AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Java

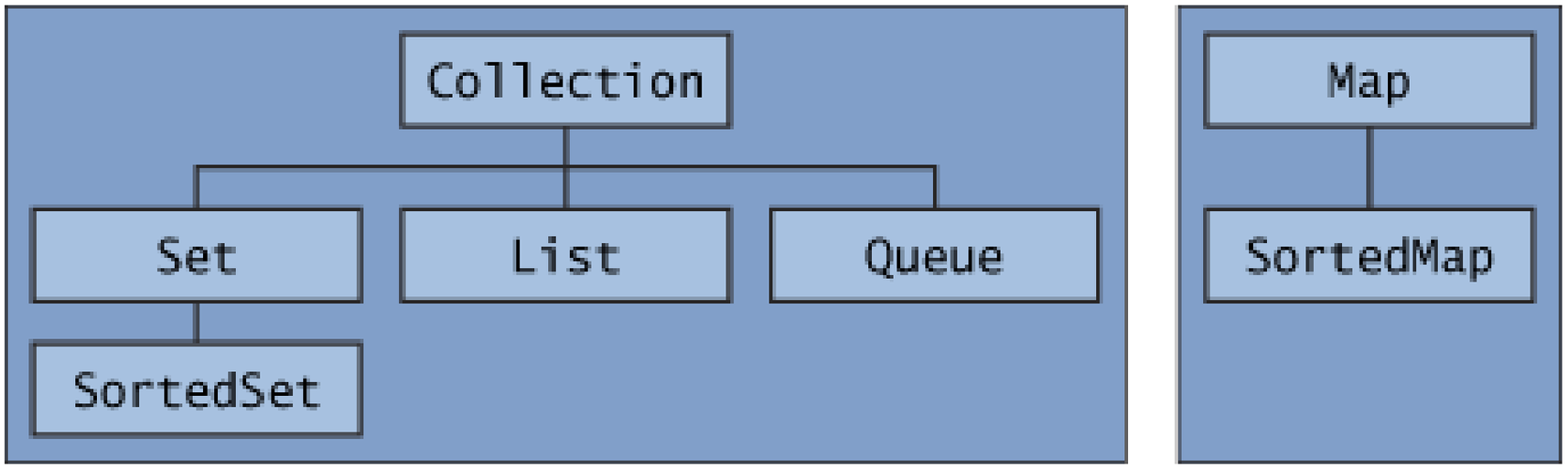
klasy kolekcji

- Wprowadzenie
- Podstawowe interfejsy w Java Collection Framework
- Implementacje
- Algorytmy
- Wskazówki implementacyjne
- Interoperabilność

Wprowadzenie

- Kolekcja: obiekt grupujący w sobie inne obiekty, np. talia kart (kolekcja kart), folder email (kolekcja wiadomości) etc.
- Poznane przykłady: tablica, Hashtable, Vector
- Java Collection Framework: spójna architektura do manipulacji kolekcjami:
  - interfejsy: abstrakcyjne typy danych
  - implementacje: konkretne implementacje abstrakcji
  - algorytmy: metody, które wykonują obliczenia np. sortowanie, wyszukiwanie <-> obiekt implementujący interfejs. Właściwość: polimorfizm (jedna metoda może posiadać wiele różnych implementacji)
- Zalety używania:
  - redukcja nakładu: poprzez dostarczenie przydatnych struktur danych i algorytmów
  - zwiększenie wydajności i jakości: m. in. możliwość szybkiej zmiany implementacji
  - interoperabilność: pomiędzy niezwiązanymi API
  - Reużywalność kodu

Interfejsy



- Uwaga: wszystkie interfejsy są zgeneryzowane:

```
public interface Collection<E>
```

- Collection: nadklasa hierarchii. Grupa elementów.
- Set: zbiór, kolekcja w której nie może być duplikatów
- List: kolekcja posortowana, inaczej sekwencja, może zawirać duplikaty
- Queue: magazyn FIFO (wyjątek: kolejki priorytetowe), dostarcza dodatkowe operacje: wstawianie, ekstrakcja, inspekcja
- Map: matematycznym odpowiednikiem jest funkcja, mapuje klucze na wartości
- SortedSet i SortedMap: dostarcza sortowania elementów/kluczy

Interfejs Collection

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Kolekcje bez typów

```
class Circle {}
class Square {}

List collection = new LinkedList();
Square square = new Square();
Circle circle = new Circle();
collection.add(square);
collection.add(circle);

System.out.println(collection.size());
System.out.println(collection.contains(square));
System.out.println(collection.contains(circle));
System.out.println(collection.contains(new Circle()));
```

Wynik:

2

true

true

false

Kolekcje z typami

```
class Circle {}
class Square {}

List<Circle> circles = new LinkedList<>();
Square square = new Square();
Circle circle = new Circle();
//  circles.add(square); Błąd kompilacji
circles.add(circle);

System.out.println(circles.size());
System.out.println(circles.contains(square));
System.out.println(circles.contains(circle));
System.out.println(circles.contains(new Circle()));
```

Wynik:

