Objektovo orientované programovanie

(kolekcie, algoritmy)

11. prednáška (1. časť)

Vladislav Novák
FEI STU v Bratislave
25.11.2014

Obsah

Kolekcie

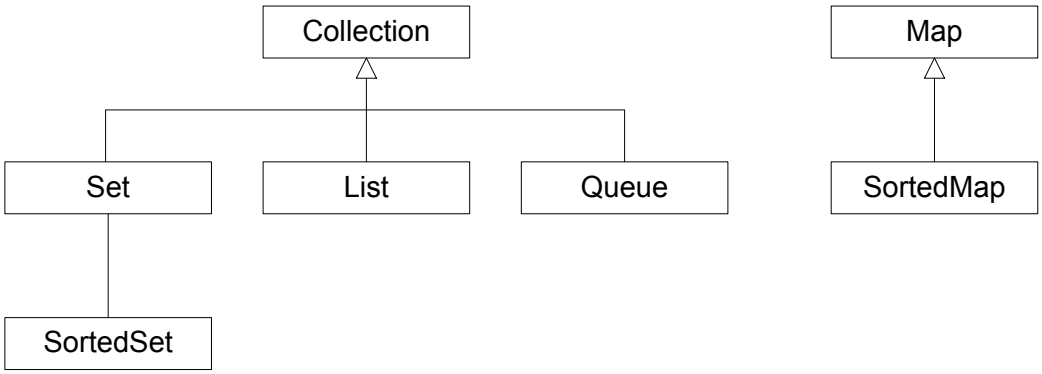
Kolekcia (*collection*) – *kontainer* – je objekt, ktorý zoskupuje viacero prvkov.

Platforma *Collection Framework* – je jednotná architektúra pre reprezentáciu a prácu kolekcii. Obsahuje tieto prvky:

- rozhrania – definujú prácu s kolekciami nezávisle od implementácie
- implementácie – konkrétne implementácie rozhraní
- algoritmy – metódy pre prácu s kolekciami (napr. hľadanie, alebo usporiadavanie prvkov). Tieto algoritmy sú polymorfné, to znamená, že rovnakú metódu možno použiť pre rôzne druhy kolekcii (v rôznych implementáciach príslušného rozhrania).

Rozhrania

Kľúčové rozhrania kolekcii tvoria hierarchiu. Hierarchia zahŕňa dva samostatné stromy.



Všetky kľúčové rozhrania sú generické. Pri deklarácii inštancie rozhrania `Collection` je vhodné uviesť typ objektov, ktoré budú v kolekcii (aj keď to nie je povinné). Určením typu umožníte aby prekladač overil, či je typ objektu umiestneného v kolekcii správny.

Voliteľné operácie

Niektoré metódy v rozhraniach sú špecifikované ako *voliteľné*. Implementácia voliteľných metód môže byť taká, že tieto metódy nerobia nič, iba vyhodia výnimku `UnsuportedOperationException`, ktorá informuje že daná operácia nie je zmysluplne implementovaná. Tieto operácie sú v dokumentácii označené ako *optional operation*.

Rozhranie Collection

Reprezentuje skupinu objektov označovaných ako *prvky*. Obsahuje všeobecné (spoločné) metódy pre prácu s kolekciami. Java neposkytuje žiadnu priamu implementáciu rozhrania `Collection`.

Rozhranie umožňuje predávanie referencie na kolekciu objektov v prípadoch, kedy je požadovaná maximálna všeobecnosť.

Podľa konvencie všetky univerzálne implementácie kolekcii obsahujú *konverzný konštruktor* s argumentom typu `Collection`, ktorý inicializuje novú kolekciu tak, aby obsahovala všetky prvky inej kolekcie. Konverzný konštruktor umožňuje prevody medzi typmi kolekcii.

Základné Metódy

`int` **`size`**`()`
vráti počet prvkov v kolekcii

`boolean` **`isEmpty`**`()`
vráti true, ak kolekcia neobsahuje prvky

`boolean` **`add`**`(E e)`
pridanie prvku do kolekcie (voliteľná metóda)

`boolean` **`remove`**`(Object o)`
odobratie špecifikovaného objektu z kolekcie (voliteľná metóda)

`boolean` **`contains`**`(Object o)`
vráti `true`, ak kolekcia obsahuje špecifikovaný objekt (ak kolekcia obsahuje aspoň jeden prvok `e` taký, že `(o==null ? e==null : o.equals(e))`)

`Iterator<E>` **`iterator`**`()`
vráti iterátor pre danú kolekciu (príklad ďalej)

`boolean` **`equals`**`(Object o)`
porovnanie kolekcie so špecifikovaným objektom

`int` **`hashCode`**`()`
vráti hash kód pre kolekciu

Hromadné operácie

`boolean` **`addAll`**`(Collection<? extends E> c)`
pridá všetky prvky zo špecifikovanej kolekcie do danej kolekcie (voliteľná metóda)

`void` **`clear`**`()`
odoberie všetky prvky z kolekcie (voliteľná metóda)

`boolean` **`containsAll`**`(Collection<?> c)`
vráti true, ak kolekcia obsahuje všetky prvky zo špecifikovanej kolekcie

`boolean` **`removeAll`**`(Collection<?> c)`
odobratie všetkých prvkov z kolekcie, ktoré sú zároveň v špecifikovanej kolekcii (voliteľná metóda)

`boolean` **`retainAll`**`(Collection<?> c)`
odobratie všetkých prvkov z kolekcie, ktoré nie sú v špecifikovanej kolekcii (voliteľná metóda)

Operácie s poliami

`Object[]` **`toArray`**`()`
vráti pole, ktoré obsahuje všetky prvky kolekcie. Výstupné pole je typu `Object[]`

`<T> T[]` **`toArray`**`(T[] a)`
vráti pole, ktoré obsahuje všetky prvky kolekcie. Typ výstupného pola je daný typom pola, ktorý je vstupným parametrom. Ak je počet prvkov kolekcie menší alebo rovný ako dĺžka pola, potom metóda uloží prvky do vstupného pola, inak vytvorí nové pole (príklad nižšie)

Prechádzanie kolekcii s použitím iterátoru

príklad:

```
public static void main(String[] args) {
    Collection<String> retazce = new ArrayList<String>();
    retazce.add("prvy");
    retazce.add("druhy");
    retazce.add("treti");
    retazce.add("stvrty");

    for(Iterator<String> it=retazce.iterator(); it.hasNext(); ) {
            String str = it.next();
            System.out.println(str);
    }
}
```

výstup:

```
prvy
druhy
treti
stvrty
```

Iterátor je objekt, ktorý identifikuje prvok kolekcie. Objekt typu `Iterator<E>` možno pre danú kolekciu získať volaním metódy `iterator`. Implementuje toto rozhranie:

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

Metódy rozhrania `Iterator<E>`

`boolean` **`hashNext`**`()`
vráti hodnotu `true`, ak sa v kolekcii nachádza nasledujúci prvok

`E` **`next`**`()`
každé volanie tejto metódy vráti nasledujúci prvok, ak existuje

`void` **`remove`**`()`
Odoberie posledný prvok vrátený metódou `next()`. Metódu `remove()` možno po volaní `next()` zavolať iba raz. Metóda `remove()` predstavuje jediný bezpečný spôsob modifikovania kolekcie počas iterácie.

príklad (použitie metódy remove):

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

Prechádzanie kolekcii pomocou konštrukcie for-each

Príkaz `for` poskytuje dva typy konštrukcie cyklov. Konštrukciu for-each možno použiť nie len na prechádzanie polí, ale aj na prechádzanie ľubovoľného typu ktorý implementuje rozhranie `Iterable`.
Rozhranie `Collection` rozširuje rozhranie `Iterable`. To znamená, že každá implementácia rozhrania `Collection` musí poskytovať iterátor typu `Iterator` (prácu s iterátorom uvádzajú predchádzajúce príklady).
Vďaka tomu možno konštrukciu for-each použiť aj na kolekcie.

príklad:

```
public static void main(String[] args) {
    Collection<String> retazce = new ArrayList<String>();
    retazce.add("prvy");
    retazce.add("druhy");
    retazce.add("treti");
    retazce.add("stvrty");

    //príkaz for v tvare konštrukcie for-each
    for(String s : retazce) {
        System.out.println(s);
    }
}
```

výstup:

```
prvy
druhy
treti
stvrty
```

Operácie vracajúce pole

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

príklad:

```
public static void main(String[] args) {
    Collection<String> kolekcia = new ArrayList<String>();
    kolekcia.add("prvy prvok");
    kolekcia.add("druhy prvok");

    Object[] poleA = kolekcia.toArray();
    System.out.println("poleA: " + poleA.length);
    System.out.println("poleA: " + Arrays.toString(poleA));
    System.out.println();

    String[] pomocne;

    pomocne = new String[0];
    String[] poleB = kolekcia.toArray(pomocne);
    System.out.println("poleB: " + (poleB == pomocne)); //false
    System.out.println("poleB: " + poleB.length);
    System.out.println("poleB: " + Arrays.toString(poleB));
    System.out.println();

    pomocne = new String[kolekcia.size()];
    String[] poleC = kolekcia.toArray(pomocne);
    System.out.println("poleC: " + (poleC == pomocne)); //true
    System.out.println("poleC: " + poleC.length);
    System.out.println("poleC: " + Arrays.toString(poleC));
}
```

výstup:

```
poleA: 2
poleA: [prvy prvok, druhy prvok]

poleB: false
poleB: 2
poleB: [prvy prvok, druhy prvok]

poleC: true
poleC: 2
poleC: [prvy prvok, druhy prvok]
```

Rozhranie Set

Kolekcia typu `Set` nemôže obsahovať duplicitné prvky. Modeluje abstrakciu matematickej množiny.
Rozhranie `Set` obsahuje iba metódy zdedené z rozhrania `Collection` a doplňuje obmedzenia, ktoré zakazujú duplicitné prvky. Tiež pridáva silnejší predpis pre chovanie `equals()` a `hasCode()`, takže možno inštancie typu `Set` zmysluplne porovnávať aj vtedy, keď sa líšia ich implementované typy. Dve inštancie typu `Set` sú zhodné vtedy, ak obsahujú rovnaké prvky.

Napríklad metóda `add(E e)` pridá prvok do množiny iba vtedy ak sa daný prvok v množine ešte nenachádza.

Príklady tried implementujúcich rozhranie `Set`:
- `HashSet` - vo väčšine prípadov najvýhodnejšia implementácia
- `TreeSet`
- `LinkedHashSet`

príklad (zisťovanie duplicitných slov)
```
public static void main(String[] args) {
    String veta = "jeden dva tri styri dva styri dva pat";
    Set<String> mnozina = new HashSet<String>();
    for(String slovo : veta.split(" ")) {
        if( ! mnozina.add(slovo) ) {
            System.out.println("duplicita: " + slovo);
        }
    }
    System.out.println("vsetky slova vo vete: " + mnozina);
}
```
výstup:
```
duplicita: dva
duplicita: styri
duplicita: dva
vsetky slova vo vete: [styri, pat, dva, jeden, tri]
```
príklad (zisťovanie duplicitných slov – pomocou hromadných operácii):
```
public static void main(String[] args) {
    String veta = "jeden dva tri styri dva styri dva pat";
    Set<String> jedinecne  = new HashSet<String>();
    Set<String> duplicitne = new HashSet<String>();
    for(String slovo : veta.split(" ")) {
        if( ! jedinecne.add(slovo) ) {
            duplicitne.add(slovo);
        }
    }
    jedinecne.removeAll(duplicitne);

    System.out.println("jedinecne  slova: " + jedinecne);
    System.out.println("duplicitne slova: " + duplicitne);
}
```
výstup:
```
jedinecne  slova: [pat, jeden, tri]
duplicitne slova: [styri, dva]
```

V príklade sú typy premenných `jedinecne` a `duplicitne` typu `Set`, nie sú typu `HashSet`. Čiže typ premennej je rozhranie, nie zvolená implementácia. Tento systém voľby typu premennej je často výhodný, pretože umožňuje pružne meniť implemntáciu iba zmenou konštruktora. Na druhej strane, ale zamedzuje použitie špecializovaných (neštandardných) operácii, ktoré nie sú v rozhraní.
Ak by sme napríklad chceli vypísať slová v abecednom poradí, stačí zmeniť implementáciu typu `Set` z `HashSet` na `TreeSet`. Výstup s použitím `TreeSet`:

```
jedinecne  slova: [jeden, pat, tri]
duplicitne slova: [dva, styri]
```

Rozhranie List

Kolekcia typu `List` reprezentuje usporiadaný zoznam (sekvenciu) prvkov. Okrem operácii zdedených od rozhrania `Collection` zahrňuje operácie nasledovného typu:

- pozičný prístup – manipulácia s prvkami na základe ich číselného indexu v zozname
- hľadanie – vyhľadá v zozname určitý objekt a vráti jeho číselnú pozíciu
- iterácia – rozširuje sémantiku objektov typu `Iterator`, aby bolo možné využiť sekvenčnú povahu zoznamu
- zobrazenie rozsahu – operácie s podzoznamami v určitom rozsahu

Príklady tried implementujúcich rozhranie `List`:

- `ArrayList` -vo väčšine prípadov najvýhodnejšia implementácia
- `LinkedList`
- `Vector`

Základné metódy

`boolean` **`add`**`(E e)`
pridá prvok na koniec zoznamu (voliteľná operácia)

`boolean` **`equals`**`(Object o)`
porovná dva zoznamy. Inštancie rozhrania `List` sú rovnaké, ak obsahujú rovnaké prvky v rovnakom poradí.

