PROGRAMOWANIE OBIEKTOWE

Kolekcje

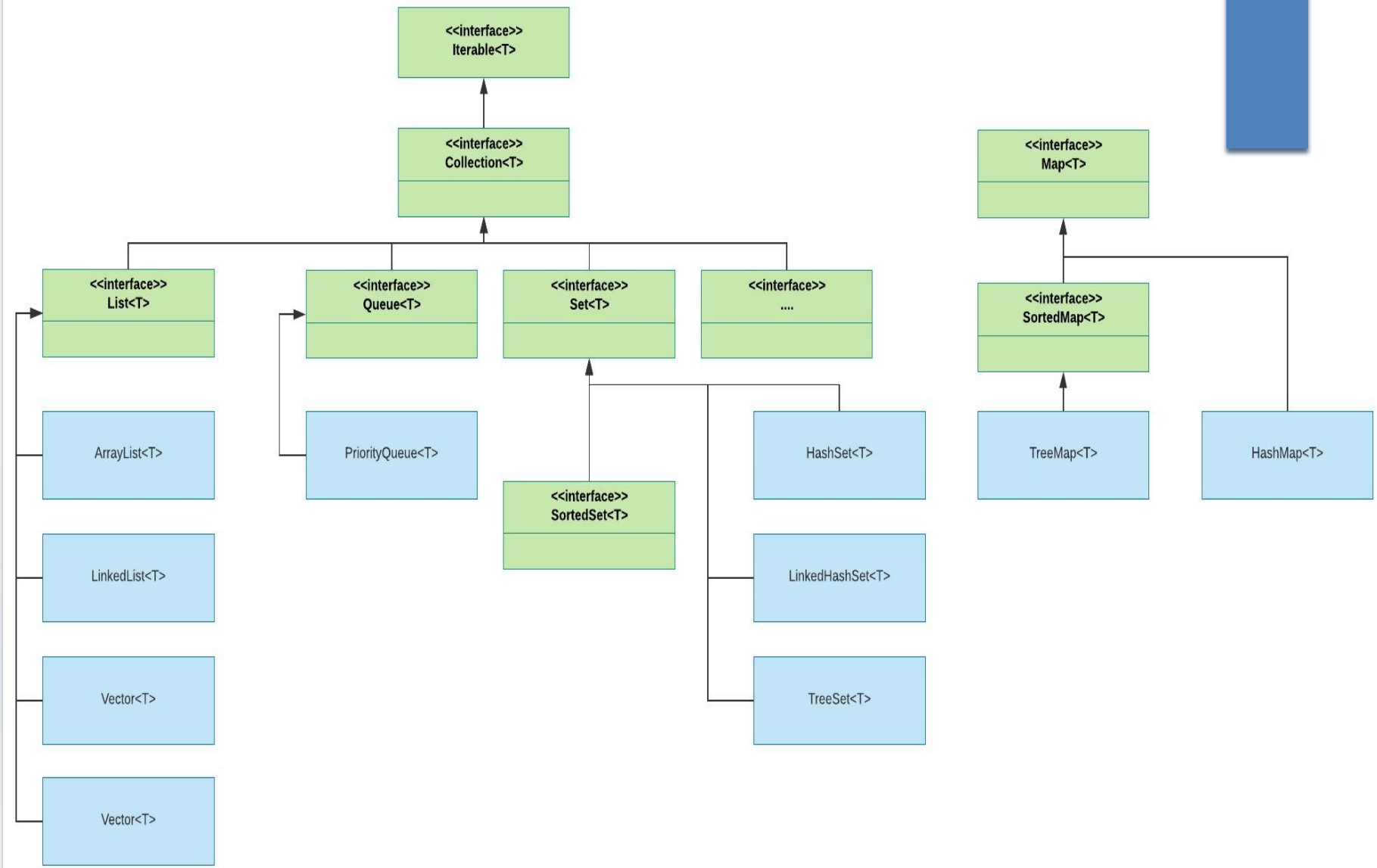
dr inż. Barbara Fryc

KOLEKCJE W JAVA

Kolekcja (kontener) – obiekt który grupuje wiele elementów w jedną całość. Służą do przechowywania i manipulowania danymi

Framework *Collections* w java posiada zunifikowaną architekturę do reprezentowania i manipulowania kolekcjami. Zawiera:

- Interfejsy
- Implementacje
- Algorytmy



STANDARDOWE KOLEKCJE

PORÓWNANIE KOLEKCJI

Lista:

- Uporządkowana – kontrola na jakiej pozycji element się znajduje
- Dostęp po indeksach
- Zezwala na duplikaty
- Szybkie wstawianie i usuwanie w dowolnym miejscu
- Wolne wyszukiwanie

Kolejka:

- Dodawanie tylko na początku
- Pobieranie elementów tylko z końca
- Pozwala (najczęściej) na zduplikowane elementy