1

false

true

false

Przykład

```
List<Integer> numbers = new LinkedList<>();
numbers.add(1);
numbers.add(2);
numbers.add(3);
numbers.add(4);
numbers.add(0);
//numbers.remove(0); to usunie wartość o indexie 0 czyli 1
numbers.remove(new Integer(0));
```

Wynik: [1, 2, 3, 4, 0]

add, remove

```
List<Circle> circles = new LinkedList<>();
Circle redCircle = new Circle("red");
Circle blueCircle = new Circle("blue");

circles.add(redCircle);
circles.add(blueCircle);

System.out.println(circles.get(0));
System.out.println(circles.size());

circles.remove(redCircle);
System.out.println(circles.get(0));
System.out.println(circles.size());
```

Wynik:

red circle

2

blue circle

1

addAll, removeAll

```
List<Circle> duplicate = new LinkedList<>();
duplicate.addAll(circles);
System.out.println(circles);

circles.removeAll(Collections.singleton(redCircle));
System.out.println(circles.get(0));
System.out.println(circles.size());
```

Wynik:

[red circle, blue circle]

blue circle

1

asArray

```
List<Circle> circles = new LinkedList<Circle>();
circles.add(new Circle("blue"));
circles.add(new Circle("red"));

Object[] circlesArray = circles.toArray(); // zwraca tablice objektów
Circle[] circlesArray2 = circles.toArray(new Circle[0]);

Object[] objectArray = circles.toArray();

System.out.println(circlesArray);
System.out.println(circlesArray2);
System.out.println(objectArray);
```

Wynik:

[Ljava.lang.Object;@6d6f6e28

[Lmain.Main$1Circle;@135fbaa4

[Ljava.lang.Object;@45ee12a7

Sortowanie

```
List<String> words = Arrays.asList("jeden", "dwa", "trzy", "cztery");
Collections.sort(words);
System.out.println(words);
```

Wynik: [cztery, dwa, jeden, trzy]

```
Collections.sort(words, new Comparator<String>() {
    public int compare(String a, String b) {
        if(a.length() == b.length()){
            return a.compareTo(b);
        }
        return a.length() - b.length();
    }
});
```

Wynik: [dwa, trzy, jeden, cztery]

Przechodzenie po kolekcji

- Sposób 1, for-each:

```
for (Object o : collectionName)
    System.out.println(o);
```

- Sposób 2, poprzez iterator: //TypeParameters: `E`-the type of elements returned by this iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- Przykład, filtrowanie elementów:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

Bulk operations

- Wykonują operacje na całej kolekcji
- Możliwe do zaimplementowanie samemu z wykorzystaniem podstawowych operacji jednak jest to MAŁO EFEKTYWNE!
- Operacje:
  - containsAll — zwraca true jest kolekcja zawiera wszystkie elementy z podanej kolekcji.
  - addAll — dodaje wszystkie elementy z jednej do drugiej.
  - removeAll —usuwa wszystkie z kolekcji, które są w podanej kolekcji.
  - retainAll — usuwa z jednej wszystkie, które nie są w podanej.
  - clear — usuwa wszystko