`int` **`hashCode`**`()`
vráti hash kód zoznamu (závisí aj od poradia prvkov (na rozdiel od rozhrania `Set`))

`boolean` **`isEmpty`**`()`
vráti true, ak je zoznam prázdny

`int` **`size`**`()`
vráti počet prvkov v zozname

Operácie pozičného prístupu

`E` **`get`**`(int index)`
vráti prvok na pozícii špecifikovanej indexom

`E` **`remove`**`(int index)`
odstráni prvok zoznamu špecifikovaný indexom (voliteľná operácia)

`E` **`set`**`(int index, E element)`
nahradí prvok špecifikovaný indexom prvkom ktorý je argument (voliteľná operácia)

`void` **`add`**`(int index, E element)`
pridá prvok na pozíciu danú indexom (voliteľná operácia)

Operácia hľadania

`int` **`indexOf`**`(Object o)`
vráti index prvého výskytu špecifikovaného prvku, alebo -1 ak sa prvok nenachádza v zozname

`int` **`lastIndexOf`**`(Object o)`
vráti index posledného výskytu špecifikovaného prvku, alebo -1 ak zoznam neobsahuje daný prvok

`boolean` **`contains`**`(Object o)`
vráti true ak zoznam obsahuje špecifikovaný objekt

`boolean` **`remove`**`(Object o)`
odstráni prvý výskyt špecifikovaného prvku (voliteľná operácia) (pri porovnávaní prvkov využíva metódu equals)

Metódy vracajúce iterátory

`Iterator<E>` **`iterator`**`()`
vráti iterátor typu `Iterator<E>`

`ListIterator<E>` **`listIterator`**`()`
vráti iterátor typu `ListIterator<E>` (príklad ďalej)

`ListIterator<E>` **`listIterator`**`(int index)`
vráti iterátor typu `ListIterator<E>`, ktorého pozícia je daná indexom

Hromadné operácie

`boolean` **`addAll`**`(Collection<? extends E> c)`
pridá všetky prvky v špecifikovanej kolekcii na koniec zoznamu

`boolean` **`addAll`**`(int index, Collection<? extends E> c)`
do zoznamu pridá všetky elementy v špecifikovanej kolekcii od špecifikovanej pozície

`void` **`clear`**`()`
odstráni všetky prvky zo zoznamu (voliteľná operácia)

`boolean containsAll(Collection<?> c)`
vráti true ak zoznam obsahuje všetky prvky z kolekcie ktorá je argumentom

`boolean removeAll(Collection<?> c)`
odstráni všetky prvky zoznamu, ktoré sú obsiahnute v kolekcii špecifikovanej parametrom (voliteľná operácia)

`boolean retainAll(Collection<?> c)`
v zozname ponechá iba tie prvky, ktoré sú tiež v špecifikovanej kolekcii (voliteľná operácia)

Operácie zobrazenia rozsahu

`List<E> subList(int fromIndex, int toIndex)`
vráti zobrazenie časti zoznamu medzi indexami fromIndex (vrátane) a toIndex (nie vrátane) (príklad ďalej)

Operácia s poliami

`Object[] toArray()`
vráti pole obsahujuce všetky prvky zoznamu

`<T> T[] toArray(T[] a)`
vrtáti pole obsahujúce všetky prvky zoznamu. Typ vráteného pola je daný typom argumentu

Iterátor typu `ListIterotor<E>`

Iterátor typu `Iterator<E>` umožňuje prejsť všetky prvky zoznamu (jedným smerom). Iterátor typu `ListIterator<E>`, umožňuje prechádzať zoznam oboma smermi, meniť zoznam, zistiť aktuálnu polohu iterátora v zozname.

Rozhranie `ListIterator<E>`:

```
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
}
```

Prechádzanie prvkov zoznamu využitím pozičného prístupu

```
public static void main(String[] args) {
    List<String> zoznam = new ArrayList<String>();
    zoznam.add("nula");
    zoznam.add("jeden");
    zoznam.add("dva");
    zoznam.add("tri");
    zoznam.add("styri");

    for (int i = 0; i < zoznam.size(); i++) {
        System.out.println(zoznam.get(i));
    }
}
```

výstup:

```
nula
jeden
dva
tri
styri
```

Operácie zobrazenia rozsahu

Metóda `List<E> subList(int fromIndex, int toIndex)` vráti časť zoznamu ako inštanciu typu List. Vrátená inštancia predstavuje zobrazenie, t.j. zmeny vo vrátenej časti zoznamu sa prejavia v celom zozname a naopak.

príklad:

```
public static void main(String[] args) {
    List<String> zoznam = new ArrayList<String>();
    zoznam.add("nula");
    zoznam.add("jeden");
    zoznam.add("dva");
    zoznam.add("tri");
    zoznam.add("styri");
    zoznam.add("pat");
    zoznam.add("sest");
    zoznam.add("sedem");

    System.out.println(zoznam);

    List<String> podzoznam = zoznam.subList(2, 5);
    System.out.println(podzoznam);

    zoznam.set(2, "DVA"); //prejaví sa aj v podzozname
    System.out.println(podzoznam);

    podzoznam.clear(); //prejaví sa aj v hlavnom zozname
    System.out.println(zoznam);
}
```

výstup:

```
 [nula, jeden, dva, tri, styri, pat, sest, sedem]
[dva, tri, styri]
[DVA, tri, styri]
[nula, jeden, pat, sest, sedem]
```

Ak potrebujeme pole ako typ `List`

Trieda `Arrays` obsahuje metódu `asList()`, ktorá dovoľuje zobraziť pole ako kolekciu typu `List`. Táto metóda nekopíruje pole. Veľkosť zobrazenia (zoznamu) sa nedá meniť (podobne ako sa nedá meniť veľkosť pola).

príklad:

```
public static void main(String[] args) {
    String[] pole = {"aaaa", "bbbb", "cccc", "dddd"};
    List<String> zoznam = Arrays.asList(pole);

    zoznam.set(1,"BBBB"); //zmena sa prejavý aj v poli
    //zoznam.add("eeee"); //v tomto zobrazení nie je podporovaná

    System.out.println(Arrays.toString(pole));
    System.out.println(zoznam);
}
```

výstup:

```
[aaaa, BBBB, cccc, dddd]
[aaaa, BBBB, cccc, dddd]
```

Príklad na využitie konverzného konštruktora

Najprv vytvoríme zoznam s opakujúcimi sa slovami. Potom vytvoríme množinu slov, v ktorej sa každé slovo zo zoznamu bude nachádzať iba raz.

```
public static void main(String[] args) {
    List<String> zoznam = new ArrayList<String>() {};
    zoznam.add("aaaa");
    zoznam.add("aaaa");
    zoznam.add("bbbb");
    zoznam.add("cccc");
    zoznam.add("dddd");
    zoznam.add("eeee");
    zoznam.add("bbbb");
    zoznam.add("bbbb");

    System.out.println(zoznam);

    Set<String> mnozina = new HashSet<String>(zoznam);
    System.out.println(mnozina);

    Set<String> usporiadanaMnozina = new TreeSet<String>(zoznam);
    System.out.println(usporiadanaMnozina);
}
```

výstup:

```
[aaaa, aaaa, bbbb, cccc, dddd, eeee, bbbb, bbbb]
[bbbb, dddd, aaaa, eeee, cccc]
[aaaa, bbbb, cccc, dddd, eeee]
```

Rozhranie Queue

Kolekcia typu `Queue` uchováva prvky určené ku spracovaniu. Ku operáciám zdedeným z rozhrania `Collection` pridáva operácie fronty:

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

`Queue` zpravidla (ale nie vždy) radia prvky spôsobom FIFO. K výnimkám patria prioritné fronty, ktoré radia prvky podľa ich hodnôt.

