Teoria do lab #13

Kolekcje

Definicja kolekcji

W kontekście programowania obiektowego w Javie, kolekcja (ang. 'collection') odnosi się do struktury danych, która grupuje obiekty w pojedynczym zbiorniku. Java oferuje szeroki zestaw klas kolekcji w ramach Java Collections Framework, które pozwalają na przechowywanie, przetwarzanie i manipulację zbiorami danych w elastyczny i efektywny sposób.

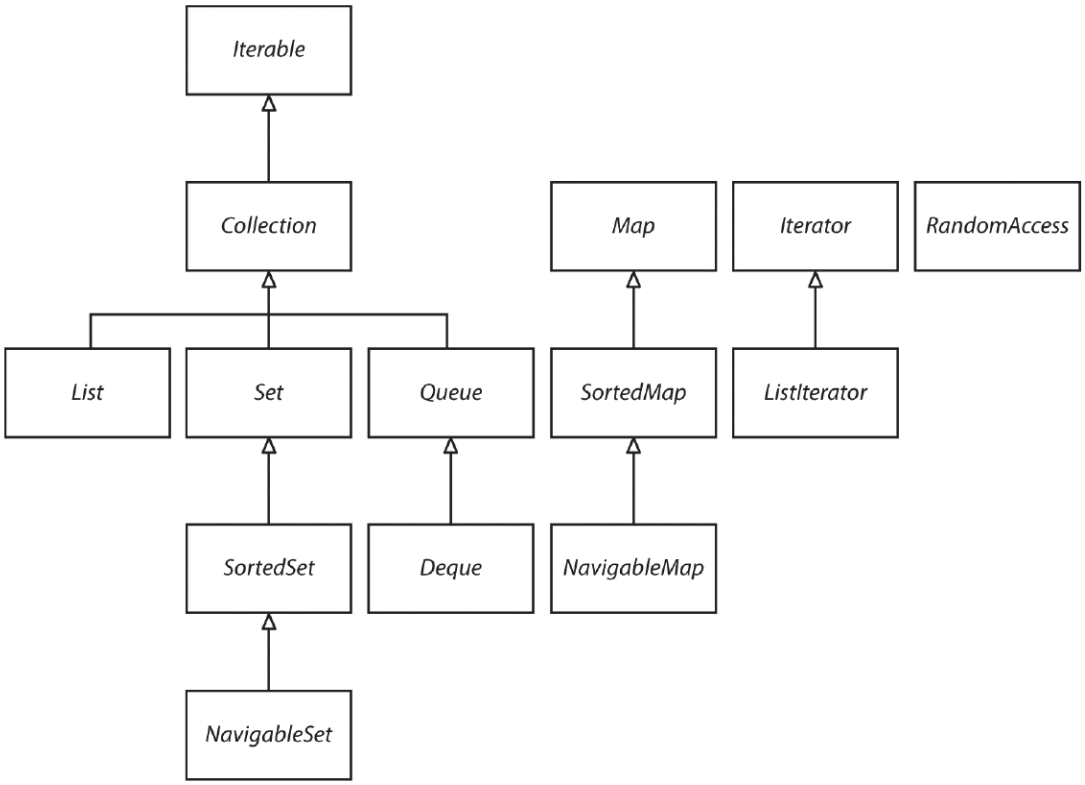
Uwaga: w internecie można znaleźć misz masz własności, w szczególności pod kątem złożoności obliczeniowej. Warygodnym źródłem jest dokumentacha.

Dobra lektura

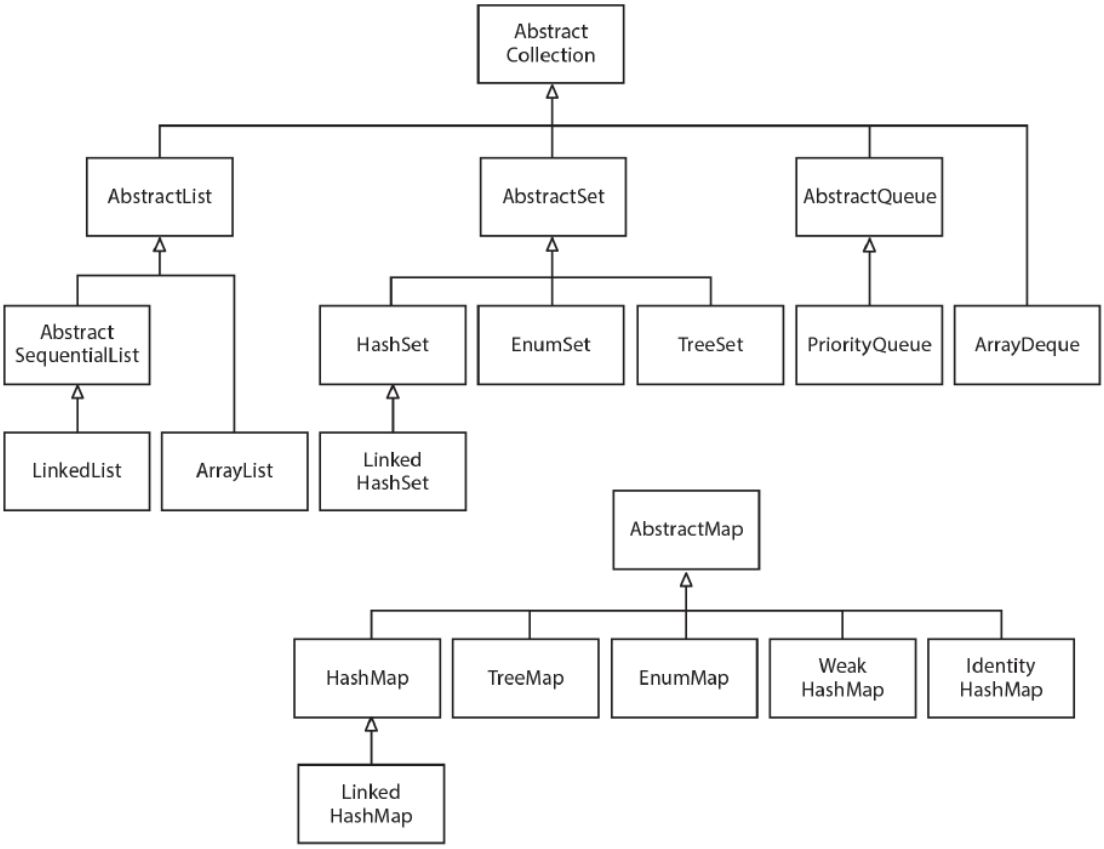
Książka w trakcie powstawania:

Maurice Naftalin, Philip Wadler, Java Generics and Collections, O'Reilly Media, Inc. - wydanie drugie planowane na czerwiec 2024, choć można znaleźć fragemty w internecie.

