Queue<String> kolejka2 = new PriorityQueue<>();

        for(int x=0;x<10;x++){
            int var = rand.nextInt(100);
            System.out.print(var + " ");
            kolejka1.add(var);
        }
        System.out.println("\n" + kolejka1);

        for(int x=0;x<10;x++){
            String var = losujString();
            System.out.print(var + " ");
            kolejka2.add(var);
        }
        System.out.println("\n" + kolejka2);
    }
}
```

Komunikat w konsoli:

```
12 32 20 77 99 96 32 50 54 17
```

```
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
```

```
xxbwa rxxzr atolj jxhsz ycehd wjpnm vzbui msbfg nqyjg rvbzz  
[atolj, jxhsz, rxxzr, msbfg, rvbzz, wjpnm, vzbui, xxbwa, nqyjg, ycehd]
```

7. Dodawanie grup elementów

Klasy `Arrays` i `Collections` z biblioteki `java.util` udostępniają metody narzędziowe grupujące elementy do postaci kolekcji `Collection`. Metoda `Arrays.asList()` przyjmuje tablicę albo listę elementów wymienionych po przecinku (za pomocą zmiennych list argumentów) i zwraca obiekt `List`. Z kolei `Collections.addAll()` przyjmuje obiekt `Collection` oraz tablicę bądź listę elementów i dodaje zawartość tablicy bądź komplet elementów do wskazanego obiektu `Collection`. Poniższy listing ilustruje obie metody, a także bardziej konwencjonalną metodę `addAll()`, będącą na wyposażeniu wszystkich podtypów `Collection`.

Listing 6: Przykład dodawania grup elementów:

```
public class Listing6 {  
    public static void main(String[] args) {  
        Integer[] inty = {12, 17, 20, 50, 32, 96, 32, 77, 54, 99};  
  
        System.out.println("Użycie konstruktora i Arrays.asList()");  
        List<Integer> lista = new ArrayList<>(Arrays.asList(inty));  
        Set<Integer> zbior = new HashSet<>(Arrays.asList(inty));  
        Queue<Integer> kolejka = new  
            PriorityQueue<>(Arrays.asList(inty));  
  
        System.out.println(lista + "\n" + zbior + "\n" + kolejka);  
  
        lista.clear();  
        zbior.clear();  
        kolejka.clear();  
  
        System.out.println("Użycie Collections.addAll()");  
        Collections.addAll(lista, inty);  
        Collections.addAll(zbior, inty);  
        Collections.addAll(kolejka, inty);  
  
        System.out.println(lista + "\n" + zbior + "\n" + kolejka);  
  
        lista.clear();  
        zbior.clear();  
        kolejka.clear();  
  
        System.out.println("Użycie domyślnej metody addAll i "  
            + "Arrays.asList()");  
        lista.addAll(Arrays.asList(inty));  
        zbior.addAll(Arrays.asList(inty));  
        kolejka.addAll(Arrays.asList(inty));  
    }  
}
```

```

        System.out.println(lista + "\n" + zbior + "\n" + kolejka);
    }
}

```

Komunikat w konsoli:

```

Użycie konstruktora i Arrays.asList()
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
[17, 50, 32, 99, 54, 20, 96, 77, 12]
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
Użycie Collections.addAll()
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
[17, 50, 32, 99, 54, 20, 96, 77, 12]
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
Użycie domyślnej metody addAll i Arrays.asList()
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]
[17, 50, 32, 99, 54, 20, 96, 77, 12]
[12, 17, 20, 50, 32, 96, 32, 77, 54, 99]

```

8. Interfejs Map

Odwzorowania (mapy, słowniki, tablice asocjacyjne) są kontenerami przechowującymi pary klucz-wartość. Elementy typu klucz pełnią niejako rolę indeksów, zaś wartość – właściwych elementów kolekcji. Interfejs `Map<Typ klucza, Typ wartości>` posiada metody:

- `boolean containsKey(Object klucz)` - zwraca true, gdy dany klucz istnieje,
- `boolean containsValue(Object wartosc)` - zwraca true, gdy dana wartość istnieje,
- `Object get(Object klucz)` - zwraca przypisaną wartość pod konkretny klucz,
- `Set keySet()` - zwraca zbiór kluczy,
- `Object put(Object klucz, Object wartosc)` - wstawia pod konkretny klucz żadaną wartość,
- `Object remove(Object klucz)` - zwraca wartość klucza i usuwa dany wpis.

Interfejs `Map` implementują m.in. klasy:

- `HashMap` – implementacja oparta o tablice haszowe, zapewnia szybkie wyszukiwanie elementu na podstawie klucza,
- `TreeMap` – implementacja oparta na drzewach czerwono-czarnych, zapewnia posortowanie elementów (według klucza).

Otrzymujemy również klasę `LinkedHashMap` dziedziczącą z `HashMap`. To odwzorowanie na zewnątrz prezentuje porządek zgodny z kolejnością wstawiania.

Listing 7: Prezentacja kolekcji Map:

```
public class Listing7 {
    public static String losujString(){
        char[] chars = "abcdefghijklmnopqrstuvwxyz".toCharArray();
        StringBuilder sb = new StringBuilder();
        Random random = new Random();
        for (int i = 0; i < 5; i++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        Random rand = new Random();
        Map<Integer, String> map1 = new HashMap<>();
        Map<Integer, String> map2 = new LinkedHashMap<>();
        Map<Integer, String> map3 = new TreeMap<>();

        for(int x=0;x<10;x++){
            int var = rand.nextInt(1000);
            String s = losujString();

            map1.put(var, s);
            map2.put(var, s);
            map3.put(var, s);
        }

        System.out.println("HashMap:\n" + map1);
        System.out.println("LinkedHashMap:\n" + map2);
        System.out.println("TreeMap:\n" + map3);
    }
}
```

Komunikat w konsoli:

```
HashMap:
{221=ifpah, 414=zeasz, 854=nbhtd, 277=btjkt, 233=cmlxr, 77=ivrwp, 798=pjdvm,
40=bfkde, 709=rkzrm, 405=zirma}
LinkedHashMap:
{798=pjdvm, 709=rkzrm, 40=bfkde, 405=zirma, 233=cmlxr, 221=ifpah, 854=nbhtd,
77=ivrwp, 414=zeasz, 277=btjkt}
TreeMap:
{40=bfkde, 77=ivrwp, 221=ifpah, 233=cmlxr, 277=btjkt, 405=zirma, 414=zeasz,
709=rkzrm, 798=pjdvm, 854=nbhtd}
```

9. Interfejs Iterator

Aby użyć kontenera, trzeba znać jego dokładny typ. Przypuśćmy, że chcielibyśmy napisać kawałek kodu, który nie wie lub nie dba o to, z jakiego typu kontenerem ma do czynienia, tak by mógł być zastosowany dla różnorodnych kontenerów bez potrzeby przepisywania kodu.

Do uzyskania takiej abstrakcji może być wykorzystane pojęcie iteratora. Jest to obiekt, którego zadaniem jest przemieszczanie się po sekwencji elementów i wybieranie każdego z napotkanych obiektów bez wiedzy programisty-użytkownika lub przejmowania się wewnętrzną strukturą takiej sekwencji. Dodatkowo iterator jest "lekkim" obiektem - jego stworzenie mało kosztowne. Z tego powodu często napotkamy pozornie dziwne ograniczenia iteratorów, na przykład może przesuwać się tylko w jednym kierunku. Można z nim zrobić niewiele:

- 1) Poprosić kolekcję Collection o udostępnienie iteratora, wywołując jej metodę o nazwie iterator(). Iterator ten będzie gotów do zwrócenia pierwszego elementu sekwencji.
- 2) Uzyskać następny obiekt z ciągu dzięki metodzie next().
- 3) Sprawdzić, czy są jakieś inne obiekty dalej w sekwencji za pomocą hasNext().
- 4) Usunąć ostatni zwrócony element przez iterator, stosując remove().

Listing 8: Przykład użycia iteratora dla kolekcji Collection:

```
public class Listing8 {  
    public static void main(String[] args) {  
        Random rand = new Random();  
        Set<Integer> zbior = new HashSet<>();  
  
        for(int x=0;x<10;x++) zbior.add(rand.nextInt(1000));  
  
        System.out.println(zbior);  
  
        Iterator<Integer> it = zbior.iterator();  
        int i = 0;  
        while(it.hasNext()){  
            System.out.print(it.next() + " ");  
            if(i++ == 5) it.remove();  
        }  
  
        System.out.println("\n" + zbior);  
    }  
}
```

Komunikat w konsoli:

```
[668, 702, 326, 745, 87, 477, 502, 996, 809, 657]  
668 702 326 745 87 477 502 996 809 657  
[668, 702, 326, 745, 87, 502, 996, 809, 657]
```

Listing 9: Przykład użycia iteratora dla kolekcji Map:

```
public class Listing9 {  
    public static String losujString(){  
        char[] chars = "abcdefghijklmnopqrstuvwxyz".toCharArray();  
        StringBuilder sb = new StringBuilder();  
        Random random = new Random();  
        for (int i = 0; i < 5; i++) {  
            char c = chars[random.nextInt(chars.length)];  
            sb.append(c);  
        }  
    }  
}
```

```

        return sb.toString();
    }

    public static void main(String[] args) {
        Random rand = new Random();
        Map<Integer, String> map = new HashMap<>();

        for(int x=0;x<10;x++){
            int var = rand.nextInt(1000);
            String s = losujString();
            map.put(var, s);
        }

        System.out.println("HashMap:\n" + map);

        Iterator<Integer> it = map.keySet().iterator();
        while(it.hasNext()){
            int var = it.next();
            System.out.print(var + ":" + map.get(var) + " || ");
        }
    }
}

```

Komunikat w konsoli:

```

HashMap:
{493=lzuhv, 357=vpzhg, 823=yunno, 822=eobxb, 953=eflrv, 353=ijhtf, 67=hkaha,
27=pvwhi, 388=rgsmx, 465=cxgee}
493:lhuv || 357:vpzhg || 823:yunno || 822:eobxb || 953:eflrv || 353:ijhtf
|| 67:hkaha || 27:pvhwi || 388:rgsmx || 465:cxgee ||

```

ListIterator to nieco rozbudowany podtyp interfejsu Iterator, zwracany jedynie przez klasy List. Jest on dwukierunkowy. Może zwracać indeksy elementu na poprzedniego i następnego, względem bieżącej pozycji iteratora na liście, a także pozwala na zastępowanie ostatnio odwiedzonego elementu za pomocą metody set(). ListIterator wskazujący na początek listy tworzy się wywołaniem metody listIterator(). Można także utworzyć ListIterator zaczynając od wybranego elementu n, wystarczy wywołać listIterator(n).

Listing 10: Przykład użycia ListIterator:

```

public class Listing10 {
    public static void main(String[] args) {
        Random rand = new Random();
        List<Integer> lista = new ArrayList<>();

        for(int x=0;x<10;x++) lista.add(rand.nextInt(1000));

        System.out.println(lista);

        ListIterator<Integer> it = lista.listIterator();

        while(it.hasNext()){
            System.out.print(it.next() + " ");
        }
        System.out.println();
    }
}

```

```

        int i = 9;
        while(it.hasPrevious()){
            System.out.print(it.previous() + " ");
            if(i-- == 5) it.set(-24);
        }

        System.out.println("\n" + lista);
    }
}

```

Komunikat w konsoli:

```

[501, 821, 748, 420, 344, 899, 569, 396, 363, 479]
501 821 748 420 344 899 569 396 363 479
479 363 396 569 899 344 420 748 821 501
[501, 821, 748, 420, 344, -24, 569, 396, 363, 479]

```

10. Kolekcje a polimorfizm

Korzyści jakie możemy osiągnąć z tytułu dziedziczenia byłyby bardzo ograniczone, gdyby nie polimorfizm. Polimorfizm oznacza możliwość traktowania obiektów różnych podtypów pewnego wspólnego typu w taki sam sposób.

Wyobraźmy sobie teraz, że chcemy zaimplementować w naszej aplikacji sklepu internetowego koszyk, który przechowuje wybrane przez klienta artykuły i który potrafi odpowiedzieć na pytanie – jaka jest łączna wartość zamówienia. Aby policzyć wartość zamówienia trzeba zsumować ceny poszczególnych produktów, niezależnie od tego, jakiego typu są te produkty. Wygodnie jest móc traktować wszystkie obiekty reprezentujące produkty takie jak książki, płyty muzyczne, gry i wszelakie inne w ten sam sposób. Możemy wówczas przejrzeć kolekcję produktów z koszyka i dla każdego z nich wywołać metodę zwracającą ich cenę. Suma wyników będzie wartością zamówienia.

Na listingu poniżej zostanie zaprezentowany przykład koszyka z zakupami. Zobaczymy, jak działa polimorfizm i w jaki sposób działają metody/pola przestonięte oraz jak użyć metod konkretnych klas.

Listing 11: Przykład użycia polimorfizmu w kolekcji:

```

public class Produkt {
    private int cena;
    private String nazwa;

    public Produkt(int cena, String nazwa){
        this.cena = cena;
        this.nazwa = nazwa;
    }

    public int getCena(){
        return cena;
    }

    public String getNazwa(){
        return nazwa;
    }
}

```

```

    }

    public void wypisz(){
        System.out.println(nazwa + ", cena: " + cena);
    }
}

public class Ksiazka extends Produkt {
    private String autor;

    public Ksiazka(int cena, String nazwa, String autor){
        super(cena, nazwa);
        this.autor = autor;
    }

    @Override
    public void wypisz(){
        System.out.println("Książka " + super.getNazwa()
            + ", autor " + autor
            + ", cena: " + super.getCena());
    }
}

public class AudioCD extends Produkt {
    private String nazwa, wykonawca;

    public AudioCD(int cena, String nazwa, String wykonawca){
        super(cena, null);
        this.nazwa = nazwa;
        this.wykonawca = wykonawca;
    }

    @Override
    public void wypisz(){
        System.out.println(nazwa + ", wykonawca " + wykonawca
            + ", cena: " + super.getCena());
    }

    public String getNazwa2(){
        return nazwa;
    }
}

public class Listing11 {
    public static void main(String[] args) {
        List<Produkt> koszyk = new ArrayList<>();
        int doZaplaty = 0;

        koszyk.add(new Produkt(0, "test"));
        koszyk.add(new Ksiazka(10, "Ksiazka1", "Autor1"));
        koszyk.add(new AudioCD(30, "AudioCD1", "Wykonawca1"));
        koszyk.add(new Ksiazka(25, "Ksiazka2", "Autor2"));
        koszyk.add(new AudioCD(50, "AudioCD2", "Wykonawca2"));

        Iterator<Produkt> it = koszyk.iterator();
        while(it.hasNext()){
            Produkt p = it.next();
            p.wypisz();
        }
    }
}

```



```

        doZapłaty += p.getCena();

        //Sposób na dostanie się do konkretnego produktu
        if(p instanceof Ksiazka) ((Ksiazka)p).wypisz();
        else if(p instanceof AudioCD){
            ((AudioCD)p).wypisz();
            System.out.println(p.getNazwa() + " :: "
                               + ((AudioCD)p).getNazwa2());
        }
        System.out.println();
    }

    System.out.println("Kwota do zapłaty: " + doZapłaty);
}

```

Komunikat w konsoli:

```

test, cena: 0

Książka Ksiazka1, autor Autor1, cena: 10
Książka Ksiazka1, autor Autor1, cena: 10

AudioCD1, wykonawca Wykonawca1, cena: 30
AudioCD1, wykonawca Wykonawca1, cena: 30
null :: AudioCD1

Książka Ksiazka2, autor Autor2, cena: 25
Książka Ksiazka2, autor Autor2, cena: 25

AudioCD2, wykonawca Wykonawca2, cena: 50
AudioCD2, wykonawca Wykonawca2, cena: 50
null :: AudioCD2