Všetky metódy rozhrania `Queue` existujú v dvoch verziách
- jedna pri neúspešnej operácii spôsobí výnimku
- druhá pri neúspešnej operácii vráti špeciálnu hodnotu (`null`, alebo `false`)

	spôsobí výnimku	vráti špec. hodnotu
vloženie (typom fronty je určené kam):	**`add`**`(e)`	**`offer`**`(e)`
odobratie prvého prvku:	**`remove`**`()`	**`poll`**`()`
vrátenie prvého prvku (ale neodoberie prvok):	**`element`**`()`	**`peek`**`()`

Imlementácie rozhrania `Queue` obvykle neumožňujú vkladať hodnotu `null`. Z historických dôvodov povoľuje vložiť hodnotu `null` trieda `LinkedList`. Metódy `poll()` a `peek()`, ale vracajú hodnotu `null` pri neúspechu.

Príklady tried implementujúcich rozhranie `Queue`:
- `LinkedList`
- `PriorityQueue`

Rozhranie Map

Kolekcia typu `Map` mapuje kľúče na hodnoty. Mapa nemôže obsahovať duplicitné kľúče a každý kľúč môže identifikovať nanajvýš jednu hodnotu. Rozhranie `Map` modeluje abstrakciu matematickej funkcie.

Príklady tried implementujúcich rozhranie `Map`:

- `HashMap` - vo väčšine prípadov najvýhodnejšia implementácia
- `TreeMap`
- `LinkedHashMap`

Základné operácie

`V` **`get`**`(Object key)`
vráti hodnotu prislúchajúcu špecifikovanému kľúču, alebo hodnotu `null`, ak mapa neobsahuje kľúč

`V` **`put`**`(K key, V value)`
namapuje špecifikovaný kľúč na špecifikovanú hodnotu (voliteľná operácia)

`V` **`remove`**`(Object key)`
odstráni mapovanie špecifikovaného kľúča (voliteľná operácia)

`boolean` **`containsKey`**`(Object key)`
vráti true, ak mapa obsahuje mapovanie špecifikovaného kľúča

`boolean` **`containsValue`**`(Object value)`
vrtáti true, ak mapa obsahuje špecifikovanú hodnotu

`boolean` **`isEmpty`**`()`
vráti true, ak je mapa prázdna

`int` **`size`**`()`
vráti počet mapovaní kľúč → hodnota

`boolean` **`equals`**`(Object o)`
vráti true ak sú mapy rovnaké. Dve mapy sú rovnaké, ak reprezentujú rovnaké mapovanie kľúčov na hodnoty.

`int` **`hashCode`**`()`
vráti hašovací kód mapy

Hromadné operácie

`Set<K>` **`keySet`**`()`
vráti zobrazenie množiny kľúčov v mape

`void` **`clear`**`()`
odstráni všetky prvky z mapy (voliteľná operácia)

`void` **`putAll`**`(Map<? extends K,? extends V> m)`
kopíruje mapovanie zo špecifikovanej mapy do mapy nad ktorou sa volá táto operácia (voliteľná operácia)

Operácie zobrazenia

`Set<Map.Entry<K,V>>` **`entrySet`**`()`
vráti zobrazenie mapy ako množinu dvojic (kľuč, hodnota). Využiva sa vnorené rozhranie `Map.Entry`

`Collection<V>` **`values`**`()`
vráti zobrazenie hodnôt ako kolekciu typu `Collection`

Všetky univerzálne implementácie rozhrania `Map` podľa konvencie poskytujú konštruktory, ktoré majú vstupný parameter typu `Map` a inicializujú nový objekt, ktorý obsahuje rovnaké mapovanie kľúčov a hodnôt.
Tento mechanizmus je analogický ku štandardnemu konštruktoru rozhrania `Collection`

Vnorené rozhranie:

```
// Interface for entrySet elements
public interface Entry {
    boolean equals(Object o);
    K getKey(); //vráti kľúč
    V getValue(); // vráti hodnotu
    int hashCode();
    V setValue(V value); //nastaví hodnotu
}
```

príklad (frekvenčná analýza slov):

```
public static void main(String[] args) {
    String veta = "aaa bbb ccc ddd aaa ccc ddd ddd eee";
    Map<String, Integer> mapa = new HashMap<String,Integer>();

    for(String s : veta.split(" ")) {
        Integer pocet = mapa.get(s);
        mapa.put(s, (pocet==null) ? 1 : pocet+1);
    }

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

výstup:

```
{aaa=2, ddd=3, ccc=2, bbb=1, eee=1}
```

príkad: Testovanie, či je mapa `m2` podmapou mapy `m1`

```
m1.entrySet().containsAll(m2.entrySet())
```

Viacnásobné mapy

Viacnásobná mapa (*multimap*) sa podobá na objekty typu `Map`, ale môže mapovať každý kľúč na viacero hodnôt. Java Collectin Framework neobsahuje žiadne rozhranie viacnásobnej mapy, pretože sa nepoužíva príliš často. Je pomerne jednoduché implementovať viacnásobnú mapu ako objekt typu `Map`, ktorého hodnotami sú inštancie rozhrania `List`.

príklad:

```
import java.util.*;

public class ViacnasobnaMapa {
    public static void main(String[] args) {
        Map<String, List<String>> multiMap
                        = new HashMap<String, List<String>>();

        List l1 = new ArrayList<String>();
        l1.add("prve");
        l1.add("druhe");
        multiMap.put("kluc1", l1);

        List l2 = new ArrayList<String>();
        l2.add("tretie");
        l2.add("stvrte");
        l2.add("piate");
        multiMap.put("kluc2", l2);

        for( String kluc : multiMap.keySet() ) {
            List<String> zoznamHodnot = multiMap.get(kluc);
            System.out.println(kluc + ": " + zoznamHodnot);
        }
    }
}
```

výstup:

```
kluc2: [tretie, stvrte, piate]
kluc1: [prve, druhe]
```

Rozhranie SortedSet

Rozhranie `SortedSet` je variantou rozhrania `Set`. Uchováva prvky v zostupnom poradí. Poradie je dané prirodzeným usporiadaním, alebo komparátorom ktorý je uvedený pri vytvorení kolekcie typu `SortedSet`.
Komparátormi sa budeme zaoberať v ďalšej časti.

Príklad tried implementujúcej rozhranie `SortedSet`:

- `TreeSet<E>`

```
public interface SortedSet<E> extends Set<E> {

    //vráti zobrazenie rozsahu kde fromElement<=prvok<toElement
    SortedSet<E> subSet(E fromElement, E toElement);

    //vráti zobrazenie rozsahu kde sú prvky menšie ako toElement
    SortedSet<E> headSet(E toElement);

   //vráti zobr. r. kde sú prvky väčšie, alebo rovné ako fromElement
    SortedSet<E> tailSet(E fromElement);

    //vráti najmenší prvok
    E first();

    //vráti najväčší prvok
    E last();

    //vráti komparátor použitý pre triedenie
    Comparator<? super E> comparator();
}
```

Rozhranie SortedMap

Rozhranie `SortedMap` je verzia rozhranie `Map`, ktorá uchováva prvky v zostupnom poradí. Poradie je dané prirodzeným poradím kľúčov, alebo objektom typu `Comparator`, ktorý je uvedený pri vytváraní kolekcie typu `SortMap` (komparátormi sa budeme zaoberať neskôr).

Príklad tried implementujúcej rozhranie `SortedMap`:

- `TreeMap`

```
public interface SortedMap<K, V> extends Map<K, V>{
    //vráti komparátor použitý pre triedenie
    Comparator<? super K> comparator();

    //vráti zobr. rozsahu kde pre kluč platí fromKey<=kluc<toKey
    SortedMap<K, V> subMap(K fromKey, K toKey);

    //vráti zobrazenie rozsahu kde kľúče prvkov sú menšie ako toKey
    SortedMap<K, V> headMap(K toKey);

    //vráti zobr. rozs. kde kľúče su väčšie, alebo rovné ako fromKey
    SortedMap<K, V> tailMap(K fromKey);

    //vráti zobrazenie ako množinu dvojíc
    Set<Map.Entry<K,V>> entrySet();

    //vráti zobrazenie kľúčov ako množinu
    Set<K> keySet();

    //vráti zobrazenie hodnôt ako kolekciu
    Collection<V> values();

    //vráti prvý (najmänší) kľúč
    K firstKey();

    //vráti posledný (najväčší kľúč)
    K lastKey();
}
```

Algoritmy – trieda Collections

Trieda `Collections` slúži pre prácu s kolekciami podobne, ako trieda `Arrays` pre prácu s poliami. Obsahuje *polymorfné algoritmy*, ktoré sa pri práci s kolekciami často používajú. Majú formu statických metód. Väčšina algoritmov pracuje s kolekciami typu `List`.

Usporiadanie prvkov zoznamu

Operácia **`sort`**`()` je mierne optimalizovaný algoritmus merge sort, ktorý je rýchly a stabilný.
Rýchly – znamená že sa vykoná v čase n (log n), alebo rýchlejšie.
Stablilný – znamená, že nemení poradie prvkov s rovnakou hodnotou.
Metóda je určená pre typ `List`.

Rozhranie `Comparable<T>`

Zoznam objektov typu `List` možno usporiadať pomocou metódy `sort()` v tvare:
`Collections.sort(List<T> list).`
príklad:

```
import java.util.*;

public class UsporiadanieZoznamu {
    public static void main(String[] args) {
        List<String> l = new ArrayList<String>();
        l.add("prvy");
        l.add("druhy");
        l.add("treti");
        l.add("stvrty");
        System.out.println(l); //[prvy, druhy, treti, stvrty]
        Collections.sort(l);
        System.out.println(l); //[druhy, prvy, stvrty, treti]
    }
}
```

Tento spôsob usporiadanie prvkov funguje ak prvky v zozname implementujú rozhranie `Comparable`. Ak sa pokúsime takýmto spôsobom usporiadať prvky, ktoré neimplementujú rozhranie `Comparable`, potom `Collection.sort(List<T> list)` spôsobý výnimku `ClassCastException`.

Podobne pre usporiadanie pola je možné využiť niektorú preťaženú metódu `sort()` v triede `Arrays`.

Príklady tried, ktoré implementujú rozhranie `Comparable`:

- `Byte`
- `Character`
- `Long`
- `Integer`
- `Short`
- `Double`
- `Float`
- `BigInteger`
- `BigDecimal`
- `Boolean` (`Boolean.FALSE` < `Boolean.TRUE`)

`File` (lexikografické usporiadanie podľa cesty - systémovo závisle)
`String` (lexikografické usporiadanie)
`Date` (chronologicky)
`CollationKey` (lexikografické usporiadanie závisle na miestnom nastavení)

Prvky rôznych typov môžu byť porovnateľné, ale žiadna z uvedených tried neumožňuje porovnanie inštancii rôznych tried.

Rozhranie comparable:

```
public interface Comparable<T> {
    int   compareTo(T o);
}
```

Metóda `compareTo(T o)` vráti

- záporné číslo ak objekt this je menší ako objekt ktorý je parametrom
- nulu ak sú objekty rovnaké
- kladné číslo – ak je objekt this väčší ako objekt ktorý je parametrom

Implementácia tohoto rozhrania udáva tzv. *prirodzené usporiadanie*.

Príklad implementácie rozhrania `Comparable`:

Definícia triedy implementujúcej `Comparable`:

```
public class Student implements Comparable<Student>{
    private String meno;
    private String priezvisko;
    private int rocnik;

    public Student(String meno, String priezvisko, int rocnik) {
        this.meno = meno;
        this.priezvisko = priezvisko;
        this.rocnik = rocnik;
    }

    public String dajMeno() {
        return meno;
    }

    public String dajPriezvisko() {
        return priezvisko;
    }

    public int dajRocnik() {
        return rocnik;
    }

    @Override
    public String toString() {
        return "meno: " + priezvisko + " " + meno + ", rocnik=" + rocnik;
    }

    //Metóda compareTo porovnáva študentov najprv podľa priezviska
    //Ak sú priezviská rovnaké, tak podľa mena.
    // Ak sú priezviská aj mená rovnaké, tak porovnáva podľa ročníka.
    @Override
    public int compareTo(Student dalsi) {
        int vysledok = priezvisko.compareTo(dalsi.priezvisko);
        vysledok = (vysledok != 0) ? vysledok : meno.compareTo(dalsi.meno);
        vysledok = (vysledok != 0) ? vysledok : rocnik - dalsi.rocnik;
        return vysledok;
    }
```