Ilustacji z książki: Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.

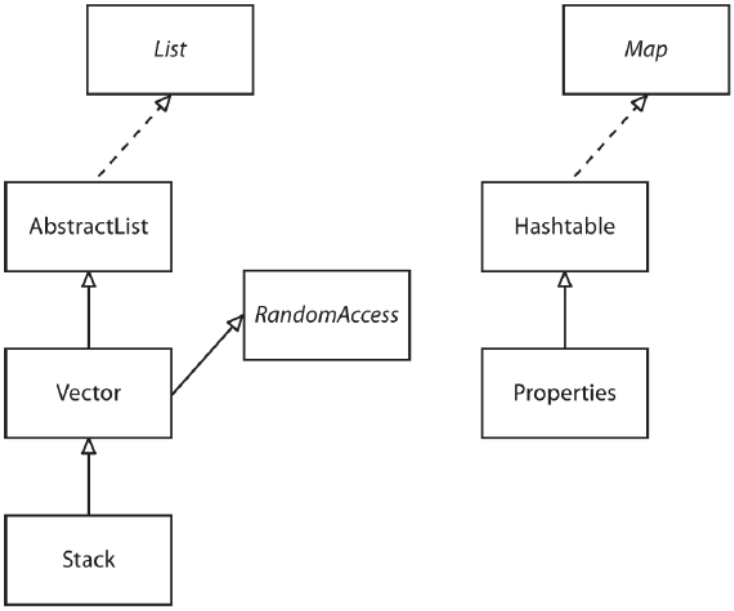


Rysunek 9.4. Interfejsy architektury kolekcji



Rysunek 9.5. Klasy z systemu kolekcji Javy

Rysunek 9.12.
Stare klasy
w ramowym
systemie kolekcji



Do nauki wersje generyczne:

- interfejs `Iterator`
- interfejs `Collection`
- `ArrayList`
- `LinkedList`
- `HashSet`
- `TreeSet`
- `HashMap`
- `TreeMap`

Interfejs `Iterator`

- https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Iterator.html

Generyczny interfejs `Iterator<E>` jest fundamentalnym elementem Java Collections Framework, umożliwiającym przeglądanie elementów kolekcji, takich jak listy, zbiory (sets) czy kolejki (queues).

Metody:

- `boolean hasNext()`: Ta metoda sprawdza, czy w kolekcji są jeszcze jakieś elementy do przetworzenia. Zwraca `true`, jeśli kolekcja posiada kolejne elementy.
- `E next()`: Zwraca następny element z kolekcji. Gdy nie ma więcej elementów, rzucany jest wyjątek `NoSuchElementException`.
- `void remove()`: Usuwa z kolekcji ostatni element zwrócony przez metodę `next()`. Metoda ta może rzucić wyjątek `UnsupportedOperationException`, jeśli operacja usuwania nie jest wspierana przez daną kolekcję.
- `forEachRemaining(Consumer<? super E> action)`: Służy do wykonania danej akcji dla każdego pozostałego elementu w iteracji, zaczynając od aktualnej pozycji iteratora.

Interfejs `Collection`

- https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collection.html

Interfejs `Collection<E>` jest jednym z podstawowych interfejsów w Java Collections Framework i służy jako korzeń dla innych interfejsów kolekcji, takich jak `List`, `Set`, czy `Queue`. Jest to generyczny interfejs, co oznacza, że można go parametryzować różnymi typami obiektów (reprezentowanymi przez `E`).

Metody:

Interfejs `Collection<E>` w Javie zawiera wiele metod, które są fundamentalne dla manipulowania i dostępu do kolekcji danych. Oto niektóre z najpopularniejszych i najczęściej używanych metod tego interfejsu:

1. `add(E e)`: Dodaje określony element do kolekcji. Zwraca `true`, jeśli kolekcja zmieniła się w wyniku wywołania tej metody.
2. `addAll(Collection<? extends E> c)`: Dodaje wszystkie elementy z określonej kolekcji do bieżącej kolekcji. Zwraca `true`, jeśli bieżąca kolekcja zmieniła się w wyniku wywołania tej metody.
3. `clear()`: Usuwa wszystkie elementy z kolekcji. Po wykonaniu tej metody kolekcja jest pusta.
4. `contains(Object o)`: Sprawdza, czy kolekcja zawiera określony element. Zwraca `true`, jeśli kolekcja zawiera przynajmniej jedno wystąpienie określonego elementu.
5. `containsAll(Collection<?> c)`: Sprawdza, czy kolekcja zawiera wszystkie elementy z określonej kolekcji.
6. `isEmpty()`: Sprawdza, czy kolekcja jest pusta (nie zawiera żadnych elementów). Zwraca `true`, jeśli kolekcja nie zawiera żadnych elementów.

7. `iterator()`: Zwraca iterator do przeglądania elementów w kolekcji. Iterator pozwala na sekwencyjne przechodzenie przez elementy kolekcji.
8. `remove(Object o)`: Usuwa jedno wystąpienie określonego elementu z kolekcji, jeśli istnieje. Zwraca `true`, jeśli kolekcja zmieniła się w wyniku wywołania tej metody.
9. `removeAll(Collection<?> c)`: Usuwa z tej kolekcji wszystkie jej elementy, które są zawarte w określonej kolekcji.
10. `retainAll(Collection<?> c)`: Zachowuje tylko te elementy w kolekcji, które są zawarte w określonej kolekcji. Innymi słowy, usuwa z tej kolekcji wszystkie elementy, których nie ma w określonej kolekcji.
11. `size()`: Zwraca liczbę elementów w kolekcji.
12. `toArray()`: Zwraca tablicę zawierającą wszystkie elementy kolekcji.

