

Java Collections Framework

Zpracováno s využitím The Java™
Tutorials od firmy ORACLE

Trail: Collections

Java Collections Framework

- Úvod
- Interfejsy
- Implementace
- Algoritmy
- Vlastní implementace
- Interoperabilita

Úvod

- Kolekce (collection or container) je objekt, který sdružuje více elementů do jedné jednotky.
- Java Collections Framework je jednotná architektura pro reprezentaci a práci s kolekcemi, která obsahuje:
 - Interfejsy
 - Implementace
 - Algoritmy
- Výhody Java Collections Framework

Interfejsy

- `java.util.Comparator<T>`
- `java.util.Enumeration<E>`
- `java.lang.Iterable<T>`
 - `java.util.Collection<E>`
 - `java.util.List<E>`
 - `java.util.Queue<E>`
 - `java.util.Deque<E>`
 - `java.util.Set<E>`
 - `java.util.SortedSet<E>`
 - » `java.util.NavigableSet<E>`
- `java.util.Iterator<E>`
 - `java.util.ListIterator<E>`
- `java.util.Map<K,V>`
 - `java.util.SortedMap<K,V>`
 - `java.util.NavigableMap<K,V>`
- `java.util.Map.Entry<K,V>`

Interfejs Collection<E>

- Kolekce reprezentuje iterovatelnou skupinu objektů(elementů) typu E.
- Interfejs Collection<E> obsahuje metody společné pro všechny typy kolekcí.
- Některé typy kolekcí povolují duplicitu, jiné nikoliv.
- Některé typy kolekcí jsou uspořádané, jiné nikoliv.

Interfejs Collection<E>

- Implementace obsahují pro obecné použití konverzní konstruktory.
- Java 8 rozšiřuje interfejs o defaultní metody `parallelStream()` a `stream()` pomocí nichž lze provádět agregované operace.
- Traversing operace lze provádět:
 - s použitím agregovaných operací
 - s použitím cyklu for-each
 - s použitím iterátorů

Interfejs Collection<E>

- Interface Iterator<E> umožňuje iterování přes prvky kolekce.
- Použití defaultní metody `remove()` je jediný bezpečný způsob odstranění prvku z kolekce.

Příklad polymorfní metody

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

Interfejs Collection<E>

- Hromadné (bulk) operace (operations) jsou operace s celou kolekcí.
- Jedná se o tyto metody: **containsAll**, **addAll**, **removeAll**, **retainAll**, **clear**.
- Příklad konverze kolekce c na pole
`Object[] a = c.toArray();`
`String[] a = c.toArray(new String[0]);`

Interfejs **Set<E>**

- Instance typu **Set** je kolekce, která nemůže obsahovat duplicity prvků.
- Interface **Set** obsahuje pouze metody zděděné z **Collection**, ale přidává omezení, že duplicita elementů není možná.
- Interface **Set** přidává silnější kontrakt i na metody **equals** a **hashCode**.
- Dvě instance typu **Set** se rovnají, právě když mají stejné prvky.

Interfejs Set<E>

- Existují tři implementace interfejsu Set<E>:
 - HashSet<E>
 - TreeSet<E>
 - LinkedHashSet<E>
- `Collection<Type> noDups = new HashSet<Type>(c);`
- ```
public static <E> Set<E>
removeDups(Collection<E> c)
{ return new HashSet<E>(c); }
```

# Interfejs Set<E>

```
import java.util.*;

public class FindDups {
 public static void main(String[] args) {
 Set<String> s = new HashSet<String>();
 for (String a : args)
 s.add(a);
 System.out.println(s.size() + " distinct words:
" + s);
 }
}
```

# Interfejs Set<E>

- Necht' s1 a s2 jsou instance typu **Set**. Interfejs **Set<E>** nabízí tyto hromadné operace:
  - `s1.containsAll(s2)`
  - `s1.addAll(s2)`
  - `s1.retainAll(s2)`
  - `s1.removeAll(s2)`