Konwersje

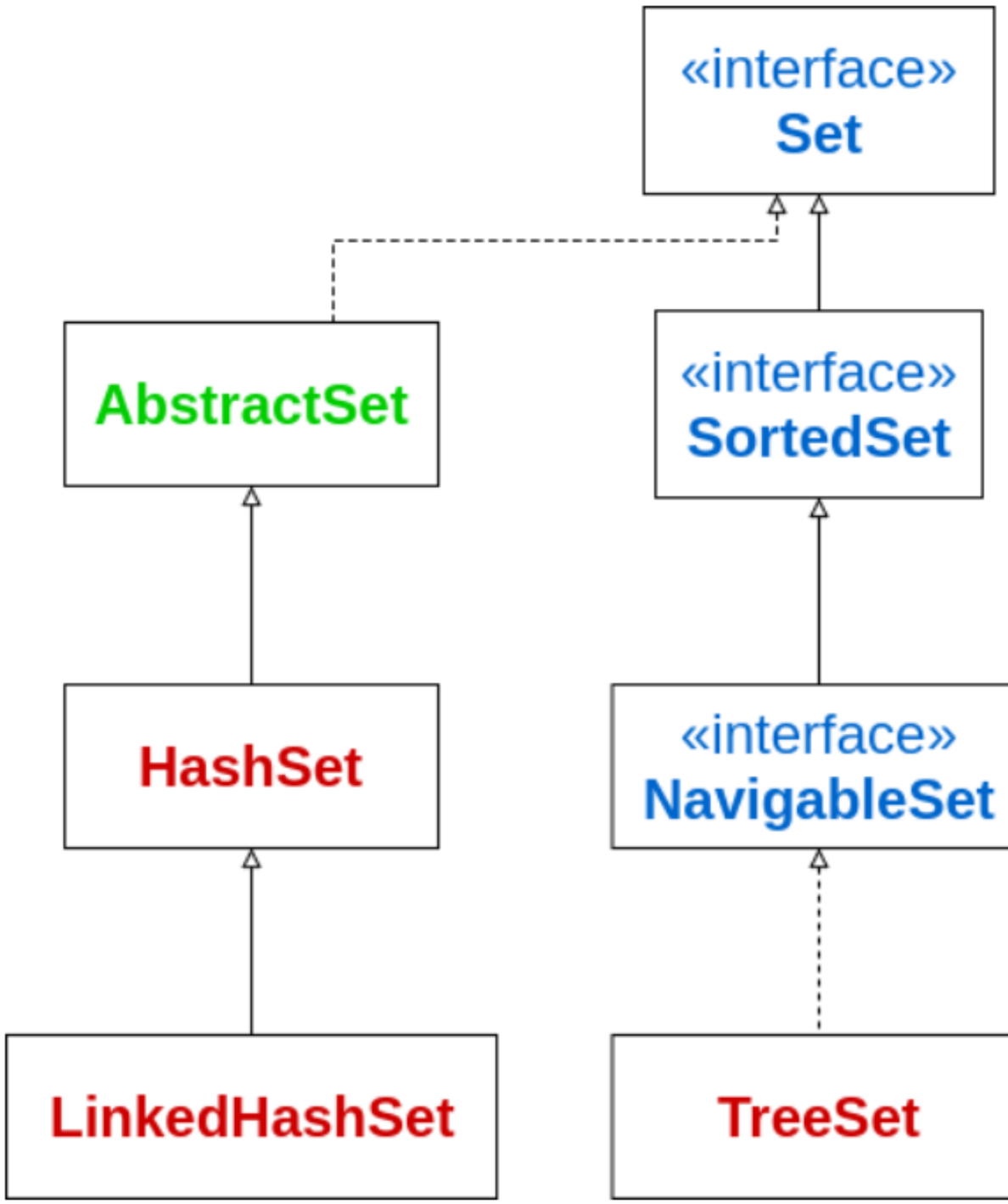
- Operacja toArray zwraca kolekcję w postaci tablicy array.
- Przykład (c jest kolekcją obiektów typu Object):

```
Object [] a = c.toArray();
```

Interfejs Set

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

`T`-the type of objects that this object may be compared to


«interface»
Set
AbstractSet
«interface»
SortedSet
HashSet
«interface»
NavigableSet
LinkedHashSet
TreeSet

Implementacje

- HashSet: tablica hasz, najwydajniejsza lecz brak gwarancji co do kolejności przy sortowaniu
- TreeSet: przechowuje elementy w drzewie czerwono-czarnym, zachowuje kolejność, wyraźnie wolniejsza implementacja od HashSet
- LinkedHashSet: sortuje według kolejności wstawiania, ciut wolniejsza impelemtacja od HashSet za cenę gwarancji kolejności

Klasa	Cechy	Zastosowanie
HashSet	• zbiór nieposortowany • kolejność iteracji nieokreślona, może się zmieniać • dodanie elementu oraz sprawdzenie czy istnieje ma złożoność O(1) • pobranie kolejnego elementu ma złożoność O(n/h), gdzie h to parametr wewnętrzny (pesymistyczny przypadek: O(n))	Sytuacje, kiedy nie potrzebujemy dostępu do konkretnego elementu, ale potrzebujemy często sprawdzać, czy dany element już istnieje w kolekcji (czyli chcemy zbudować zbiór unikalnych wartości).W praktyce często znajduje zastosowanie do raportowania/zliczeń oraz jako 'przechowalnia' obiektów do przetworzenia (jeśli ich kolejność nie ma znaczenia).
Linked-HashSet	• analogiczne, jak HashSet (dziedziczy po nim) • kolejność elementów używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez elementy w tej samej kolejności) • pobranie kolejnego elementu ma złożoność O(1)	Jesto to mniej znana i rzadziej stosowana implementacja, często może ona zastąpić HashSet poza specyficznymi przypadkami. W praktyce do iterowania w określonej kolejności często używane sa inne kolekcje (Listy, kolejki)
TreeSet	• Przechowuje elementy posortowane wg porządku naturalnego (jeśli implementują one interjfejs Comparable, w przeciwnym wypadku porządek jest nieokreslony) • Wszystkie operacje (dodanie, sprawdzenie czy istnieje oraz pobranie kolejnego elementu) mają złożoność O(log n)	Jeśli potrzebujemy, aby nasz zbiór był posortowany bez dodatkowych operacji (sortowanie nastepuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.

equals and hashCode

```
class Circle {
    private String color;

    public Circle(String color){
            this.color = color;
    }

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

    @Override
    public boolean equals(Object obj) {
            Circle that = (Circle) obj;
            return this.color.equals(that.color);
    }
}
```

equals and hashCode cd.

```
Set<Circle> set = new HashSet<>();
set.add(new Circle("red"));
set.add(new Circle("red"));

System.out.println(set.size());
System.out.println(set.contains(new Circle("red")));
```

Bez zaimplementowanych metod hashCode i equals:

2
false

Z zaimplementowanymi metodami equals i hashCode:

1
true

TreeSet - przykład

```
TreeSet<Integer> numbers = new TreeSet<>();
numbers.add(1);
numbers.add(4);
numbers.add(2);
numbers.add(3);
numbers.add(0);
System.out.println(numbers);
```

Wynik: [0, 1, 2, 3, 4]

LinkedHashMap - przykład

```
LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
numbers.add(1);
numbers.add(4);
numbers.add(2);
numbers.add(3);
numbers.add(0);
System.out.println(numbers);
```

Wynik: [1, 4, 2, 3, 0]

Problem

- Załóżmy, że mamy jakąś kolekcję zawierającą duplikaty i chcemy zrobić kolekcję bez duplikatów:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

- Z zachowaniem kolejności:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

- Metoda działająca dla dowolnej kolekcji:

```
public static <E> Set<E> removeDups(Collection<E> c) {
    return new LinkedHashSet<E>(c);
}
```

Przykład

- Wypisanie duplikatów, liczby unikalnych słów i pełną listę bez duplikatów:

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplikat: " + a);

        System.out.println(s.size() + " unikalne słowa: " + s);
    }
}
```

Bulk operations

- `s1.containsAll(s2)` — true jeśli s2 jest podzbiorem s1. returns true if s2 is a subset of s1.
- `s1.addAll(s2)` — z s1 robi sumę s1 i s2.
- `s1.retainAll(s2)` — z s1 robi przecięcie s1 i s2.
- `s1.removeAll(s2)` — z s1 robi różnicę zbiorów s1 i s2 (uwaga: operacja asymetryczna).

Komentarz

- Jeśli chcemy wykonać operacje mnogościowe na zbiorach bez uszkadzania ich należy zrobić kopię jednego zbioru przed wywołaniem operacji bulk:

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

Przykład

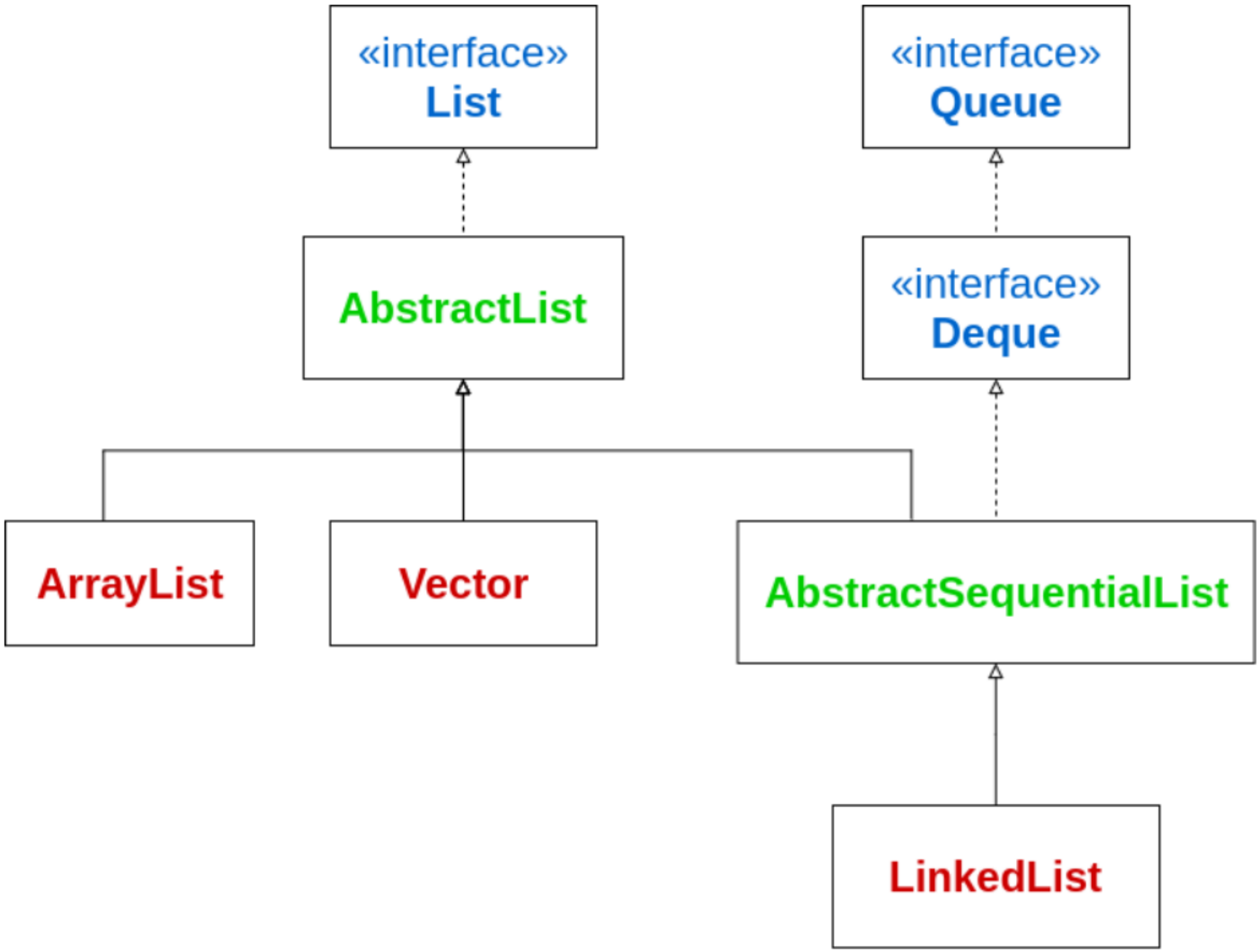
- Modyfikacja programu FindDups: tym razem chcemy wiedzieć, które słowo na liście pojawiło się raz, a które więcej niż jeden raz – nie chcemy wypisywać duplikatów wiele razy.

