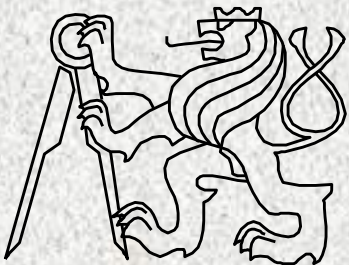


# Kolekce



BD6B36PJV 5  
Fakulta elektrotechnická  
České vysoké učení technické

# Obsah přednášky

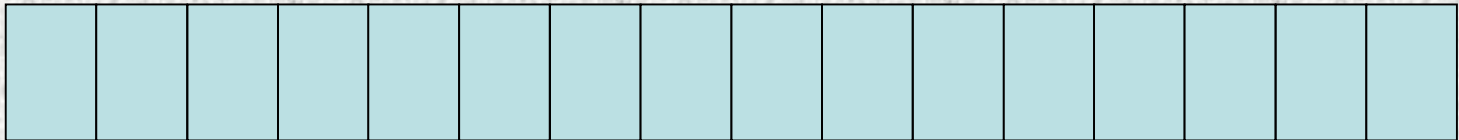
- Třída `java.util.Arrays`
- Genericita
- Kolekce
  - Rozhraní
  - Třídy
  - Algoritmy
- Třída `java.util.Collections`
- Komparátory
- Iterátory
- Hešování, `hashCode` a `equals`

# „Zobecněná pole“ a kolekce

- **Třída `Arrays`** (pole s dalšími službami)
  - Pole má neměnnou velikost a omezené služby
    - řešení třída `Arrays` + další služby
- **Kolekce** (rozhraní `Collection`, `Map`) - kontejnery pro ukládání a vyhledávání objektů
  - proměnná velikost
  - efektivnější práce než s poli, jednotný přístup
  - široký výběr možností služeb
  - efektivní implementace
  - vhodné pro implementaci ADT
  - implementované dodatečné funkce – vyhledávání, hashování

# Třída `java.util.Arrays`

`fill`, `binarySearch`,  
`sort`, `asList`,...



# Třída `java.util.Arrays`

Tato třída poskytuje řadu statických metod **usnadňujících práci s poli** (nad polem).

Metody jsou zpravidla přetížené pro různé typy.

- **`toString`** – výpis pole, převod na řetězec
- **`equals`** – test ekvivalence, porovnání obsahu dvou polí
- **`fill`** – vyplnění všech položek danou konstantou
- **`binarySearch`** – hledání v seřazeném poli či jeho části
- **`sort`** – vzestupné řazení, případně dle zadaného komparátoru
- **`asList`** – pro převod do kolekce
- **`copyOf`, `copyOfRange`** – kopie pole dle zadané délky

*Také pro vícerozměrná pole*

- **`deepEquals`** – porovnání hodnot dvou polí
- **`deepHashCode`** – porovnání kódů dvou polí, viz dále
- **`deepToString`** – výpis pole i vícedimezionálního

# java.util.Arrays – základní metody

```
public static void main(String[] args) {  
    int[] pole = {3,57,23,-100,1,7,3};  
    System.out.println(Arrays.toString(pole));  
    System.out.println("Index prvku 57 je " +  
        Arrays.binarySearch(pole,57));  
    Arrays.sort(pole);  
    System.out.println(Arrays.toString(pole));  
    System.out.println("Index prvku 57 je " +  
        Arrays.binarySearch(pole,57)); }  
}
```

převede pole na  
formátovaný řetězec

vyhledávání binárním  
půlením

[3, 57, 23, -100, 1, 7, 3]

Index prvku 57 je -8

..... nenalezeno, možný index,  
 $\langle \text{abs}(-8) - 1 \rangle = 7$

[-100, 1, 3, 3, 7, 23, 57]