Co zyskujemy?

Możliwość tworzenie algorytmów generycznych opartych na kolekcjach.

Projekt W13, example25

```
package example25;

import java.util.*;

public class Test25 {

    public static void main(String[] args) {
        Integer[] numbers = {1, -2, 33, 76, 5};
        System.out.println(max(Arrays.asList(numbers)));
        LinkedList<String> strings = new LinkedList<>();
        strings.add("one");
        strings.add("two");
        strings.add("three");
        System.out.println(max(strings));
    }

    public static <T extends Comparable<T>> T max(Collection<T> c) {
        if (c == null || c.isEmpty()) {
            throw new NoSuchElementException("Collection is null or empty");
        }
        Iterator<T> iterator = c.iterator();
        T largest = iterator.next();
```

```
        while (iterator.hasNext()) {
            T next = iterator.next();
            if (largest.compareTo(next) < 0) {
                largest = next;
            }
        }
        return largest;
    }
}
```

`LinkedList` (lista powiązana)

`LinkedList<E>` w Javie jest implementacją listy powiązanej, która jest częścią Java Collection Framework. Jest to dynamiczna struktura danych, co oznacza, że może dynamicznie rosnąć i zmniejszać się, dodając lub usuwając elementy.

1. **Implementacja**: `LinkedList` implementuje interfejsy `List`, `Deque`, i `Queue`. Może więc działać jako lista, dwustronna kolejka (deque) lub kolejka (queue).
2. **Węzły**: Każdy element (węzeł) w `LinkedList` przechowuje dane oraz referencje do poprzedniego i następnego elementu, co umożliwia łatwe dodawanie i usuwanie elementów.
3. **Dostęp do elementów**: Dostęp do elementów w `LinkedList` jest sekwencyjny, co oznacza, że czas dostępu do elementów jest proporcjonalny do ich położenia.
4. **Złożoność czasowa**: Dodawanie/usuwanie na początku lub końcu listy ma złożoność O(1). Wyszukiwanie, dodawanie lub usuwanie elementów w środku listy ma złożoność O(n), gdzie n to liczba elementów w liście.
5. **Użycie pamięci**: Każdy element listy wiązanej wymaga dodatkowej pamięci na przechowywanie referencji do następnego i poprzedniego elementu, co czyni `LinkedList` bardziej wymagającą pod względem pamięci w porównaniu z `ArrayList`.

Metody: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

1. **void add(int index, E element)**: Wstawia określony element na określonej pozycji na liście.
2. **boolean add(E e)**: Dodaje określony element na końcu listy.
3. **boolean addAll(int index, Collection<? extends E> c)**: Wstawia wszystkie elementy z określonej kolekcji do tej listy, zaczynając od określonej pozycji.
4. **boolean addAll(Collection<? extends E> c)**: Dodaje wszystkie elementy z określonej kolekcji na końcu tej listy, w kolejności zwracanej przez iterator danej kolekcji.

5. **void addFirst(E e)**: Wstawia określony element na początku listy.
6. **void addLast(E e)**: Dodaje określony element na końcu listy.
7. **void clear()**: Usuwa wszystkie elementy z tej listy.
8. **Object clone()**: Zwraca płytką kopię tej listy `LinkedList`.
9. **boolean contains(Object o)**: Zwraca `true`, jeśli lista zawiera określony element.
10. **Iterator** descendingIterator(): Zwraca iterator po elementach tej dwukierunkowej kolejki w odwrotnej kolejności sekwencyjnej.
11. **E element()**: Pobiera, ale nie usuwa, głowę (pierwszy element) tej listy.
12. **E get(int index)**: Zwraca element na określonej pozycji na liście.
13. **E getFirst()**: Zwraca pierwszy element na liście.
14. **E getLast()**: Zwraca ostatni element na liście.
15. **int indexOf(Object o)**: Zwraca indeks pierwszego wystąpienia określonego elementu na liście, lub -1, jeśli lista nie zawiera tego elementu.
16. **int lastIndexOf(Object o)**: Zwraca indeks ostatniego wystąpienia określonego elementu na liście, lub -1, jeśli lista nie zawiera tego elementu.
17. **ListIterator** listIterator(int index): Zwraca list-iterator elementów tej listy (w odpowiedniej kolejności), zaczynając od określonej pozycji na liście.
18. **boolean offer(E e)**: Dodaje określony element jako ogon (ostatni element) tej listy.
19. **boolean offerFirst(E e)**: Wstawia określony element na początku tej listy.
20. **boolean offerLast(E e)**: Wstawia określony element na końcu tej listy.
21. **E peek()**: Pobiera, ale nie usuwa, głowę (pierwszy element) tej listy.
22. **E peekFirst()**: Pobiera, ale nie usuwa, pierwszy element tej listy lub zwraca `null`, jeśli lista jest pusta.
23. **E peekLast()**: Pobiera, ale nie usuwa, ostatni element tej listy lub zwraca `null`, jeśli lista jest pusta.
24. **E poll()**: Pobiera i usuwa głowę (pierwszy element) tej listy.
25. **E pollFirst()**: Pobiera i usuwa pierwszy element tej listy, lub zwraca `null`, jeśli lista jest pusta.
26. **E pollLast()**: Pobiera i usuwa ostatni element tej listy, lub zwraca `null`, jeśli lista jest pusta.
27. **E pop()**: Usuwa i zwraca element ze stosu reprezentowanego przez tę listę.