```
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups    = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        uniques.removeAll(dups);

        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

Interfejs List

- Lista jest posortowaną kolekcją elementów i dopuszcza duplikaty.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index, Collection<? extends E> c); //optional
    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    // Range-view
    List<E> subList(int from, int to);
}
```


«interface»
List
«interface»
Queue
AbstractList
«interface»
Deque
ArrayList
Vector
AbstractSequentialList
LinkedList

Klasa	Cechy	Zastosowanie
ArrayList	• w 'tle' przechowuje dane w tablicy, której samodzielnie zmienia rozmiar wg potrzeb • dostęp do dowolnego elementu w czasie O(1) • dodanie elementu O(1), pesymistyczny przypadek O(n) • usunięcie elementu O(n)	Najczęstszy wybór z racji najbardziej uniwersalnego zastosowania. Inne imeplementacje mają przewagę tylko w bardzo specyficznych przypadkach. Jeśli nie wiesz, jakiej listy potrzebujesz, wybierz tą.
LinkedList	• przechowuje elementy jako lista powiązanych ze sobą obiektów (tj. pierwszy element wie, gdzie jest drugi, drugi wie, gdzie trzeci itd) • dostęp do dowolnego elementu O(n) (iteracyjnie O(1)) • dodanie elementu O(1) • usunięcie elementu O(1)	LinkedList ma przewagę w przypadku dodawania elementów pojedynczo, w dużej ilości, w sposób trudny do przewidzenia wcześniej, kiedy przejmujemy się ilością zajmowanej pamięci.W praktyce nie spotkałem się z sytuacją, w której LinkedList byłoby wydajniejsze od ArrayList
Vector	• Istnieje od początku Javy, z założenia miała to być obiektowa reprezentacja tablicy • Funkcjonalność i cechy analogiczne do ArrayList, ale znacznie mniej wyrafinowane	API oficjalnie zaleca korzystanie z klasy ArrayList zamiast Vector

Przykład z Arrays.asList()

```
List<String> words = Arrays.asList("jeden", "dwa", "trzy");
System.out.println(words.get(0));
System.out.println(words.set(0,"zero"));
System.out.println(words.get(0));
System.out.println(words.add("dwa"));
```

Wynik:

```
jeden
jeden
zero
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(AbstractList.java:148)
    at java.util.AbstractList.add(AbstractList.java:108)
    at main.Main.main(Main.java:85)
```

Przykład cd.

```
List<String> words = new LinkedList<>();
    words.add("jeden");
    words.add("dwa");
    words.add("trzy");
    System.out.println(words);
    words.subList(1,3).clear();
    System.out.println(words);
    words.subList(0,1).add("sześć");
    System.out.println(words);
```

Wynik:

[jeden, dwa, trzy]

[jeden]

[jeden, sześć]

Lista: dostęp do elementów i wyszukiwanie

- Dostęp poprzez: get, set, remove
- Operacja zmiany elementów na liście (operacja polimorficzna! Zamiana el. bez względdu na wybraną implementację):

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

- Tasowanie elementów z wykorzystaniem powyższej metody:

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

Iteratory dla list

- Oprócz iteratorów zwracanych przez operacje odziedziczone z Collection lista dostarcza bogatszego iteratora: ListIterator:

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

- 2 metody do pobierania iteratora: jedna pozycjonuje kursor na początku, dr

Operacje previous i next

- Uwaga: nextIndex zwraca index elementy, który będzie zwracany przez next. Analogicznie previousIndex zwróci index elementu zwracanego przez previous.
- Liczba zwracana przez nextIndex jest zawsze 1 większa niż zwaracana przez previousIndex.
- Zachowania graniczne: kursor na początku to previousIndex zwraca -1. Kursor na końcu to nextIndex zwraca collection.size().

Fragment listy

- Operacja subList(int from, int to) zwraca fragment listy jako listę
- Usunięcie fragmentu listy:

```
list.subList(fromIndex, toIndex).clear();
```

- Poszukiwanie elementu w zadanym zakresie:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Uwaga: zwracany jest index w liście będącej wynikiem subList

Przykład

- Pobranie rozdania kart z talii:

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

Przykład c.d.

- Program wykorzystuje operacje shuffle oraz dealHand do rozdania z talii 52 kart. Pobiera z lini poleceń 2 parametry: listę rozdań oraz listę kart w rozdaniu:

```
import java.util.*;

public class Deal {
  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("Usage: Deal hands cards");
      return;
    }
    int numHands = Integer.parseInt(args[0]);
    int cardsPerHand = Integer.parseInt(args[1]);