Index prvku 57 je 6

..... tak je to správně

# java.util.Arrays funkce equals

```
public class ArraysEqualsZakladniDatovePrvky {
    final static int PO CET = 10;
    public static void main(String[] args) {
        int[] pole1 = new int[PO CET];
        int[] pole2 = new int[PO CET * 2];
        int[] pole3 = new int[PO CET];
        for (int i = 0; i < pole1.length; i++) {
            pole1[i] = i;
        }
        System.arraycopy(pole1, 0, pole2, 0, pole1.length);
        System.arraycopy(pole1, 0, pole3, 0, pole1.length);
        System.out.println("Pole 1 a 2 se rovnaji: " +
            Arrays.equals(pole1, pole2));
        System.out.println("Pole 1 a 3 se rovnaji: " +
            Arrays.equals(pole1, pole3));
        System.out.println("Zmena prvku pole3");
        pole3[3] = 123;
        System.out.println("Pole 1 a 3 se rovnaji: " +
            Arrays.equals(pole1, pole3));
    }
}
```

```
Pole 1 a 2 se rovnaji: false
Pole 1 a 3 se rovnaji: true
Zmena prvku pole3
Pole 1 a 3 se rovnaji: false
```

# Kolekce (kontejnery)

- Objekty, které slouží k ukládání, načítání, zpracování a přenášení konečného většího, předem neznámého počtu objektů, prvků stejného typu (umožňuje zpracovat agregovaná (sdružená) data)
- Implementační datové struktury dosud známé - nevýhody:
  - **pole** – konečný počet prvků, přístup indexem, řazení není vlastností pole, implementace datových typů nepružná
  - **seznamy** – jednoúčelové programy, zcela primitivní struktury
- **Java Collection Framework** – jednotné prostředí pro manipulaci se „skupinami“ objektů (kolekcemi resp. kontejnery objektů)
  - Jedná se o implementační prostředí datových typů **polymorfního charakteru**
    - Umožňuje ukládání objektů, jejich získávání a zpracování, dále vypočítat souhrnné údaje, apod.
    - Typickými skupinami objektů jsou **abstraktní datové typy**: množiny, seznamy, fronty, tabulky, ..



# Kolekce – výhody/nevýhody

## Výhody

- *Ulehčení programování,*
  - *menší kód,*
  - *rychlejší algoritmy*
  - *„neomezený rozsah“ počtu ukládaných objektů*
- *Zvýšení čitelnosti programů, opakované použití SW*
- *Lepší přenositelnost, kompatibilita*
- *Možnost pozdějších úprav, změnou dat či dalšími API*