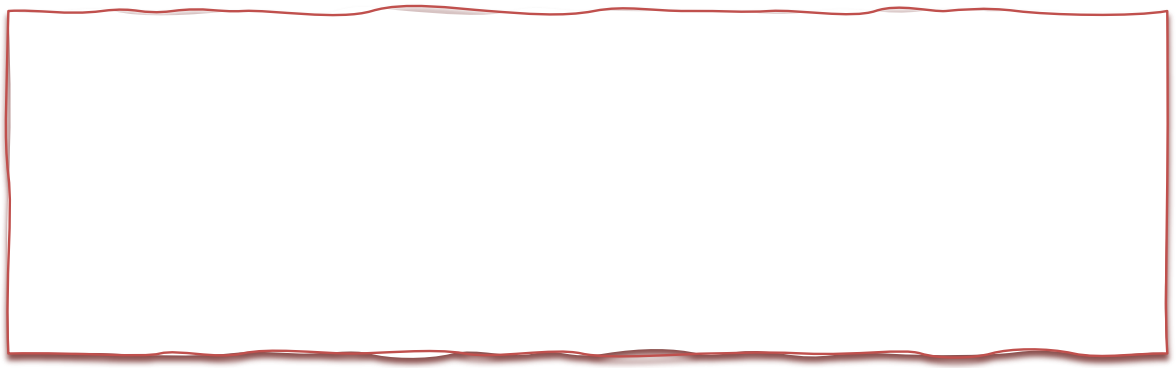
Zbiór:

- Nie pozwala na zduplikowane elementy
- Pozwala szybko stwierdzić, czy element jest w kolekcji

Porównanie kolekcji CD

Mapa:

- Tworzy uporządkowanie klucz -> wartość
- Szybkie wyszukiwanie po kluczu



PRZYKŁAD LISTY - ARRAYLIST

Collections:

- add(E e)
- addAll(Collection<? Extends E> c)
- clear()
- contains(Object o)
- isEmpty()
- remove(Object o)
- size()

ArrayList:

- add(int index, E e)
- addAll(int idx, Collection<> c)
- get(int idx)
- indexOf(Object o)
- remove(int idx)
- sublist(int fromIdx, int toIdx)

ALGORYTMY

- Sortowanie
- Mieszanie
- Wyszukiwanie
- Kompozycja
- Wyszukiwanie ekstremów
- Odwrócenie kolejności
- Wypełnienie listy
- Kopiowanie fragmentów listy
- Zamiana dwóch elementów

Większość algorytmów wyłącznie dla list!

PRZYKŁADY ALG. - SORTOWANIE

```
public static void main(String[] args) {
    ArrayList<Integer> aint = new ArrayList<>(Arrays.asList(2,8,3,12,56,19,4,22));
    Collections.sort(aint);
    for (Integer i : aint) {
        System.out.print(i+" ");
    }
}
```

W przyp
naturalny
Java) nale
Comparato

```
@FunctionalInterface
public interface Comparator<T> {
    /** Compares its two arguments for order.  Returns a negative integ
    @Contract(pure = true)
    int compare(T o1, T o2);
```

REFERENCJE METOD

Referencje metod umożliwiają przekazywanie metod jako parametrów funkcji.

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
        "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

Reference to a static method	ContainingClass::staticMethodName
Reference to an instance method of a particular object	containingObject::instanceMethodName
Reference to an instance method of an arbitrary object of a particular type	ContainingType::methodName
Reference to a constructor	ClassName::new

INTERFEJS LIST

- ArrayList
- LinkedList
- AbstractList
- AbstractSequentialList
- AttributeList
- CopyOnWriteArrayList
- RoleList
- RoleUnresolvedList
- Stack
- Vector

LINKEDLIST VS ARRAYLIST

- LinkedList oznacza, że są w**ykorzystane powiązania między elementami**
- ArrayList zaś informuje, że **jest wykorzystana tablica**.
- W ArrayList dostęp do danych jest natychmiastowy tzn. O(1), ponieważ **używamy indeksów**.
- W przypadku LinkedList, gdzie elementy są powiązane ze sobą (każdy zawiera informację tylko o poprzednim i następnym elemencie w liście), aby znaleźć element złożoność obliczeniowa jest równa O(n). W przypadku, gdy element jest na początku listy wszystko będzie ok, co jeśli wyszukiwany element będzie 100000 elementem w liście?
- Usuwanie szybciej wykonuje to LinkedList. Wystarczy jej O(1), aby usunąć dany element, ponieważ do **usunięcia wystarczy zmienić dwa adresy.**

LINKEDLIST VS ARRAYLIST

- Wnioski są proste, powinniśmy wykorzystywać LinkedList do list, których **operacją dominującą będzie wstawianie oraz usuwanie elementów**. Wtedy nawet podczas miliona operacji wstawiania/usuwania aplikacja będzie miała możliwość działać szybciej.

- Jednak w przypadku, gdy mamy listę, którą **bardzo często przeszukujemy warto wybrać ArrayList**. Dzięki zastosowaniu tablic mamy dostęp do danych O(1) – czyli błyskawicznie.