    // Make a normal 52-card deck.
    String[] suit = new String[] {
        "spades", "hearts", "diamonds", "clubs" };
    String[] rank = new String[] {
        "ace","2","3","4","5","6","7","8",
        "9","10","jack","queen","king" };

    // Shuffle the deck.
        Collections.shuffle(deck);

        if (numHands * cardsPerHand > deck.size()) {
          System.out.println("Not enough cards.");
          return;
        }

        for (int i=0; i < numHands; i++)
          System.out.println(dealHand(deck, cardsPerHand));
      }//main

      public static <E> List<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
      }
    }
```

Algorytmy

- sort — sortuję listę algorytmem merge sort
- shuffle — permutacja
- reverse — odwrócenie
- rotate — rotacja o zadaną odległość
- swap — zamiana elementów na podanych pozycjach.
- replaceAll — zamiana wszystkich wystąpień wartości inną
- fill — napisanie każadej wartości w liście wartością zadaną.
- copy — kopiowania od listy-źródła do listy-celu.
- binarySearch — wyszukuje w posrtowanej liście elementu wykorzystując wysukiwanie binarne.
- indexOfSubList
- lastIndexOfSubList

Interface Queue

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

- Każda operacja istnieje w dwóch wariantach:
  - rzucający wyjątek gdy operacja zawiedzie
  - zwracająca specjalną wartość (null lub false w zależności od metody)
- Operacje:
  - add, offer
  - remove, poll
  - element, peek

Przykład

- Program pobiera parametr liczbowy a następnie tworzy kolejkę od zadanej liczby do 0 i co sekundę usuwa jeden element.

```
import java.util.*;

public class Countdown {
    public static void main(String[] args)
            throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--)
            queue.add(i);
        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

Comparator

Domyślnie java wie w jaki sposób priorytetować typy “podstawowe”, np Integer jest porównywany po jego wartości, natomiast String po wartości ASCII każdego jego znaku.

Na to zachowanie możemy wpłynąć. Jeżeli elementy przechowywane w kolejce będą implementować interfejs: java.lang.Comparable, jak ma to miejsce w przypadku klas *String* i *Integer*, to zostaną one ułożone zgodnie z implementacją metody: java.lang.Comparable#compareTo.

```
class Circle implements Comparable{
    private String color;
    private Integer value;

    @Override
    public int compareTo(Object o) {
            Circle that = (Circle) o;
            return this.value.compareTo(that.value);
    }
}
```

Kolejka priorytetowa - Comparator

```
PriorityQueue<Circle> queue = new PriorityQueue<>();
queue.offer(new Circle("red", 2));
queue.offer(new Circle("blue", 1));
queue.offer(new Circle("white", 0));
queue.offer(new Circle("black", 3));
```

```
List<Circle> list = new ArrayList<>();
while (!queue.isEmpty()) {
      list.add(queue.poll());
}
System.out.println(list);
```

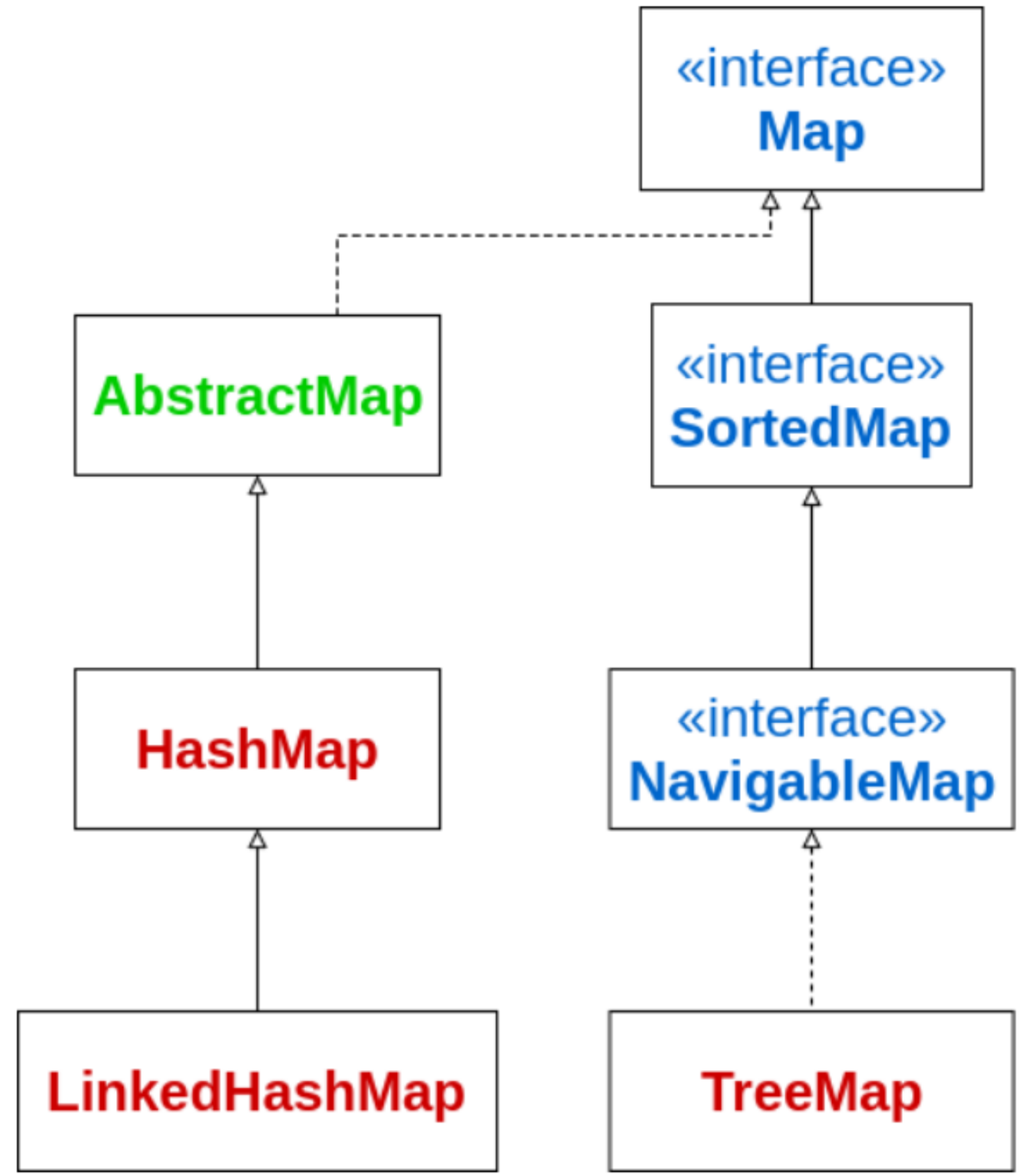
Wynik: [white, blue, red, black]