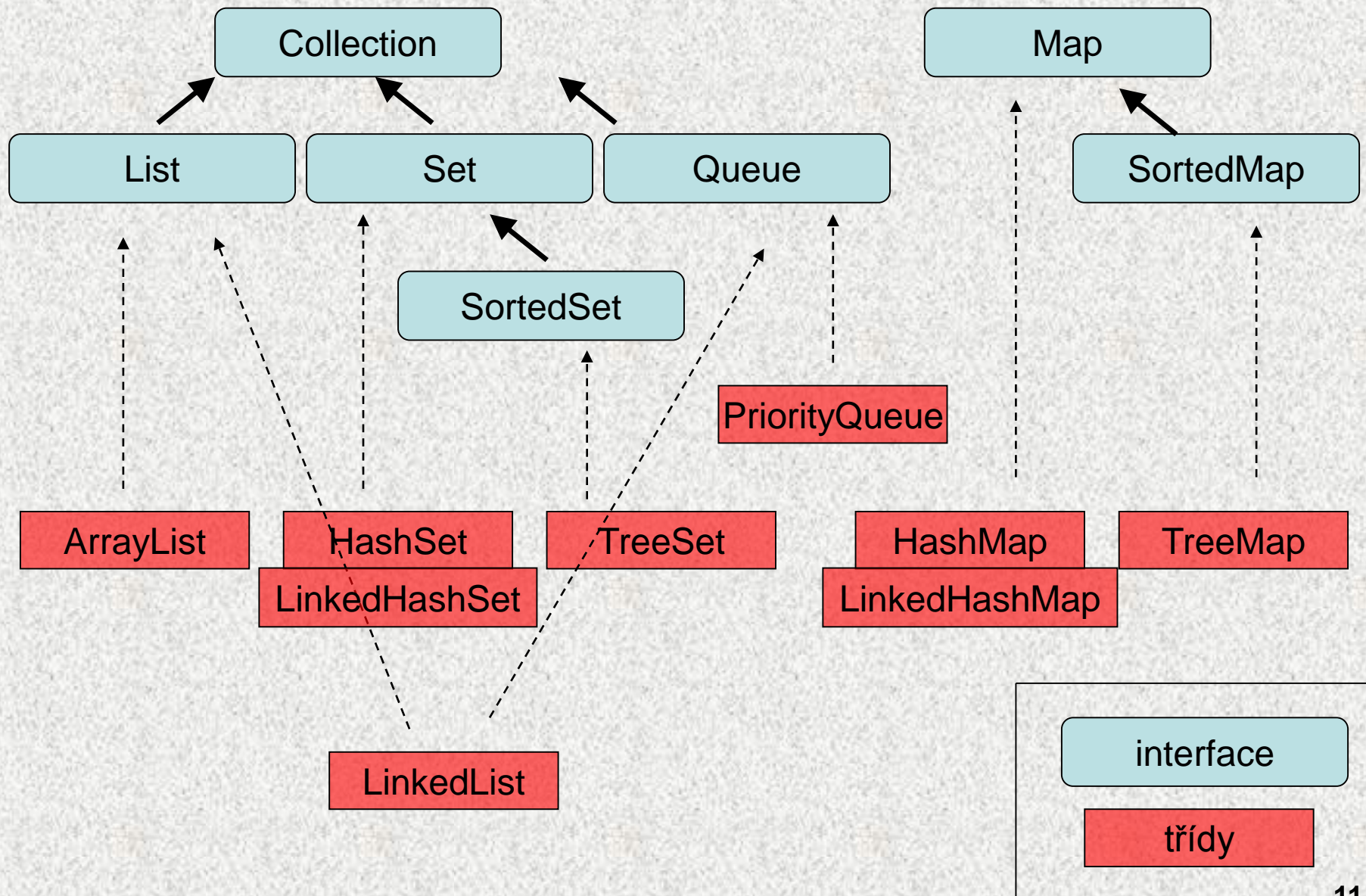
## Nevýhody oproti poli:

- ?? od Javy 1.5. „lze“, díky autoboxingu, lze vkládat přímo primitivní typy, někdy vadí kontrola typu
- !! pomalejší přístup k objektům