```
}
```

Metóda `main`:

```
public static void main(String[] args) {
    List<Student> studenti = new ArrayList<>();

    studenti.add(new Student("Samo","Chalupka",  1));
    studenti.add(new Student("Adam","Troskovic", 1));
    studenti.add(new Student("Jan","Benovsky",   1));
    studenti.add(new Student("Roman","Chalupka", 2));
    studenti.add(new Student("Moric","Benovsky", 2));


    for(Student student : studenti) {
        System.out.println(student);
    }
    System.out.println();

    Collections.sort(studenti);

    for(Student student : studenti) {
        System.out.println(student);
    }
}
```

výstup:

```
meno: Chalupka Samo, rocnik=1
meno: Troskovic Adam, rocnik=1
meno: Benovsky Jan, rocnik=1
meno: Chalupka Roman, rocnik=2
meno: Benovsky Moric, rocnik=2

meno: Benovsky Jan, rocnik=1
meno: Benovsky Moric, rocnik=2
meno: Chalupka Roman, rocnik=2
meno: Chalupka Samo, rocnik=1
meno: Troskovic Adam, rocnik=1
```

Rozhranie `Comparator`

Ak chceme usporiadať postupnosť prvkov, ktoré neimplementujú rozhranie `Comparable`, alebo toto rozhranie implementujú, ale chceme postupnosť usporiadať podľa iných pravidiel, potom môžeme vytvoriť a použiť inštanciu implementujúcu rozhranie `Comparator`:

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

Metóda `compare(T o1, T o2)` vráti

- záporné číslo ak objekt `o1` je menší ako objekt `o2`
- nulu ak sú objekty rovnaké
- kladné číslo – ak je objekt `o1` väčší ako objekt `o2`

Pre takéto usporiadanie je určená iná verzia preťaženej metódy sort:
`Collections.sort(List<T> list, Comparator<? super T> c)`

Táto metóda tiež spôsobí výnimku `ClassCastException` ak prvky nie sú navzájom porovnateľné pomocou komparátora.

Podobne pre usporiadanie pola je možné využiť niektorú preťaženú metódu `sort()` v triede `Arrays`.

V nasledujúcom príklade využijeme definíciu triedy `Student` uvedenú v predchádzajúcom príklade. Vytvoríme objekt slúžiaci na usporiadanie zoznamu študentov podľa iných kritérií. Najprv definujeme triedu implementujúcu rozhranie `Comparator<T>`. Jej metóda `compare` bude porovnávať študentov najprv podľa ročníka. Ak budú ročníky rovnaké, tak bude porovnávať podľa priezviska. Ak budú aj priezviská rovnaké, tak podľa mena.

```
public class StudentComparator implements Comparator<Student> {

    @Override
    public int compare(Student s1, Student s2) {
        int vysledok = s1.dajRocnik() - s2.dajRocnik();
        vysledok = (vysledok != 0) ? vysledok :
                           s1.dajPriezvisko().compareTo(s2.dajPriezvisko());
        vysledok = (vysledok != 0) ? vysledok :
                                     s1.dajMeno().compareTo(s2.dajMeno());
        return vysledok;
    }
}
```

main:

```
public static void main(String[] args) {
    List<Student> studenti = new ArrayList<>();

    studenti.add(new Student("Samo","Chalupka",  1));
    studenti.add(new Student("Adam","Troskovic", 1));
    studenti.add(new Student("Jan","Benovsky",   1));
    studenti.add(new Student("Roman","Chalupka", 2));
    studenti.add(new Student("Moric","Benovsky", 2));

    for(Student student : studenti) {
        System.out.println(student);
    }
    System.out.println();

    Collections.sort(studenti, new StudentComparator());

    for(Student student : studenti) {
        System.out.println(student);
    }
}
```

výstup:

```
meno: Chalupka Samo, rocnik=1
meno: Troskovic Adam, rocnik=1
meno: Benovsky Jan, rocnik=1
meno: Chalupka Roman, rocnik=2
meno: Benovsky Moric, rocnik=2

meno: Benovsky Jan, rocnik=1
meno: Chalupka Samo, rocnik=1
meno: Troskovic Adam, rocnik=1
meno: Benovsky Moric, rocnik=2
meno: Chalupka Roman, rocnik=2
```

Metóda `equals` v rozhraní `Comparator` slúži na porovnávanie komparátorov. Môže vrátiť `true` iba ak je vstupným parametrom komparátor definujúci rovnaké usporiadanie prvkov. Metódu `equals` nie je potrebné prekryť. Avšak prekrytie tejto metódy môže v niektorých prípadoch zvýšiť efektivitu vykonávania programu.

Premiešanie prvkov zoznamu

Premiešanie je určené pre inštancie typu `List`. Vykonáva sa preťaženou metódou **`shuffle`**`()`. Táto metóda premieša prvky v zozname. Vstupom do metódy môže byť aj generátor náhodných čísiel, ktorý sa použije pri premiešavaní prvkov.

Rutinná práca s údajmi

Trieda `Collections` poskytuje viacero algoritmov pre rutinnú prácu s údajmi.

Metóda **`reverse`**`()` obráti poradie prvkov v kolekcii typu `List`.

Metóda **`copy`**`()` má dva vstupné argumenty typu `List`. Kopíruje prvky zo zdrojového zoznamu do cieľového zoznamu, pričom prepisuje obsah cieľového zoznamu. Cieľový zoznam musí byť minimálne tak dlhý, ako zdrojový zoznam. Ak je cieľový zoznam dlhší, potom jeho zvyšné prvky zostanú bez zmeny.

Metóda **`swap`**`()` vymení dva prvky v kolekcii typu `List`. Ich miesto je určené indexmi.

Metóda **`addAll`**`()` pridá všetky prvky do kelekcie typu `Collection`. Pridávané prvky možno určiť jednotlivo, alebo ako pole.

Metóda **`replaceAll`**`()` nahradí všetky výskyty jednej hodnoty inou.

Metóda **`rotate`**`()` rotuje všetky prvky v kolekcii typu `List` o určenú vzdialenosť.

Metóda **`fill`**`()` prepíše všetky prvky kolekcie typu `List` špecifikovanou hodnotou.

Metóda **`indexOfSubList`**`()` vráti index prvého výskytu podzoznamu v zozname

Metóda **`lastindexOfSubList`**`()` vráti index posledného výskytu podzoznamu v zozname

Hľadanie prvku v zozname

Na hľadanie v zozname typu `List` sa používa preťažená metóda **`binarySearch`**`()`. Pred volaním tejto metódy musí byť zoznam usporiadaný. Táto metóda podobne ako metóda `sort()` má dve formy. Prvá má dva argumenty, ktorými sú zoznam a hľadaný prvok, druhá forma má o jeden argument viac. Týmto argumentom je komparátor.

Zoznam musí byť pred hľadaním prvku usporiadaný podľa tých pravidiel, ktoré sa použijú aj pri hľadaní (napr. treba použiť ten istý komparátor).

Návratová hodnota je index nájdeného prvku. Ak metóda hľadaný prvok nenájde, potom je návratovou hodnotou hodnota `(-bod_vloženia)-1)`, kde `bod_vloženia` označuje index prvého prvku, ktorý je väčší ako hľadaná hodnota (alebo hodnota `zoznam.size()`). Tento trik kombinuje logickú informáciu najdený/nenájdený a celočíselný údaj (index). To znamená, že pri nenájdení hľadaného prvku je návratová hodnota záporná.

Testy zloženia

Metóda **`frequency`**`()` spočíta počet výskytov špecifikovaného prvku v špecifikovanej kolekcii typu `Collection`.

Metóda **`disjoint`**`()` vráti `true`, ak dve kolekcie nemajú spoločný prvok

Hľadanie extrémnych hodnôt

Metóda **`min`**`()` vráti najmenší prvok v kolekcii typu `Collection`.

Metóda **`max`**`()` vráti najväčší prvok v kolekcii typu `Collection`.

Zreťazené hromadné operácie (sequence of aggregate operations)

V tejto časti budú popísané hromadné operácie, ktoré sú v anglickej literatúre označované ako *aggretage operations*. Vstupnými parametrami týchto operácii sú „funkcie“, ktoré sa majú vykonať nad prvkami. Pri ich použití definujeme čo sa má (nad každým prvkom) vykonať, ale nedefinujeme spôsob iterácie cez tieto prvky.

Zreťazenie týchto operácii sa v angličtine nazýva *pipeline*.

Zreťazené hromadné operácie umožňujú tvorbu prehľadného kódu spracovávajúceho údaje napríklad v kolekciách. Nasledujúci príklad bude nadväzovať na kód v predchádzajúcich častiach. Vypíše všetkých študentov v 2. ročníku. Najprv pomocou cyklu for