28. **void push(E e)**: Wstawia element na stos reprezentowany przez tę listę.
29. **E remove()**: Pobiera i usuwa głowę (pierwszy element) tej listy.
30. **E remove(int index)**: Usuwa element na określonej pozycji na liście.
31. **boolean remove(Object o)**: Usuwa pierwsze wystąpienie określonego elementu z tej listy, jeśli jest obecny.
32. **E removeFirst()**: Usuwa i zwraca pierwszy element z tej listy.
33. **boolean removeFirstOccurrence(Object o)**: Usuwa pierwsze wystąpienie określonego elementu na tej liście (podczas przeglądania listy od głowy do ogona).
34. **E removeLast()**: Usuwa i zwraca ostatni element z tej listy.
35. **boolean removeLastOccurrence(Object o)**: Usuwa ostatnie wystąpienie określonego elementu na tej liście (podczas przeglądania listy od głowy do ogona).
36. **LinkedList** reversed(): Zwraca widok tej kolekcji w odwrotnej kolejności.
37. **E set(int index, E element)**: Zastępuje element na określonej pozycji na liście określonym elementem.
38. **int size()**: Zwraca liczbę elementów na tej liście.
39. **Spliterator** spliterator(): Tworzy późno wiążący i szybko reagujący na błędy `Spliterator` po elementach tej listy.
40. **Object[] toArray()**: Zwraca tablicę zawierającą wszystkie elementy tej listy w odpowiedniej kolejności (od pierwszego do ostatniego elementu).
41. T[] toArray(T[] a): Zwraca tablicę zawierającą wszystkie elementy tej listy w odpowiedniej kolejności (od pierwszego do ostatniego elementu); typ czasu wykonania zwróconej tablicy jest taki sam, jak określonej tablicy.

Przykład

Projekt W13, example26

```
package example26;

import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Tworzenie trzech różnych LinkedList
        LinkedList<String> list1 = new LinkedList<>();
```

```
        LinkedList<Integer> list2 = new LinkedList<>();
        LinkedList<Double> list3 = new LinkedList<>();

        // Dodawanie elementów
        list1.add("Jabłko");
        list1.add("Banana");
        list2.add(1);
        list2.add(2);
        list3.add(1.1);
        list3.add(2.2);

        // Dodawanie na początku i na końcu
        list1.addFirst("Pomarańcza");
        list1.addLast("Gruszka");

        // Usuwanie elementów
        list2.removeFirst();
        list2.removeLast();

        // Pobieranie i ustawianie wartości
        String firstItem = list1.getFirst();
        list1.set(1, "Kokos");

        // Rozmiar listy
        int size1 = list1.size();
        int size2 = list2.size();

        // Sprawdzanie czy lista zawiera element
        boolean containsApple = list1.contains("Jabłko");

        // Iteracja przez listę
        for(String item : list1) {
            System.out.println(item);
        }
    }
}
```

`HashSet` (zbiór mieszający)

`HashSet<E>` to implementacja zbioru, która jest częścią Java Collections Framework. Jest to kolekcja, która przechowuje unikalne elementy, nie dopuszczając duplikatów.

1. **Unikalność**: `HashSet` przechowuje tylko unikalne elementy. Próba dodania duplikatu nie spowoduje błędu, ale element nie zostanie dodany.
2. **Brak kolejności**: Elementy w `HashSet` nie są przechowywane w żadnej określonej kolejności. Kolejność elementów może się różnić przy każdym uruchomieniu programu.
3. **Szybkość działania**: `HashSet` zapewnia stały czas potrzebny do operacji dodawania, usuwania i sprawdzania, czy element istnieje, co sprawia, że jest bardzo wydajny.
4. **Null**: `HashSet` pozwala na przechowywanie maksymalnie jednego elementu `null`.
5. **Implementacja**: `HashSet` jest zaimplementowany na bazie tablicy mieszającej (`HashMap`), gdzie klucze to elementy zbioru, a wartości są obiektami-pustymi miejscami.
6. **Brak gwarancji kolejności**: Kolejność elementów w `HashSet` może się zmieniać w miarę dodawania lub usuwania elementów.

Metody: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html

1. **boolean add(E e)**: Dodaje określony element do tego zbioru, jeśli jeszcze się w nim nie znajduje.
2. **void clear()**: Usuwa wszystkie elementy z tego zbioru.
3. **Object clone()**: Zwraca płytką kopię tej instancji `HashSet`: same elementy nie są klonowane.
4. **boolean contains(Object o)**: Zwraca `true`, jeśli ten zbiór zawiera określony element.
5. **boolean isEmpty()**: Zwraca `true`, jeśli ten zbiór nie zawiera żadnych elementów.
6. **Iterator iterator()**: Zwraca iterator po elementach tego zbioru.
7. **static HashSet newHashSet(int numElements)**: Tworzy nowy, pusty `HashSet` odpowiedni dla oczekiwanej liczby elementów.
8. **boolean remove(Object o)**: Usuwa określony element z tego zbioru, jeśli jest obecny.
9. **int size()**: Zwraca liczbę elementów w tym zbiorze (jego moc).
10. **Spliterator spliterator()**: Tworzy późno wiążący i szybko reagujący na błędy `Spliterator` po elementach tego zbioru.
11. **Object[] toArray()**: Zwraca tablicę zawierającą wszystkie elementy tej kolekcji.
12. **T[] toArray(T[] a)**: Zwraca tablicę zawierającą wszystkie elementy tej kolekcji; typ czasu wykonania zwróconej tablicy jest taki sam, jak określonej tablicy.

Przykład

Projekt W13, example27

```
package example27;

import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        // Tworzenie HashSet
        HashSet<String> stringSet = new HashSet<>();
        HashSet<Integer> intSet = new HashSet<>();

        // Dodawanie elementów
        stringSet.add("Jabłko");
        stringSet.add("Banana");
        intSet.add(1);
        intSet.add(2);

        // Sprawdzanie, czy zbiór zawiera element
        boolean containsBanana = stringSet.contains("Banana");

        // Usuwanie elementu
        stringSet.remove("Jabłko");

        // Sprawdzanie, czy zbiór jest pusty
        boolean isEmptyStringSet = stringSet.isEmpty();

        // Pobieranie rozmiaru zbioru
        int sizeIntSet = intSet.size();

        // Iteracja przez zbiór
        for(String s : stringSet) {
            System.out.println(s);
        }

        // Czyszczenie zbioru
        intSet.clear();
    }
}
```

`TreeSet` (zbiór drzewiasty)

`TreeSet<E>` to implementacja zbioru oparta na drzewie czerwono-czarnym, będąca częścią Java Collections Framework. Jest to posortowana i automatycznie sortująca się kolekcja unikalnych elementów.