# Kolekce

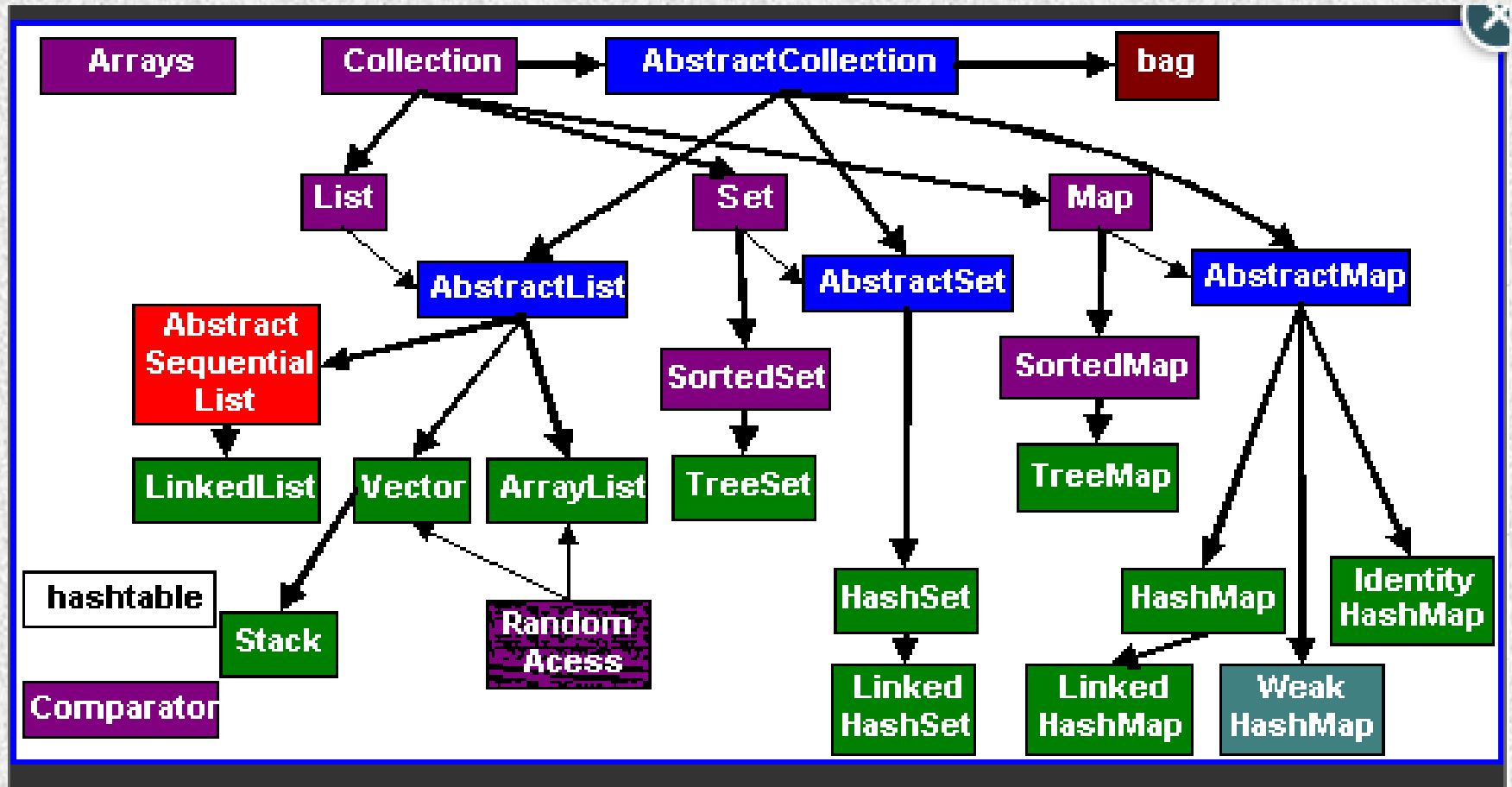
- Základní pojmy:
  - **Rozhraní** – definuje množinu abstraktních metod pro zpracování prvků kolekcí, rozhraní (interface) jsou v hierarchii
  - **Třídy**, které implementují (realizují) rozhraní kolekcí
  - **Algoritmy kolekcí** – polymorfní realizace metod tříd a to jak metod rozhraní, tak vlastní metody tříd
- Nejznámější, příklad:
  - **ArrayList** - „ekvivalent pole“ „index“ >>prvek
    - `add()`, `set()`, `get()`
  - **HashSet** - „ekvivalent množiny“
    - `add()`, `remove()`, `contains()`
  - **HashMap** - „ekvivalent tabulky“

# Kolekce, základní rozhraní a třídy



# Kolekce

- Java Collection Framework – jednotné prostředí pro manipulaci se „skupinami“ objektů



# Genericita

- Do kolekcí bylo možné vkládat objekty libovolného typu
- Při výběru použitelné po přetypování, výhoda
- Nevýhoda – nutná programátorská kontrola
- Od JDK 1.5 je možné vkládané objekty typovat, tj. předepsat, jaké typy je možné vkládat, JDK zkontroluje – bezpečnější kód
- Typ objektu se vkládá do <>