# Interfejs Set<E>

```
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)) // add vrací true, pokud se prvek vloží
 dups.add(a);

 // Destructive set-difference
 uniques.removeAll(dups); // vypustení všech prvků s duplicitou

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

# Interfejs List<E>

- **List** je uspořádaná kolekce (insertion order), které může obsahovat duplicity prvků.
- Obsahuje navíc metody pro:
  - Poziční přístup – **get**, **set**, **add**, **addAll**, a **remove**.
  - Vyhledávání – **indexOf**, **lastIndex**
  - Iterování – **listIterator** – obousměrný
  - Práci s částí seznamu – **subList**

# Interfejs List<E>

- Existují dvě obecně použitelné implementace:
  - **ArrayList<E>**
  - **LinkedList<E>**
  - **Vector<E>**
- Metoda **remove** odstraní vždy první výskyt specifikovaného prvku a metody **add** a **addAll** vždy přidávají nový prvek(y) na konec.
- Dvě instance typu **List** se rovnají právě když obsahují tytéž prvky a ve stejném pořadí.

# Interfejs List<E>

- Příklad metody, která vymění dva prvky seznamu.

```
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);
}
```



# Interfejs List<E>

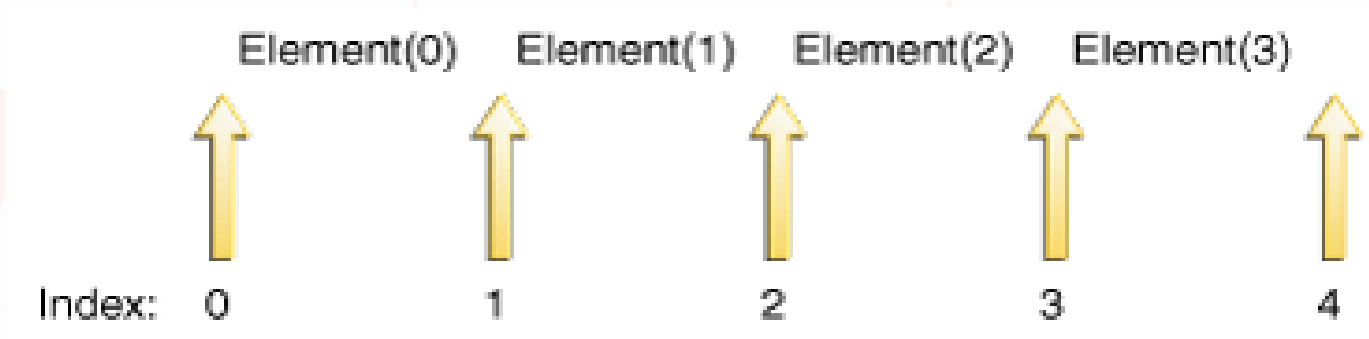
- Následující metoda provede náhodnou permutaci prvků seznamu.

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

- Tento algoritmus je obsažen v metodě ve třídě [Collections](#).

# Interfejs List<E>

- Interfejs **List<E>** nabízí metodu **listIterator()**, která vrací entitu typu **ListIterator<E>**, což je rozšíření obecnějšího iterátoru typu **Iterator<E>**.



# Interfejs List<E>

Příklady použití metody **listIterator()**

Iterování přes prvky seznamu od konce.

```
for (ListIterator<Type> it =
 list.listIterator(list.size()); it.hasPrevious();) {
 Type t = it.previous();
 ...
}
```

# Interfejs List<E>

## Příklady použití metody `listIterator()`

Metoda **replace** provede náhradu všech výskytů hodnoty **val** seznamem prvků **newVals**. Metoda **add** vkládá nový prvek před aktuální pozici kurzoru.