1. **Sortowanie**: Elementy w `TreeSet` są zawsze posortowane według naturalnego porządku lub według Comparatora dostarczonego przy tworzeniu zbioru. To oznacza, że elementy są automatycznie sortowane w momencie ich dodawania.
2. **Unikalność**: Podobnie jak `HashSet`, `TreeSet` przechowuje tylko unikalne elementy, nie dopuszczając duplikatów.
3. **Wykonanie**: Dzięki strukturze drzewa czerwono-czarnego, operacje takie jak wyszukiwanie, dodawanie i usuwanie elementów mają złożoność czasową logarytmiczną O(log n), co jest bardziej efektywne niż liniowa złożoność niektórych innych struktur danych dla dużych zestawów danych.
4. **Null**: `TreeSet` zwykle nie akceptuje wartości `null` jako elementów. Próba dodania `null` może skutkować `NullPointerException`, w zależności od użytego Comparatora.
5. **Iteracja**: Iteracja przez elementy `TreeSet` odbywa się w uporządkowany sposób, co jest korzystne, gdy potrzebna jest uporządkowana prezentacja danych.

Metody: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

1. **boolean add(E e)**: Dodaje określony element do tego zbioru, jeśli jeszcze się w nim nie znajduje.
2. **boolean addAll(Collection<? extends E> c)**: Dodaje wszystkie elementy z określonej kolekcji do tego zbioru.
3. **void addFirst(E e)**: Rzuca wyjątek `UnsupportedOperationException`.
4. **void addLast(E e)**: Rzuca wyjątek `UnsupportedOperationException`.
5. **E ceiling(E e)**: Zwraca najmniejszy element w tym zbiorze większy lub równy podanemu elementowi, lub `null`, jeśli taki element nie istnieje.
6. **void clear()**: Usuwa wszystkie elementy z tego zbioru.
7. **Object clone()**: Zwraca płytką kopię tej instancji `TreeSet`.
8. **Comparator<? super E> comparator()**: Zwraca komparator używany do porządkowania elementów w tym zbiorze, lub `null`, jeśli zbiór używa naturalnego porządkowania swoich elementów.
9. **boolean contains(Object o)**: Zwraca `true`, jeśli ten zbiór zawiera określony element.

10. **Iterator descendingIterator()**: Zwraca iterator po elementach tego zbioru w porządku malejącym.
11. **NavigableSet descendingSet()**: Zwraca widok w odwrotnej kolejności elementów zawartych w tym zbiorze.
12. **E first()**: Zwraca pierwszy (najniższy) element aktualnie w tym zbiorze.
13. **E floor(E e)**: Zwraca największy element w tym zbiorze mniejszy lub równy podanemu elementowi, lub `null`, jeśli taki element nie istnieje.
14. **SortedSet headSet(E toElement)**: Zwraca widok części tego zbioru, którego elementy są ściśle mniejsze niż toElement.
15. **NavigableSet headSet(E toElement, boolean inclusive)**: Zwraca widok części tego zbioru, którego elementy są mniejsze niż (lub równe, jeśli inclusive jest `true`) toElement.
16. **E higher(E e)**: Zwraca najmniejszy element w tym zbiorze ściśle większy niż podany element, lub `null`, jeśli taki element nie istnieje.
17. **boolean isEmpty()**: Zwraca `true`, jeśli ten zbiór nie zawiera żadnych elementów.
18. **Iterator iterator()**: Zwraca iterator po elementach tego zbioru w porządku rosnącym.
19. **E last()**: Zwraca ostatni (najwyższy) element aktualnie w tym zbiorze.
20. **E lower(E e)**: Zwraca największy element w tym zbiorze ściśle mniejszy niż podany element, lub `null`, jeśli taki element nie istnieje.
21. **E pollFirst()**: Pobiera i usuwa pierwszy (najniższy) element, lub zwraca `null`, jeśli ten zbiór jest pusty.
22. **E pollLast()**: Pobiera i usuwa ostatni (najwyższy) element, lub zwraca `null`, jeśli ten zbiór jest pusty.
23. **boolean remove(Object o)**: Usuwa określony element z tego zbioru, jeśli jest obecny.
24. **int size()**: Zwraca liczbę elementów w tym zbiorze (jego moc).
25. **Spliterator spliterator()**: Tworzy późno wiążący i szybko reagujący na błędy `Spliterator` po elementach tego zbioru.
26. **NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)**: Zwraca widok części tego zbioru, którego elementy mieszczą się w zakresie od fromElement do toElement.
27. **SortedSet subSet(E fromElement, E toElement)**: Zwraca widok części tego zbioru, którego elementy mieszczą się w zakresie od fromElement, włącznie, do toElement, wyłącznie.
28. **SortedSet tailSet(E fromElement)**: Zwraca widok części tego zbioru, którego elementy są większe lub równe fromElement.

29. **NavigableSet tailSet(E fromElement, boolean inclusive)**: Zwraca widok części tego zbioru, którego elementy są większe niż (lub równe, jeśli inclusive jest `true`) fromElement.