Interfejs Map

- Modeluje matematyczną abstrakcję funkcji

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

`K` - the type of keys maintained by this map
`V` - the type of mapped values


«interface»
Map
AbstractMap
«interface»
SortedMap
HashMap
«interface»
NavigableMap
LinkedHashMap
TreeMap

Klasa	Cechy	Zastosowanie
HashMap	• mapa nieposortowana • kolejność iteracji nieokreślona, może się zmieniać • dodanie elementu oraz sprawdzenie czy klucz istnieje ma złożoność O(1) • pobranie kolejnego elementu ma złożoność O(h/n), gdzie h to parametr wewnętrzny	Ogólny przypadek, tworzenie lokalnej pamięci podręcznej czy 'słownika' o ograniczonym rozmiarze, zliczanie wg klucza.
Linked-HashMap	• analogiczne, jak HashMap (dziedziczy po niej) • kolejność kluczy używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez klucze w tej samej kolejności) • pobranie kolejnego elementu ma złożoność O(1)	Podobnie jak LinkedHashSet, rzadziej znana i stosowana. Przydatna, jeśli potrzebujemy iterować po kluczach w założonej kolejności.
TreeMap	• Przechowuje elementy posortowane wg porządku naturalnego kluczy (jeśli implementują one interjfejs Comparable, w przeciwnym wypadku porządek jest nieokreslony) • Wszystkie operacje (dodanie, sprawdzenie czy klucz istnieje oraz pobranie kolejnego elementu) mają złożoność O(log n)	Jeśli potrzebujemy, aby nasz zbiór był posortowany wg kluczy bez dodatkowych operacji (sortowanie nastepuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.
Hashtable	• Historyczna klasa, która w Javie 1.2 została włączona do Java Collecion API	Oficjalnie zaleca się korzystanie z HashSet w większości wypadków zamiast Hashtable

Podstawowe metody

```
Map<String,Integer> numbers = new HashMap<>();
numbers.put("jeden", 1);
numbers.put("dwa", 2);
numbers.put("trzy", 3);
System.out.println(numbers.get("dwa"));
System.out.println(numbers.remove("dwa"));
System.out.println(numbers.get("dwa"));
```

Wynik:

2

2

null

```
Map<String,Integer> numbers = new HashMap<>();
numbers.put("jeden", 1);
numbers.put("dwa", 2);
numbers.put("trzy", 3);
System.out.println(numbers.containsKey("jeden"));
System.out.println(numbers.containsKey("cztery"));
System.out.println(numbers.containsValue(1));
System.out.println(numbers.containsValue(4));
System.out.println(numbers.size());
```

Wynik:

true

false

true

false

3

Przykład

- Zliczanie słów na liście argumentów:

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

Przejście po mapie

- Z wykorzystaniem operacji zwracających kolekcje:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

- Z wykorzystaniem iteratora:

```
// Filter a map based on some property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

Iterowanie po mapie

Iterowanie po mapie z pobieraniem wartości i klucza:

```
    Map<String, String> map = new HashMap<>();
for (Map.Entry<String, String> entry : map.entrySet()){
            String key = entry.getKey();
            String value = entry.getValue();
}
```

Strumienie

Strumienie służą do przetwarzania danych. Dane mogą być przechowywane w kolekcji, mogą być wynikiem pracy z wyrażeniami regularnymi.

W strumień można opakować dowolny zestaw danych. Pozwalają w łatwy sposób zrównoleglić pracę na danych. Dzięki temu przetwarzanie dużych zbiorów może być dużo szybsze. Kładą one nacisk na operacje jakie należy przeprowadzić na danych.

Przykład

Mamy klasę BoardGame która reprezentuje grę planszową. Chcemy wybrać nazwy tych gier które spełniają następujące warunki:

- powinna mieć ocenę wyższą niż 8,
- powinna kosztować mniej niż 150 zł.