```
public static <E>
void replace(List<E> list, E val, List<? extends E> newVals) {
 for (ListIterator<E> it = list.listIterator(); it.hasNext();){
 if (val == null ? it.next() == null :
val.equals(it.next())) {
 it.remove();
 for (E e : newVals)
 it.add(e);
 }
 }
}
```

# Interfejs List<E>

## Příklady použití metody listIterator()

- Metoda **replace** nahradí všechny výskyty specifikované hodnoty jinou hodnotou.

```
public static <E> void replace(List<E> list, E val, E newVal) {
 for (ListIterator<E> it = list.listIterator(); it.hasNext();)
 if (val == null ? it.next() == null : val.equals(it.next()))
 it.set(newVal);
}
```

# Interfejs List<E>

- Program Deal provádí rozdání balíčku karet hráčům.
  - metoda rozdává jednomu hráči
  - n je počet karet na hráče
  - subList vytváří pouze view!

```
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(); //odstranění rozdanych karet z balíčku
 return hand;
}
```

# Interfejs Queue<E>

- Interfejs **Queue<E>** rozšiřuje interfejs **Collection<E>**

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

# Interfejs Queue<E>

| Type of Operation | Throws exception | Returns special value |
|-------------------|------------------|-----------------------|
| Insert            | add(e)           | offer(e) - boolean    |
| Remove            | remove()         | poll() - element      |
| Examine           | element()        | peek() - element      |



# Interfejs Queue<E>

- Prvky fronty jsou typicky řazeny způsobem FIFO (first-in-first-out).
- Prioritní fronty řadí prvky podle jejich priority.
- Konkrétní implementace může omezit počet prvků ve frontě (viz `java.util.concurrent`).
- Třída **LinkedList<E>** implementuje interfejs **Queue<E>**.
- Třída **PriorityQueue<E>** je implementace prioritní fronty, která využívá datovou strukturu heap.
  - přirozené třídění nebo uživatelské pomocí zadaného **Comparatoru**

# Interfejs Queue<E>

- Metoda heapSort je příkladem řadícího algoritmu, který využívá prioritní frontu.

```
static <E> List<E> heapSort(Collection<E> c) {
 Queue<E> queue = new PriorityQueue<E>(c);
 List<E> result = new ArrayList<E>();
 while (!queue.isEmpty())
 result.add(queue.remove());
 return result;
}
```

# Interfejs Deque<E>

- Deque je linární kolekce, která podporuje vkládání a odebírání prvků na obou koncích.

| Type of Operation | First Element (Beginning of the Deque instance) | Last Element (End of the Deque instance) |
|-------------------|-------------------------------------------------|------------------------------------------|
| <b>Insert</b>     | addFirst(e)<br>offerFirst(e)                    | addLast(e)<br>offerLast(e)               |
| <b>Remove</b>     | removeFirst()<br>pollFirst()                    | removeLast()<br>pollLast()               |
| <b>Examine</b>    | getFirst()<br>peekFirst()                       | getLast()<br>peekLast()                  |

# Interfejs Deque<E>

- Interfejs `Deque<E>` je implementován ve třídách `ArrayDeque` a `LinkedList`.

# Interfejs Map<K, V>

- Objekt typu **Map<K, V>** mapuje (zobrazuje) klíče typu K na hodnoty typu V.
- Nesmí obsahovat duplicitní klíče.
- Interfejs **Map<K, V>** obsahuje základní metody (**put**, **get**, **remove**, ...), metody pro hromadné operace (**putAll**, **clear**, ...) a metody pro konverzi na kolekce (**keySet**, **entrySet**, **values**).

# Interfejs Map<K,V>

- Interfejs **Map<K , V>** je implementován ve třídách HashMap<K , V>, TreeMap<K , V> a LinkedHashMap<K , V>.
- Java 8 nabízí možnost použít agregované operace pro vytvoření objektů typu **Map<K , V>**.