Przykład

Projekt W13, example28

```
package example28;

import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        // Tworzenie TreeSet
        TreeSet<String> stringSet = new TreeSet<>();
        TreeSet<Integer> intSet = new TreeSet<>();

        // Dodawanie elementów
        stringSet.add("Jabłko");
        stringSet.add("Banana");
        intSet.add(1);
        intSet.add(2);

        // Sprawdzanie, czy zbiór zawiera element
        boolean containsBanana = stringSet.contains("Banana");

        // Pobieranie pierwszego i ostatniego elementu
        String firstString = stringSet.first();
        Integer lastInt = intSet.last();

        // Usuwanie elementów
        stringSet.remove("Jabłko");
        intSet.pollFirst();

        // Pobieranie podzbioru
        TreeSet<Integer> subSet = new TreeSet<>(intSet.subSet(1, true, 3, false));

        // Iteracja przez zbiór
        for(String s : stringSet) {
            System.out.println(s);
        }
```

```
        // Czyszczenie zbioru
        intSet.clear();
    }
}
```

`HashMap` (mapa mieszająca)

`HashMap<K,V>` w Javie to kolekcja, która przechowuje pary klucz-wartość. Jest to część Java Collections Framework i oferuje efektywne sposoby przechowywania i dostępu do danych.

1. **Struktura klucz-wartość**: Każdy element w `HashMap` składa się z klucza (unikalnego identyfikatora) i przypisanej mu wartości.
2. **Szybki dostęp do danych**: Dzięki funkcji mieszającej (hashing), `HashMap` pozwala na bardzo szybkie wyszukiwanie wartości na podstawie klucza. Operacje takie jak wstawianie, wyszukiwanie lub usuwanie elementów mają zazwyczaj stałą złożoność czasową O(1).
3. **Unikalność kluczy**: W `HashMap` każdy klucz musi być unikalny. Próba wstawienia duplikatu klucza spowoduje nadpisanie starej wartości przez nową.
4. **Null jako klucz i wartość**: `HashMap` pozwala na użycie `null` jako klucza (ale tylko raz, ponieważ klucze są unikalne) oraz `null` jako wartości.
5. **Nieuporządkowana**: Kolejność przechowywania par klucz-wartość w `HashMap` nie jest gwarantowana i może się zmieniać z czasem, zwłaszcza po operacjach, które zmieniają rozmiar mapy, np. po dodaniu nowych elementów.

Metody: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html

1. **void clear()**: Usuwa wszystkie mapowania z tej mapy.
2. **Object clone()**: Zwraca płytką kopię tej instancji `HashMap`: klucze i wartości same w sobie nie są klonowane.
3. **V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)**: Próbuje obliczyć mapowanie dla określonego klucza i jego aktualnie mapowanej wartości (lub `null`, jeśli mapowanie nie istnieje).
4. **V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)**: Jeśli określony klucz nie jest jeszcze skojarzony z wartością (lub jest mapowany na `null`), próbuje obliczyć jego wartość za pomocą podanej funkcji mapującej i wprowadza ją do tej mapy, chyba że wynik jest `null`.

5. **V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)**: Jeśli wartość dla określonego klucza jest obecna i nie jest `null`, próbuje obliczyć nowe mapowanie na podstawie klucza i jego aktualnie mapowanej wartości.
6. **boolean containsKey(Object key)**: Zwraca `true`, jeśli ta mapa zawiera mapowanie dla określonego klucza.
7. **boolean containsValue(Object value)**: Zwraca `true`, jeśli ta mapa mapuje jeden lub więcej kluczy na określoną wartość.
8. **Set<Map.Entry<K,V» entrySet()**: Zwraca widok zbioru mapowań zawartych w tej mapie.
9. **V get(Object key)**: Zwraca wartość, do której mapowany jest określony klucz, lub `null`, jeśli ta mapa nie zawiera mapowania dla klucza.
10. **boolean isEmpty()**: Zwraca `true`, jeśli ta mapa nie zawiera żadnych mapowań klucz-wartość.
11. **Set keySet()**: Zwraca widok zbioru kluczy zawartych w tej mapie.
12. **V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)**: Jeśli określony klucz nie jest już skojarzony z wartością lub jest skojarzony z `null`, kojarzy go z podaną wartością niebędącą `null`.
13. **static <K,V> HashMap<K,V> newHashMap(int numMappings)**: Tworzy nową, pustą `HashMap` odpowiednią dla oczekiwanej liczby mapowań.
14. **V put(K key, V value)**: Kojarzy określoną wartość z określonym kluczem w tej mapie.
15. **void putAll(Map<? extends K,? extends V> m)**: Kopiuje wszystkie mapowania z określonej mapy do tej mapy.
16. **V remove(Object key)**: Usuwa mapowanie dla określonego klucza z tej mapy, jeśli jest obecne.
17. **int size()**: Zwraca liczbę mapowań klucz-wartość w tej mapie.
18. **Collection values()**: Zwraca widok kolekcji wartości zawartych w tej mapie.

Przykład

Projekt W13, example29

```java
package example29;

import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        // Tworzenie HashMap
        HashMap<String, Integer> fruitMap = new HashMap<>();
        HashMap<Integer, String> reverseMap = new HashMap<>();

        // Dodawanie elementów
        fruitMap.put("Jabłko", 10);
        fruitMap.put("Banana", 20);
        reverseMap.put(1, "Jeden");
        reverseMap.put(2, "Dwa");

        // Pobieranie wartości
        Integer appleCount = fruitMap.get("Jabłko");
        String numberTwo = reverseMap.get(2);

        // Sprawdzanie obecności klucza lub wartości
        boolean hasBanana = fruitMap.containsKey("Banana");
        boolean hasTwenty = reverseMap.containsValue("Dwa");

        // Usuwanie elementu
        fruitMap.remove("Banana");

        // Iteracja przez mapę
        for (String key : fruitMap.keySet()) {
            System.out.println(key + " -> " + fruitMap.get(key));
        }

        // Pobieranie liczby elementów
        int size = reverseMap.size();

        // Czyszczenie mapy
        reverseMap.clear();
    }
}
```