```
class BoardGame {
   public String name;
   public Double rating;
   public Double price;
}
```

Przykład

Rozwiązanie standardowe:

```
List<BoardGame> games = new LinkedList<>();

for (BoardGame game : games) {
    if (game.rating > 8) {
        if (game.price < 150) {
                System.out.println(game.name.toUpperCase());
        }
    }
}
```

Przykład

Rozwiązanie z użyciem streamów:

```
games.stream()
        .filter(game -> game.rating > 8)
        .filter(game -> game.price < 150)
.map(game -> game.name.toUpperCase())
.forEach(System.out::println);
```

Operacje na strumieniach

Operacje związane ze strumieniami można podzielić na trzy rozłączne grupy:

- tworzenie strumienia,
- przetwarzanie danych wewnątrz strumienia,
- zakończenie strumienia.

Każdy strumień ma dokładnie jedną metodę, która go tworzy na podstawie danych źródłowych. Następnie dane te są przetwarzane przez dowolną liczbę operacji. Każda z tych operacji tworzy nowy strumień danych wywodzący się z poprzedniego. Na samym końcu strumień może mieć dokładnie jedną metodę kończącą pracę ze strumieniem.

Interfejs collection a strumienie

- interfejs Collection został rozszerzony o metodę stream
- metoda ta została wprowadzona aby ograniczyć wsteczną nie kompatybilność
- metoda ta jest **domyślna** w interfejsie Collection
- metoda zwraca element typu Stream

Metody interfejsu

Interfejs Stream umożliwia wykonywania metod funkcyjnych:

- allMatch
- anyMatch
- collect
- concat
- count
- distinct
- empty
- filter
- findAny
- findFirst
- flatMap
- forEach
- map
- itd.

Filter, map, collection

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

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
parzyste = numbers.stream().filter(e -> e % 2 == 0).
                   map(e -> e.toString()).
                   collect(Collectors.toList());
System.out.println(parzyste);
```

Wynik: [2, 4, 6]

```
double sum = numbers.stream().reduce(0, (x,y) -> { return x + y; });
System.out.println(sum);
```

Wynik: 21.0

Kolejność obiektów

- Sortowanie listy: Collections.sort(list)
- Jeśli w liście są obiekty String sortowanie będzie alfabetyczne, jeśli Daty to chronologicznie po dacie
- Jak się odbywa sortowanie?
  Odp.: Aby sortowanie mogło się odbyć obiekty w kolekcji muszą implementowac interfejs Comparable (def. jedną operację: compareTo).
- W przypadku próby sortowania listy z obiektami nie implementującymi tego interfejsu to dostaniemy ClassCastException

Sortowianie c.d.

- Sortowanie w innej kolejności niż naturalna lub sortowanie obiektów nie implementujących interfejsu Comparable?
- Rozwiązanie jeden: dostarczenie obiektu Comparator:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- metoda zwraca 0 gdy obiekty równe, -1 gdy pierwszy argument mniejszy od drugiego i +1 gdy większy

Przykład

- Mamy klasę Employee (atrybuty: name, number, hireDate), dla której naturalne sortowanie odbywa się po nazwisku. Szef jednak chce listy posortowanej po stażu pracy.

```
import java.util.*;
public class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER = new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
        }
    };
    // Employee database
    static final Collection<Employee> employees = ... ;
    public static void main(String[] args) {
        List<Employee>e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
     }
}
```

Interfejs SortedSet

- Zbiór posortowany według kolejności naturalnej (lub określonej przez Comparator dodany przy tworzeniu zbioru)

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Interfejs SortedMap

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Przykład

```
SortedMap<Integer, String> sm = new TreeMap<Integer, String>();
sm.put(2, "two");
sm.put(3, "three");
sm.put(5, "five");
sm.put(4, "four");
sm.put(1, "one");

sm.values().forEach(System.out::println);
```

Wynik:

```
one
two
three
four
five
```

Metody pomocnicze - Collections

- binarySearch - wyszukiwanie połówkowe
- checkedCollection - operacje modyfikacji są weryfikowane w czasie wykonania, a nie kompilacji
- disjoint - sprawdza rozłączność dwóch kolekcji
- emptyList - zwraca pustą listę
- fill - wypełnia listę dostarczonymi wartościami
- max - zwraca element największy
- rotate - przenosi elementy w kierunku końca
- synchronizedList - metody są synchronizowane, iterowanie wymaga synchronizacji na widoku
- unmodifiableList - zwraca niemodyfikowalny widok danej listy