```
Map<Boolean, List<Student>> passingFailing = students.stream()
 .collect(Collectors.partitioningBy(s -> s.getGrade() >=
 PASS_THRESHOLD));
```

# Interfejs Map<K,V>

- Základní metody: **put**, **get**, **containsKey**, **containsValue**, **size**, **isEmpty**.
- Program pro výpočet frekvenční tabulky slov ze seznamu argumentů.
- Po spuštění programu  
java Freq if it is to be it is up to me to delegate
- se vypíše  
8 distinct words: {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

# Program Freq

```
public class Freq {

 public static void main(String[] args) {
 Map<String, Integer> m = new HashMap<>();
 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);
 }
}
```



# Interfejs Map<K,V>

- Jestliže použijeme místo třídy **HashMap** třídu **TreeMap** bude výstup z programu následující:  
8 distinct words: {be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1 }
- Jestliže použijeme místo třídy **HashMap** třídu **LinkedHashMap** bude výstup z programu následující:  
8 distinct words: {if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1 }

# Interfejs Map<K,V>

- Metody **keySet**, **values** a **entrySet** (**Collection** view methods) umožňují iterovat přes klíče, hodnoty či dvojice klíč–hodnota.

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

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

# Interfejs Map<K,V>

```
// Validace slovníku atributů
static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs,
Set<K>permittedAttrs) {
 boolean valid = true;
 Set<K> attrs = attrMap.keySet();

 if (!attrs.containsAll(requiredAttrs)) {
 Set<K> missing = new HashSet<K>(requiredAttrs); // nový Set!
 missing.removeAll(attrs);
 System.out.println("Missing attributes: " + missing);
 valid = false;
 }
 if (!permittedAttrs.containsAll(attrs)) {
 Set<K> illegal = new HashSet<K>(attrs); // nový Set!
 illegal.removeAll(permittedAttrs);
 System.out.println("Illegal attributes: " + illegal);
 valid = false;
 }
 return valid;
}
```

# Interfejs Map<K,V>

- Java Collections Framework neobsahuje interfejs pro „multimaps“ (mapa s více hodnotami příslušnými k jednomu klíči). Místo toho lze použít mapu, v níž hodnota je typu **List**.
  - Tento přístup je použit v programu [Anagrams](#).
- Toto řeší rozšiřující balíčky a knihovny, např. [apache commons](#)

# Interfejs Map.Entry<K, V>

- Interfejs **Map.Entry<K, V>** je vnitřní interfejs v interfejsu **Map<K, V>**.
- Obsahuje metody pro práci s dvojicemi (key-value pair) obsaženými v kontejnerech typu **Map<K, V>**.
- Objekty typu **Map.Entry** jsou validní pouze během iterace kolekce získané pomocí metody **Map.entrySet()**.

# Řazení objektů

- Objekt typu `List<T>` lze seřadit pomocí metody

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

ze třídy **Collections**.

- Prvky seznamu musí být vzájemně porovnatelné a jejich typ musí být podtypem typu **Comparable**.
- Seznam bude seřazen vzestupně (resp. neklesající) podle přirozeného pořadí daném metodou **compareTo**(**T o**).

# Řazení objektů

- Jak implementovat interfejs **Comparable**?
- Příklad: Třída **Name** implementuje rozhraní **Comparable<Name>**.
- **Třída Name** má následující vlastnosti:
  - Objekty typu **Name** jsou immutable.
  - Konstruktor kontroluje argumenty na **null**.
  - Metoda **hashCode** je překrytá.
  - Metoda **equals** je překrytá a vrací false, je-li argument **null** či nevhodného typu.
  - Metoda **toString** je překryta a vrací „čitelný“ řetězec.

# Řazení objektů

- Metoda **compareTo** implementuje přirozené porovnání jmen, kdy příjmení má přednost před křestním jménem.
- Způsob implementace je typický a je založen na implementaci rozhraní **Comparable** ve třídě **String**.



# Řazení objektů

- Jak postupovat v případě, že chceme řadit objekty jinak než dle přirozeného uspořádání?
- V tomto případě je nutné poskytnout metodě **sort** objekt typu Comparator (interface), který je schopen porovnat dva objekty daného typu.
- Příklad.  
Předpokládejme, že máme třídu **Employee**, která implementuje interfejs **Comparable**.

# Řazení objektů