Projekt W13, example30

```java
package example30;

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

public class HashMapEntrySetExample {
    public static void main(String[] args) {
        // Pierwsza HashMap
        HashMap<String, Integer> prices = new HashMap<>();
        prices.put("Apple", 50);
        prices.put("Orange", 30);
        prices.put("Banana", 20);

        // Druga HashMap
        HashMap<Integer, String> daysOfWeek = new HashMap<>();
        daysOfWeek.put(1, "Monday");
        daysOfWeek.put(2, "Tuesday");
        daysOfWeek.put(3, "Wednesday");

        // Wyświetlanie używając entrySet() dla pierwszej mapy
        for (Map.Entry<String, Integer> entry : prices.entrySet()) {
            System.out.println(entry.getKey() + " costs " + entry.getValue());
        }

        // Wyświetlanie używając entrySet() dla drugiej mapy
        for (Map.Entry<Integer, String> entry : daysOfWeek.entrySet()) {
            System.out.println("Day " + entry.getKey() + " is " + entry.getValue());
        }
    }
}
```

`TreeMap` (mapa drzewiasta)

`TreeMap<K,V>` w Javie to posortowana mapa bazująca na czerwono-czarnym drzewie. Jest to część Java Collections Framework i działa jako mapa klucz-wartość, gdzie każdy klucz jest unikalny.

1. **Automatyczne sortowanie**: Klucze w `TreeMap` są sortowane według naturalnego porządku (jeśli implementują interfejs `Comparable`) lub według `Comparatora` przekazanego przy tworzeniu mapy. Dzięki temu, kiedy iterujesz po kluczach, są one zwracane w posortowanej kolejności.

2. **Wykonanie**: Operacje wyszukiwania, wstawiania i usuwania mają logarytmiczną złożoność czasową O(log n), co sprawia, że `TreeMap` jest wydajna dla dużych zbiorów danych.
3. **Null jako klucz**: Standardowo `TreeMap` nie akceptuje `null` jako klucza (w przeciwieństwie do `HashMap`), szczególnie gdy używa naturalnego porządkowania, ponieważ `null` nie może być porównywany.
4. **Dostęp do pierwszego i ostatniego elementu**: `TreeMap` zapewnia szybki dostęp do pierwszego (najmniejszego) i ostatniego (największego) klucza.
5. **Widoki**: `TreeMap` oferuje metody takie jak `headMap`, `tailMap` i `subMap`, które pozwalają na tworzenie widoków określonych zakresów mapy.

Metody: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html

1. **Map.Entry<K,V> ceilingEntry(K key)**: Zwraca mapowanie klucz-wartość związane z najmniejszym kluczem większym lub równym podanemu kluczowi, lub `null`, jeśli taki klucz nie istnieje.
2. **K ceilingKey(K key)**: Zwraca najmniejszy klucz większy lub równy podanemu kluczowi, lub `null`, jeśli taki klucz nie istnieje.
3. **void clear()**: Usuwa wszystkie mapowania z tej mapy.
4. **Object clone()**: Zwraca płytką kopię tej instancji `TreeMap`.
5. **Comparator<? super K> comparator()**: Zwraca komparator używany do porządkowania kluczy w tej mapie, lub `null`, jeśli mapa używa naturalnego porządkowania swoich kluczy.
6. **V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)**: Próbuje obliczyć mapowanie dla określonego klucza i jego aktualnie mapowanej wartości (lub `null`, jeśli mapowanie nie istnieje).
7. **V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)**: Jeśli określony klucz nie jest już skojarzony z wartością (lub jest mapowany na `null`), próbuje obliczyć jego wartość za pomocą podanej funkcji mapującej i wprowadza ją do tej mapy, chyba że wynik jest `null`.
8. **V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)**: Jeśli wartość dla określonego klucza jest obecna i nie jest `null`, próbuje obliczyć nowe mapowanie na podstawie klucza i jego aktualnie mapowanej wartości.
9. **boolean containsKey(Object key)**: Zwraca `true`, jeśli ta mapa zawiera mapowanie dla określonego klucza.

10. **boolean containsValue(Object value)**: Zwraca `true`, jeśli ta mapa mapuje jeden lub więcej kluczy na określoną wartość.
11. **NavigableSet descendingKeySet()**: Zwraca widok w odwrotnej kolejności `NavigableSet` kluczy zawartych w tej mapie.
12. **NavigableMap<K,V> descendingMap()**: Zwraca widok w odwrotnej kolejności mapowań zawartych w tej mapie.
13. **Set<Map.Entry<K,V» entrySet()**: Zwraca widok zbioru mapowań zawartych w tej mapie.
14. **Map.Entry<K,V> firstEntry()**: Zwraca mapowanie klucz-wartość związane z najmniejszym kluczem w tej mapie, lub `null`, jeśli mapa jest pusta.
15. **K firstKey()**: Zwraca pierwszy (najniższy) klucz obecnie w tej mapie.
16. **Map.Entry<K,V> floorEntry(K key)**: Zwraca mapowanie klucz-wartość związane z największym kluczem mniejszym lub równym podanemu kluczowi, lub `null`, jeśli taki klucz nie istnieje.
17. **K floorKey(K key)**: Zwraca największy klucz mniejszy lub równy podanemu kluczowi, lub `null`, jeśli taki klucz nie istnieje.
18. **V get(Object key)**: Zwraca wartość, do której mapowany jest określony klucz, lub `null`, jeśli ta mapa nie zawiera mapowania dla klucza.
19. **SortedMap<K,V> headMap(K toKey)**: Zwraca widok części tej mapy, której klucze są ściśle mniejsze niż toKey.
20. **NavigableMap<K,V> headMap(K toKey, boolean inclusive)**: Zwraca widok części tej mapy, której klucze są mniejsze niż (lub równe, jeśli inclusive jest `true`) toKey.
21. **Map.Entry<K,V> higherEntry(K key)**: Zwraca map

owanie klucz-wartość związane z najmniejszym kluczem ściśle większym niż podany klucz, lub `null`, jeśli taki klucz nie istnieje.