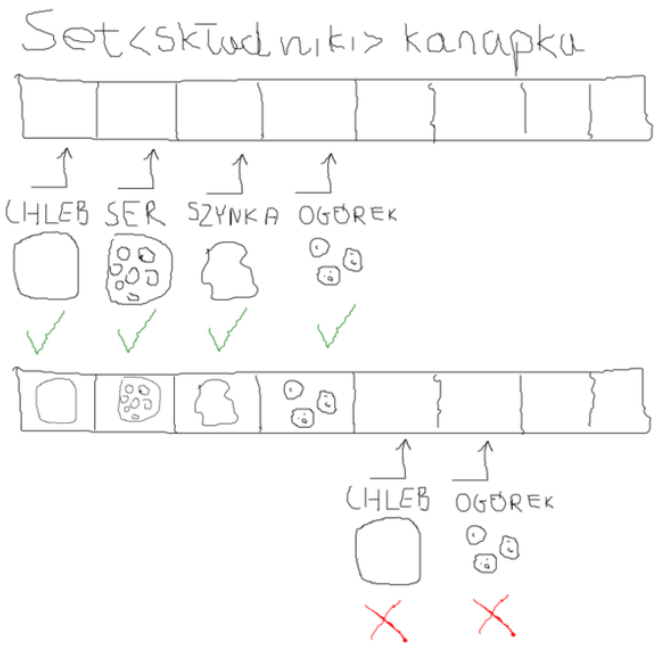
ARRAYLIST - PRZYKŁADY

Create arraylist

```
//Non-generic arraylist - NOT RECOMMENDED !!
ArrayList list = new ArrayList();

//Generic Arraylist with default capacity
List<Integer> numbers = new ArrayList<>();

//Generic Arraylist with the given capacity
List<Integer> numbers = new ArrayList<>(6);

//Generic Arraylist initialized with another collection
List<Integer> numbers = new ArrayList<>( Arrays.asList(1,2,3,4,5) );
```

INTERFEJS SET

- AbstractSet
- ConcurrentSkipListSet
- CopyOnWriteArraySet
- EnumSet
- JobStateReasons
- LinkedHashSet
- **HashSet**
- **TreeSet – przechowuje elementy posortowane**



INTERFEJS MAP

HashMap

- Prosta implementacja mapy, zachowuje się podobnie jak wszystkie implementacje z przedrostkiem **Hash**.
- Elementy **nie są posortowane**
- Nie mamy pewności, że elementy są ułożone w kolejności ich dodawania
- Złożoność obliczeniowa podstawowych operacji to O(1) **(get, put)**
- Pozwala wprowadzić **null** jako klucz i wartość

Treemap

- Podobnie jak wszystkie implementację z przedrostkiem **Tree** są zbudowane na drzewie binarnym i zachowują się podobnie:
- Elementy **są posortowane** (musimy zdefiniować nasz comparator za pomocą interfejsów Comparator<T> lub Comparable<T>)
- Operacje dodawanie i wyszukiwania mają złożoność obliczeniową O(ln(n)).

INTERFEJS MAP

EnumMAP

- Osobiście tej implementacji nigdy nie używałem, jednak jest bardzo specyficzna – **kluczami w tej mapie mogą być tylko Enumeratory.**
- Uniemożliwia wstawienie **null** jako klucza
- Wartości są przechowywane w kolejności ich dodawania
- Kluczem może być tylko enumerator

LinkedhashMap

- Tak samo jak w przypadku listy występuje przedrostek **Linked**, czyli już zapala nam się w głowie, że elementy są w jakiś sposób ze sobą połączone. Dokładnie LinkedHashMap jest zaimplementowana na podstawi HashTable oraz LinkedList, jakie są z tego korzyści?
- Elementy są przechowywane w kolejności jak były dodawane
- Złożoność podstawowych operacji O(1)
- Możliwość wstawiania **nulla**

TYPY PARAMETRYZOWNE (GENERICS)

```java
package pl.wsiz.lectures;

public class Pair<T,S> {
    private T first;
    private S second;

    public Pair(T first, S second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public S getSecond() {
        return second;
    }
}
```

Typy parametryzowane umożliwiają tworzenie szablonów typów/interfejsów, w których typy klas/metod/parametrów zastąpione są parametrami.

Podstawową zaletą typów parametryzowanych jest praca z typem danych, który został przekazany.

```java
public static void main(String[] args) {
    Pair<Integer, String> pair = new Pair<>(2, "Wykład");
    String nazwa = pair.getSecond();
}
```

TYPY PARAMETRYZOWNE (GENERICS)

```java
public class PairOld {
    private Object first;
    private Object second;
```

VS

```java
public class Pair<T,S> {
    private T first;
    private S second;
```

```java
PairOld pairOld =
        new PairOld(1, "Wyklad 2");
String nazwa2 =
        (String) pairOld.getSecond();
```

```java
Pair<Integer, String> pair =
        new Pair<>(2, "Wykład");
String nazwa = pair.getSecond();
```