```
public class Employee implements
Comparable<Employee> {
 public Name name() { ... }
 public int number() { ... }
 public Date hireDate() { ... }
 ...
}
```

# Řazení objektů

```
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);
 }
}
```

- **možná návaznost na metody hashCode a equals!**

# Interfejs SortedSet<E>

Setříděná množina podle přirozeného řazení nebo podle zadaného Comparatoru.

- U range-view operací je nutné si uvědomit, že jde o **náhled** do původní struktury.

```
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<K, V>

Mapa s řazením klíčů – princip stejný jako u SortedSet.

```
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();
}
```

# Implementace rozhraní

- Implementace pro obecné použití:
  - Implementují všechny volitelné (optional) metody.
  - Povolují jako prvky či klíče hodnoty **null**.
  - Nejsou thread-safe.
  - Nabízí *fail-fast iteration* – mohou selhat, ale po strukturální změně kolekce vyhazují výjimku
  - Jsou serializovatelné.
- Pro většinu aplikací se dá použít některá z implementací **HashSet**, **ArrayList**, **HashMap**.

# Algoritmy

- Polymorfní algoritmy jsou implementovány v podobě statických metod ve třídě **Collections**.
- Jedná se o algoritmy pro:
  - Řazení (sorting)
  - Přeskupení (shuffling)
  - Rutinní manipulace (routine data manipulation)
  - Hledání (searching)
  - Kompozice (composition)
  - Hledání extrémních hodnot (finding extreme values)

# Řazení

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

- Metoda sort implementuje modifikovaný algoritmus merge sort, který garantuje dobu běhu  $O(n \log(n))$  a je stabilní.
  - Příklad

```
import java.util.*;
public class Sort {
 public static void main(String[] args) {
 List<String> list = Arrays.asList(args);
 Collections.sort(list);
 System.out.println(list);
 }
}
```



# Řazení

- Příklad: Výpis skupin anagramů (viz příklad [Anagrams](#)) v opačném pořadí od nejdelších skupin po nejkratší.

```
// Make a List of all anagram groups above size threshold.
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
 if (l.size() >= minGroupSize)
 winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
 public int compare(List<String> o1, List<String> o2) {
 return o2.size() - o1.size();
 }});

// Print anagram groups.
for (List<String> l : winners)
 System.out.println(l.size() + ": " + l);
```

# Přeskupování a rutinní manipulace

- Přeskupování

- `public static void shuffle(List<?> list)`
- `public static void shuffle(List<?> list, Random rnd)`
- `public static <T> void fill(List<? super T> list, T obj)`

- Rutinní operace

- `public static <T> void copy(List<? super T> dest, List<? extends T> src)`
  - pozor na mělkou a hlubokou kopii – zde mělká!
- `public static <T> boolean addAll(Collection<? super T> c, T... elements)`
- `public static void reverse(List<?> list)`
- `public static void swap(List<?> list, int i, int j)`

# Hledání

- `public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`
- `public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`
- Hledání prvku `key` v seznamu `list`. V případě, že není nalezen, provede se vložení.

```
int pos = Collections.binarySearch(list, key);
if (pos < 0)
 list.add(-pos-1, key);
```

# Složení a hledání extrémních hodnot

- Složení

- `public static int frequency(Collection<?> c, Object o)`
  - počet výskytů
- `public static boolean disjoint(Collection<?> c1, Collection<?> c2)`
  - různé kolekce

- Hledání extrémních hodnot

- `public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`
- `public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)`