Kwota do zapłaty: 115

```

11. Podsumowanie

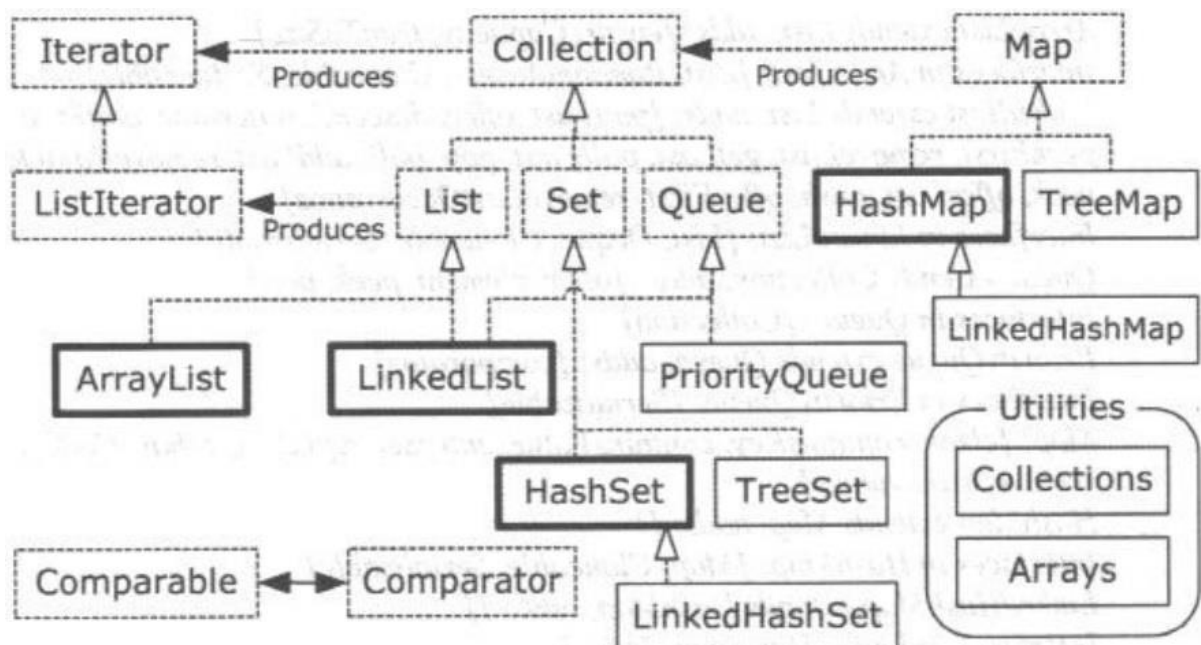
Najważniejsze informacje dotyczące kolekcji:

- 1) Tablica przypisuje indeksy liczbowe do obiektów. Przechowując obiekty znanego typu, nie trzeba więc rzutować wyniku podczas wyszukiwania obiektu. Może być wielowymiarowa i przechowywać typy podstawowe. Jednak jej rozmiar może ulec zmianie po stworzeniu.
- 2) Collection przechowuje pojedyncze elementy, podczas gdy odwzorowanie Map - pary obiektów skojarzonych. Dzięki typom ogólnym wprowadzonym w Javie SE5 można określać typ obiektów przechowywanych w kontenerach, co blokuje próby wstawiania elementów niewłaściwego typu i eliminuje konieczność rzutowania typów elementów przy wydobywaniu ich z kontenerów. Tak kolekcje, jak i odwzorowania automatycznie dopasowują swój rozmiar w miarę dodawania elementów. Kontener nie może przechowywać wartości typów podstawowych, ale

dzięki mechanizmowi pakowania takich wartości w obiekty posługiwanie się nimi w połączeniu z kontenerami jest całkiem wygodne.

- 3) Podobnie jak tablica, również lista kojarzy indeksy liczbowe z obiektami - można sobie wyobrazić tablice i listy jako kontenery uporządkowane.
- 4) Należy stosować ArrayList, jeżeli wykonuje się wiele operacji swobodnego dostępu, oraz LinkedList - jeżeli wystąpi wiele operacji wstawiania i usuwania ze środka listy.
- 5) Zachowanie typowe dla kolejek i stosów implementują: Queue i LinkedList.
- 6) Map jest sposobem na skojarzenie nie liczb, ale obiektów z innymi obiektami. Kontener HashMap skupia się na szybkim dostępie, podczas gdy TreeMap przechowuje swoje klucze w porządku posortowanym, zatem nie jest tak szybki jak HashMap. Kontener LinkedHashMap przechowuje elementy w kolejności ich dodawania, ale dzięki haszowaniu zapewnia szybki dostęp do elementów.
- 7) Zbiór Set akceptuje tylko jeden egzemplarz każdego z rodzajów obiektu. HashSet zapewnia maksymalnie szybkie przeszukiwanie, a TreeSet utrzymuje elementy w kolejności posortowanej. Zbiór LinkedHashSet przechowuje elementy w kolejności w jakiej były dodawane.
- 8) Nie ma potrzeby stosowania przestarzałych klas Vector, Hashtable i Stack w nowym kodzie.

Przykładowa hierarchia kontenerów:



Zadania:

Zadanie 1

Utwórz dowolną klasę, a potem zainicjalizowaną tablicę obiektów tej klasy. Zawartością tablicy wypełnij listę List. Wyłuskaj z niej fragment listy metodą subList(), a następnie usuń tę podlistę z oryginalnej listy.

Zadanie 2

Utwórz i zapełnij listę List<Integer>. Utwórz drugą listę List<Integer>. Użyj ListIterator do przejrzenia elementów pierwszej listy i wstawienia ich do listy drugiej, ale w odwrotnej kolejności.

Zadanie 3

Napisz klasę o nazwie Command, która zawiera ciąg znaków String i metodę operation(), która go wypisuje. Napisz drugą klasę, z metodą wypełniającą kolejkę Queue obiektami klasy Command i zwracającą wypełniony kontener. Przekaż kontener do metody z trzeciej klasy: metoda ma konsumować obiekty z kolejki Queue, wywołując dla każdego z nich metodę operation().

Zadanie 4