22. **K higherKey(K key)**: Zwraca najmniejszy klucz ściśle większy niż podany klucz, lub `null`, jeśli taki klucz nie istnieje.
23. **Set keySet()**: Zwraca widok zbioru kluczy zawartych w tej mapie.
24. **Map.Entry<K,V> lastEntry()**: Zwraca mapowanie klucz-wartość związane z największym kluczem w tej mapie, lub `null`, jeśli mapa jest pusta.
25. **K lastKey()**: Zwraca ostatni (najwyższy) klucz obecnie w tej mapie.
26. **Map.Entry<K,V> lowerEntry(K key)**: Zwraca mapowanie klucz-wartość związane z największym kluczem ściśle mniejszym niż podany klucz, lub `null`, jeśli taki klucz nie istnieje.

27. **K lowerKey(K key)**: Zwraca największy klucz ściśle mniejszy niż podany klucz, lub `null`, jeśli taki klucz nie istnieje.
28. **V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)**: Jeśli określony klucz nie jest już skojarzony z wartością lub jest skojarzony z `null`, kojarzy go z podaną wartością niebędącą `null`.
29. **NavigableSet navigableKeySet()**: Zwraca widok `NavigableSet` kluczy zawartych w tej mapie.
30. **Map.Entry<K,V> pollFirstEntry()**: Usuwa i zwraca mapowanie klucz-wartość związane z najmniejszym kluczem w tej mapie, lub `null`, jeśli mapa jest pusta.
31. **Map.Entry<K,V> pollLastEntry()**: Usuwa i zwraca mapowanie klucz-wartość związane z największym kluczem w tej mapie, lub `null`, jeśli mapa jest pusta.
32. **V put(K key, V value)**: Kojarzy określoną wartość z określonym kluczem w tej mapie.
33. **void putAll(Map<? extends K,? extends V> map)**: Kopiuje wszystkie mapowania z określonej mapy do tej mapy.
34. **V putFirst(K k, V v)**: Rzuca wyjątek `UnsupportedOperationException`.
35. **V putLast(K k, V v)**: Rzuca wyjątek `UnsupportedOperationException`.
36. **V remove(Object key)**: Usuwa mapowanie dla tego klucza z tej mapy `TreeMap`, jeśli jest obecne.
37. **int size()**: Zwraca liczbę mapowań klucz-wartość w tej mapie.
38. **NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)**: Zwraca widok części tej mapy, której klucze mieszczą się w zakresie od fromKey do toKey.
39. **SortedMap<K,V> subMap(K fromKey, K toKey)**: Zwraca widok części tej mapy, której klucze mieszczą się w zakresie od fromKey, włącznie, do toKey, wyłącznie.
40. **SortedMap<K,V> tailMap(K fromKey)**: Zwraca widok części tej mapy, której klucze są większe lub równe fromKey.
41. **NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)**: Zwraca widok części tej mapy, której klucze są większe niż (lub równe, jeśli inclusive jest `true`) fromKey.
42. **Collection values()**: Zwraca widok kolekcji wartości zawartych w tej mapie.

Przykład

Projekt W13, example31

```java
package example31;

import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        // Tworzenie TreeMap
        TreeMap<String, Integer> fruitMap = new TreeMap<>();
        TreeMap<Integer, String> reverseMap = new TreeMap<>();

        // Dodawanie elementów
        fruitMap.put("Jabłko", 50);
        fruitMap.put("Banana", 20);
        fruitMap.put("Pomarańcza", 30);
        reverseMap.put(1, "Poniedziałek");
        reverseMap.put(2, "Wtorek");
        reverseMap.put(3, "Środa");

        // Pobieranie pierwszego i ostatniego klucza
        String firstKey = fruitMap.firstKey();
        Integer lastKey = reverseMap.lastKey();

        // Sprawdzanie obecności klucza
        boolean hasApple = fruitMap.containsKey("Jabłko");

        // Usuwanie elementu
        fruitMap.remove("Banana");

        // Pobieranie i ustawianie wartości
        Integer appleCount = fruitMap.get("Jabłko");
        reverseMap.put(3, "Środa");

        // Iteracja przez mapę
        for (String key : fruitMap.keySet()) {
            System.out.println(key + " -> " + fruitMap.get(key));
        }

        // Pobieranie podmapy
        TreeMap<String, Integer> subMap = new TreeMap<>(fruitMap.subMap("Jabłko", true, "Poma
```

```
        // Czyszczenie mapy
        reverseMap.clear();
    }
}
```

Projekt W13, example32

```
package example32;

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

public class TreeMapExample32 {
    public static void main(String[] args) {
        TreeMap<String, Double> prices = new TreeMap<>();

        // Dodawanie par klucz-wartość
        prices.put("Apple", 1.99);
        prices.put("Banana", 0.49);
        prices.put("Cherry", 2.99);
        prices.put("Date", 3.49);
        prices.put("Elderberry", 1.79);
        prices.put("Fig", 2.09);

        // Użycie metody ceilingEntry
        Map.Entry<String, Double> ceiling = prices.ceilingEntry("Cantaloupe");

        // Użycie metody firstEntry
        Map.Entry<String, Double> first = prices.firstEntry();

        // Użycie metody higherEntry
        Map.Entry<String, Double> higher = prices.higherEntry("Cherry");

        // Użycie metody lowerEntry
        Map.Entry<String, Double> lower = prices.lowerEntry("Date");

        // Wyświetlanie wyników
        System.out.println("Ceiling Entry: " + ceiling.getKey() + " -> " + ceiling.getValue()
        System.out.println("First Entry: " + first.getKey() + " -> " + first.getValue());
        System.out.println("Higher Entry: " + higher.getKey() + " -> " + higher.getValue());
        System.out.println("Lower Entry: " + lower.getKey() + " -> " + lower.getValue());
```

```
    }
}
```

Algorytmy generyczne - klasa `Collections`

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html

Klasa `Collections` w Javie pełni rolę narzędzia pomocniczego (utility class) dla kolekcji w Java Collections Framework. Zawiera ona statyczne metody, które działają na lub zwracają kolekcje. Jest to klasa czysto statyczna, co oznacza, że nie jest przeznaczona do tworzenia instancji (obiektów), lecz służy jako zbiór narzędzi dla różnych operacji na kolekcjach.