```
for(Student student : studenti) {
    if( student.dajRocnik() == 2 ) {
        System.out.println(student);
    }
}
```

Výpis pomocou zreťazených hromadných operácii

```
studenti
    .stream()
    .filter(student -> student.dajRocnik() == 2)
    .forEach(student -> System.out.println(student));
```

Metóda `stream` vracia prúd údajov. Prúd údajov je sekvencia prvkov. V tomto prípade sú to prvky z kolekcie. Zdrojom údajov nemusí byť len kolekcia.
Metóda `filter` vráti nový prúd údajov, obsahujúci len prvky ktoré spĺňajú predikát (ktorý je parametrom).
Metóda `forEach` nevracia nový prúd údajov. Jej parametrom je operácia, ktorá sa má vykonať nad každým prvkom.

Príklad nájdenia najnižšieho ročníka v ktorom študujú študenti zo zoznamu

```
int najnizsiRocnik = studenti
            .stream()
            .mapToInt(Student::dajRocnik)
            .min()
            .getAsInt();
```

Metóda `mapToInt` vracia prúd údajov typu `IntStream`. Metóda zavolá pre každého študenta metódu vracajúcu ročník v ktorom študuje. Tieto ročníky budú prvkami nového prúdu údajov. Metóda `min` vráti minimálnu hodnotu z prúdu údajov. Jej návratová hodnota je typu `OptionalInt`. Ak prúd údajov spracovávaný metódou `min` obsahuje údaje, tak metóda `getAsInt` vráti minimálnu hodnotu typu `int`, inak metóda `min` vráti prázdny objekt typu `OptionalInt` a metóda `getAsInt` vyhodí výnimku `NoSuchElementException`.

Java API obsahuje aj ďalšie metódy, ktoré vrátia jednu hodnotu vypočítanú na základe údajov z prúdu (*reduction operations*). Napríklad metódy vracajúce priemer, alebo počet prvkov v prúde. Okrem toho obsahuje aj všeobecnú metódu `reduce`, ktorá umožňuje definovať vlastnú metódu výpočtu výslednej hodnoty podľa údajov v prúde.

Nasledujúci príklad nájde najnižší ročník, ale namiesto metódy `min`, použijeme pre ukážku metódu `reduce`, kde definujeme spôsob nájdenia výslednej hodnoty

```
int najnizsiRocnik = studenti
    .stream()
    .mapToInt(Student::dajRocnik)
    .reduce((a, b) -> Math.min(a, b))
    .getAsInt();
```

Ďalší príklad ukazuje definíciu redukčnej operácie, ktorá vráti `sumu` hodnôt (pravdaže výhodnejšie by v tomto prípade bolo použitie metódy `sum`)

```
int suma = studenti
    .stream()
    .mapToInt(Student::dajRocnik)
    .reduce(0, (a, b) -> a + b);
```

Prvý parameter metódy `reduce` je inicializačná hodnota (v prípade, že prúd neobsahuje žiadne údaje, tak aj návratová hodnota). Druhý parameter je funkcia, ktorej argumenty sú čiastočná suma doteraz spočítaných údajov a ďalší údaj z prúdu.

Metóda `reduce` vždy vytvára novú hodnotu pri spracovaní ďalšieho prvku. Ak by sme potrebovali vypočítať napr. priemernú hodnotu, potrebovali by sme spočítavať sumu čísiel a počet čísiel v prúde. Potom na konci vypočítať podiel týchto hodnôt. Čiže počas spracovania prvkov modifikovať dve premenné a na konci vypočítať podiel. Pre takýto typ operácie je vhodnejšie použiť metódu `collect`. Najprv definujme triedu, potrebnú pre výpočet priemernej hodnoty

```
class VypocitavacPriemeru implements Consumer<Integer> {
    private int suma;
    private int pocet;

    public VypocitavacPriemeru() {
        suma = 0;
        pocet = 0;
```

```
    }

    public double dajPriemer() {
        return pocet > 0 ? (double)suma/pocet : 0;
    }

    @Override
    public void accept(Integer hodnota) {
        suma += hodnota;
        pocet ++;
    }

    public void kombinuj(VypocitavacPriemeru dalsi) {
        suma += dalsi.suma;
        pocet += dalsi.pocet;
    }
}
```

Výpočet priemernej hodnoty ročníkov (mohli by sme využiť metódu `average`)