1. Utwórz nową klasę Kangur ze składową int nrKangura, inicjalizowaną z poziomu konstruktora. Wyposaż klasę w metodę skok(), wypisującą wartość tej składowej i sygnalizującą wykonywanie podskoków. Utwórz kontener ArrayList i wstaw do niego obiekty Kangur (minimum 10). Teraz skorzystaj z metody get() kontenera w celu przejrzenia jego zawartości i wywołania metody skok() dla każdego umieszczonego w nim kangura.
2. Zmodyfikuj uzyskany kod tak, aby przeglądało listę (i wywoływało metodę skok()) za pomocą iteratora.
3. Weź klasę Kangur z podpunktu 1 i umieść jej elementy w kontenerze HashMap, kojarząc każdy egzemplarz Kangur (wartość) z nazwą ("Jacek", "Marta" itd.) w postaci obiektu String (klucz). Pozyskaj iterator zbioru zwracanego przez keySet() i wykorzystaj go do przejrzenia kontenera HashMap. Wypisz w konsoli imiona kangurów oraz odpowiadające im numery, zwracane poprzez metodę skok().
4. Wyodrębnij z kontenera HashMap (utworzonego w poprzednim podpunkcie) pary, posortuj je według kluczy i umieść całość w kontenerze LinkedHashMap.

Zadanie 5

W poniższym zadaniu napiszemy własną klasę odnośnie stosu, czyli co zostanie włożone na stos jako ostatnie jest pierwszym elementem, który można z niego zdjąć (LIFO).

1. Utwórz klasę Stos<T> zawierającą:
 - zainicjowane prywatne pole LinkedList<T> stos,
 - publiczne metody:
 - void push(T v) - wkłada element na stos,
 - T peek() - zwraca pierwszy element stosu, ale go nie usuwa,
 - T pop() - zwraca pierwszy element stosu i usuwa go,

- `boolean empty()` - sprawdza, czy stos jest pusty,
 - `String toString()` - wypisuje elementy naszego stosu.
2. Stosy są często używane do obliczania wyrażeń w językach programowania. Za pomocą utworzonej klasy `Stos` oblicz poniższe wyrażenie, w którym '+' oznacza "umieszczenie następnej litery na stosie", a '-' "zdejście szczytowego elementu stosu i wypisanie go na wyjściu".
Wyrażenie do wyliczenia: "+B+a+l---+a+g+a---+n--+w--+l+i+t---+e--+r+k--+a+c+h---"

Zadanie 6

Napisz prostą symulację pójścia na zakupy:

1. utwórz tablicę `String` zawierającą 10 nazw produktów dostępnych w sklepie,
2. po wejściu do sklepu zapełnij koszyk (będący stosem - użyj klasy z poprzedniego zadania) losową ilością losowych towarów (koszyk pomieści maksymalnie 15 przedmiotów),
3. umieść siebie w kolejce (`Queue`) na losową pozycję (maksymalna ilość osób w kolejce wynosi 10),
4. gdy dojdiesz do kasy wypisz produkty znajdujące się w koszyku.

Zadanie 7

1. Napisz klasę `Film` zawierającą prywatne pole `czasTrwania`, tytuł oraz `czyObejrzany` (wartość `true` gdy film został obejrzany). Dodaj konstruktor i odpowiednie metody obsługujące pola.
2. Stwórz dwie klasy dziedziczące z klasy `Film`, np. `Horror` i `Komedia`. Każda z nich ma zawierać pole `final` pole `typ` (wskazujące na typ filmu). Dodaj konstruktory i odpowiednie metody obsługujące pola. Dodaj metody wypisujące wszystkie dane odnośnie filmu.
3. W głównej klasie stwórz mapę prywatną `Kolekcja`, gdzie kluczem będzie numer egzemplarza, a wartością obiekt `Film`. Dodaj kilka elementów do utworzonej mapy.
4. Za pomocą iteratora wypisz posiadane filmy w kolekcji. Przykład:
"Nr (nr w kolekcji), Tytuł: (tytuł filmu), Czas trwania: (czas), Obejrzany: (tak/nie), Typ: (gatunek filmu)"