```
double priemer = studenti
    .stream()
    .mapToInt(Student::dajRocnik)
    .collect(VypocitavacPriemeru::new,
        VypocitavacPriemeru::accept,
        VypocitavacPriemeru::kombinuj)
    .dajPriemer();
```

Parametre metódy `collect`:
1. parameter je referencia na funkciu (konštruktor), ktorú vytvorí novú inštanciu
2. parameter je metóda, ktorá začleňuje hodnoty z prúdu do výpočtu
3. parameter slúži pre kombináciu dvoch medzivýsledkov

Tiež môžeme využiť inú preťaženú verziu metódy `collect`, ktorej argument je inštancia implementujúca rozhranie `Collector<T,A,R>`. Ďalej existuje trieda `Collectors` obsahujúca viacero metód vracajúcich preddefinované „kolektory“.

Uvedené príklady spracovávajú prvky postupne (sériovo). Java API umožňuje vytvoriť aj prúdy údajov vhodné pre paralelné spracovanie. Takéto prúdy možno vytvoriť metódou `Collection.parallelStream`, prípadne `BaseStream.parallel`. Prúdy sú rozdelené na niekoľko podprúdov, ktoré sú spracovávané paralelne a na konci sú medzivýsledky skombinované.

`VypocitavacPriemeru` možno tiež použiť pre výpočet pomocou paralelného prúdu údajov.

Implementácie rozhraní

- univerzálne
- špeciálne
- súbežné
- obálkové
- praktické
- abstraktné

Univerzálne implementácie

Univerzálne rozhrania sú najčastejšie aplikované implementácie, určené pre bežné použitie.

Každá z univerzálnych implementácii zaisťuje všetky voliteľné implementácie metód, ktoré sú súčasťou rozhrania. Všetky umožňujú pracovať s prvkami, kľúčmi a hodnotami s hodnotu `null`. Súčasťou všetkých implementácii sú iterátory s rýchlim zlyhaním (*fail-fast iterators*), ktoré ukončia svoju činnosť pri neplatných súbežných modifikáciách.
Žiadna nie je synchronizovaná (nemožno naraz používať z viacerých vlákien). Všetky implementujú rozhranie `Serializable` a implementujú metódu `clone()`.

Univerzálne implementácie:

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Vo väčšine prípadov sú najvýhodnejšie implementácie `HashSet`, `ArrayList`, `HashMap`.

Špeciálne implementácie

Sú navrhnuté pre použitie v špeciálnych situáciach, vyznačujú sa neštandardnými výkonnostnými charakteristikami, obmedzeným uplatnením.

Špeciálne implementácie rozhrania `Set`
`EnumSet` – vysokovýkonná implementácia pre enumeračné typy
`CopyOnWriteArraySet` – je založená na poli s kopírovaním pri zápise. Vhodné pre viacvláknové aplikácie v ktorých sa množina málo mení, ale často prechádza.

Špeciálne implementácie rozhrania `List`
`CopyOnWriteArrayList` – podobne ako `CopyOnWriteArraySet`

Špeciálne implementácie rozhrania `Map`
`EnumMap` – výkonná implementácia ak sú kľúčmi enumeračné typy
`WeakHashMap` – odkladá slabé odkazy na svoje kľúče – ak na kluč neodkazuje žiadna referencia z vonku, potom môže byť dvojica (kľúč,hodnota) vymazaná garbage kolektorom
`IdentityHashMap` – je založená na hašovacej tabuľke – vhodná pre uloženie transformácii objektových grafov zo zachovaním topológie napríklad pri serializácii, alebo hĺbkovom kopírovaní.

Súbežné implementácie

Sú vhodné pre súbežné použitie. Implementácie sa nachádzajú v balíku `java.util.concurrent`.

Rozhranie `ConcurrentMap` rozširuje rozhranie `Map`. Toto rozhranie je implementované triedou
`ConcurrentHashMap`

Rozhranie `BlockingQueue` rozširuje rozhranie `Queue`. Toto rozhranie je implementované triedami:
`LinkedBlockingQueue`
`ArrayBlockingQueue`
`PriorityBlockingQueue`
`DelayBlockingQueue`
`SynchronousQueue`

Obálkové implementácie

Používajú sa v spojení s ďalšími typmi implementácii, pričom rozširujú, alebo obmedzujú ich funkčnosť

Synchronizačné obálky
Doplňujú do ľubovoľnej kolekcie automatickú synchronizáciu. Tieto obálky možno získať volaním metód triedy `Collections`:
`synchronizedCollection()`
`synchronizedSet()`
`synchronizedList()`
`synchronizedMap()`
`synchronizedSortedSet()`
`synchronizedSortedMap()`

príklad:
```
List<Type> list =
          Collections.synchronizedList(new ArrayList<Type>());
```

Nemenné obálky
Odstraňujú možnosť úpravy kolekcie. Pri opužití operácia, ktorá by mala zmeniť kolekciu spôsobia výnimku `UnsupportedOperationException`.
Pre výrobu týchto obálok slúžia metódy triedy `Collections`:
`unmodifiableCollection()`
`unmodifiableSet()`
`unmodifiableList()`
`unmodifiableMap()`
`unmodifiableSortedSet()`
`unmodifiableSortedMap()`

Obálky kontrolovaných rozhraní
Sú určené pre použitie s generickými kolekciami. Pre výrobu týchto obálok slúžia metódy triedy `Collections`, ktoré majú dva vstupne parametre. Prvý je kolekcia, druhý je typ prvku:
`checkedList()`

`checkedMap()`
`checkedSet()`
`checkedSortedMap()`
`checkedSortedSet()`
Tieto metódy vracajú dynamicky typovo bezpečné zobrazenie kolekcie. Pri pokuse o pridanie prvku s nesprávneho typu, spôsobia výnimku `ClassCastException`. Mechanizmus genericity poskytuje typovú kontrolu pri preklade, ktorú však možno obísť. Dynamicka typová kontrola týchto obálok vykonáva túto kontrolu vždy.

Praktické implementácie

Zobrazenie poľa ako zoznamu

`Arrays.asList()` – používa sa ako most medzi poliami a rozhraním API (bolo uvedené vyššie)

Nemenné kolekcia typu `List` s viacerými kópiami prvku
Metóda `Collections.nCopies(int pocet, T vzorovy)` vráti nemenný zoznam typu `List`, obsahujúci `pocet` krát objekt `vzorovy`.

príklad:
```
public static void main(String[] args) {
  List<String> zoznam =
     new ArrayList<String>(Collections.nCopies(5, "prazdny"));
  System.out.println(zoznam);
}
```
výstup:
```
[prazdny, prazdny, prazdny, prazdny, prazdny]
```

Nemenná sada typu Singleton

Metódy triedy `Collections`:
```
static <T> Set<T>          singleton(T o)
static <T> List<T>         singletonList(T o)
static <K,V> Map<K,V> singletonMap(K key, V value)
```
vrátia nemennú kolekciu obsahujúcu jediný prvok
Túto metódu možno využiť napr. pri odoberaní všetkých výskytov určitého prvku z kolekcie
```
c.removeAll(Collections.singleton(e));
```

Prázdne konštanty typu `Set`, `List` a `Map`

Trieda `Collections` obsahuje metódy, ktoré vracajú prázdne kolekcie typu `Set`, `List` a `Map`:
```
static <T> List<T>       emptyList()
static <K,V> Map<K,V>    emptyMap()
static <T> Set<T>        emptySet()
```