# Genericita

- Původně (i teď) do kolekcí je možné vkládat instance třídy Object a při výběru přetypovávat zpět
  - Nevýhoda – špatná kontrola typu skutečného objektu
  - Chyby až za běhu
- Od JDK1.5 možnost typování kolekcí, tj. je možné určit instance, které třídy lze do kolekce ukládat
- **Zástupce typu vkládaných objektů** (třída či interface) se uzavírá do `<>`,

Příklad

```
public class Clovek {...}  
ArrayList <Clovek> ar = new ArrayList <Clovek>();
```

Rozšíření:

```
boolean containsAll( Collection <?> c )  
boolean addAll( Collection < ? extends E > c )
```

Zastupuje všechny možné objekty, přístupné rozhraním

Zastupuje všechny možné objekty, přístupné rozhraním a jejich potomky

# Genericita, příklad

```
ArrayList seznam = new ArrayList(); // bez genericity
seznam.add("Ahoj");
seznam.add("baf");
seznam.add("Ivanka");
seznam.add(new Integer(3));
for (int i = 0; i < seznam.size(); i++) {
    int d = ((String)seznam.get(i)).length();
    System.out.println("delka retezce"+seznam.get(i)+"je " + d);
}
```

```
ArrayList<String> seznam = new ArrayList<String>();
seznam.add("Ahoj");
seznam.add("baf");
seznam.add("Ivanka");
//seznam.add(new Integer(3)); // je možné vkládat jen String
for (int i = 0; i < seznam.size(); i++) {
    int d = seznam.get(i).length();
    System.out.println("delka retezce"+seznam.get(i)+"je " + d);
}
```

# Genericita, příklad

```
public class GenericitaBox <T> {  
    private T t;  
    public void pridat(T t){  
        this.t=t;  
    }  
    public T zjistit(){  
        return t;  
    }  
  
    public static void main(String[] args) {  
        GenericitaBox<Integer> obj = new GenericitaBox<Integer>();  
        obj.pridat(new Integer(77));  
        Integer cele = obj.zjistit();  
        System.out.println(" " + cele);  
        GenericitaBox<String> objS = new GenericitaBox<String>();  
        objS.pridat("gugu");  
        String str = objS.zjistit();  
        System.out.println(" "+ str);  
    }  
}
```

77
gugu



# Kolekce, přehled I

„Skupina objektů“ v operační paměti, organizovaná dle JCF (Java Collection Framework), balíček `java.util` a zahrnuje zejména:

## 1. Rozhraní:

### `Collection<E>`

- nejobecnější rozhraní kolekcí objektů
- **společný předek**, slouží i k předávání kolekcí
- zajištěn polymorfismus

### `List<E>`

- indexovaná sbírka objektů označovaná jako sekvence – „seznamy“
- pořadí prvků je významné
- je možné indexovat
- **duplicita prvků povolena**

### `Set<E>` (`SortedSet <E>`)

- reprezentace množiny, **sbírka unikátních objektů** (příp. uspořádaná)
- nelze indexovat – pomocné indexování jen pomocí tzv. iterátorů

### `Map<K,V>` (`SortedMap <K,V>`)

- **sbírka dvojic** (zobrazení) objektů, mapuje klíč na hodnoty
- mapuje „key → (jedinou) value“ (příp. uspořádaná)
- neobsahuje duplicitní klíče

### `Queue<E>`

- fronta ( FIFO )

-----

`Comparator<T>`, `java.lang.Comparable` - pro porovnávání objektů

`Iterator<E>`, `ListIterator<E>`, `RandomAccess` - přístup k prvkům

# Kolekce, přehled II

## 2. Třídy implementující Collection:

**ArrayList**

– „pole proměnné délky“,

**LinkedList**

– „spojový seznam“

**HashSet, LinkedHashSet**

– „reprezentace množiny“

**HashMap, LinkedHashMap**

– „mapa, klíč~hodnota“

**TreeSet**

– „seřazená množina“

**TreeMap**

– „seřazená mapa“

## 3. Algoritmy

pro řazení, přesouvání, doplňování, kopírování a vyhledávání

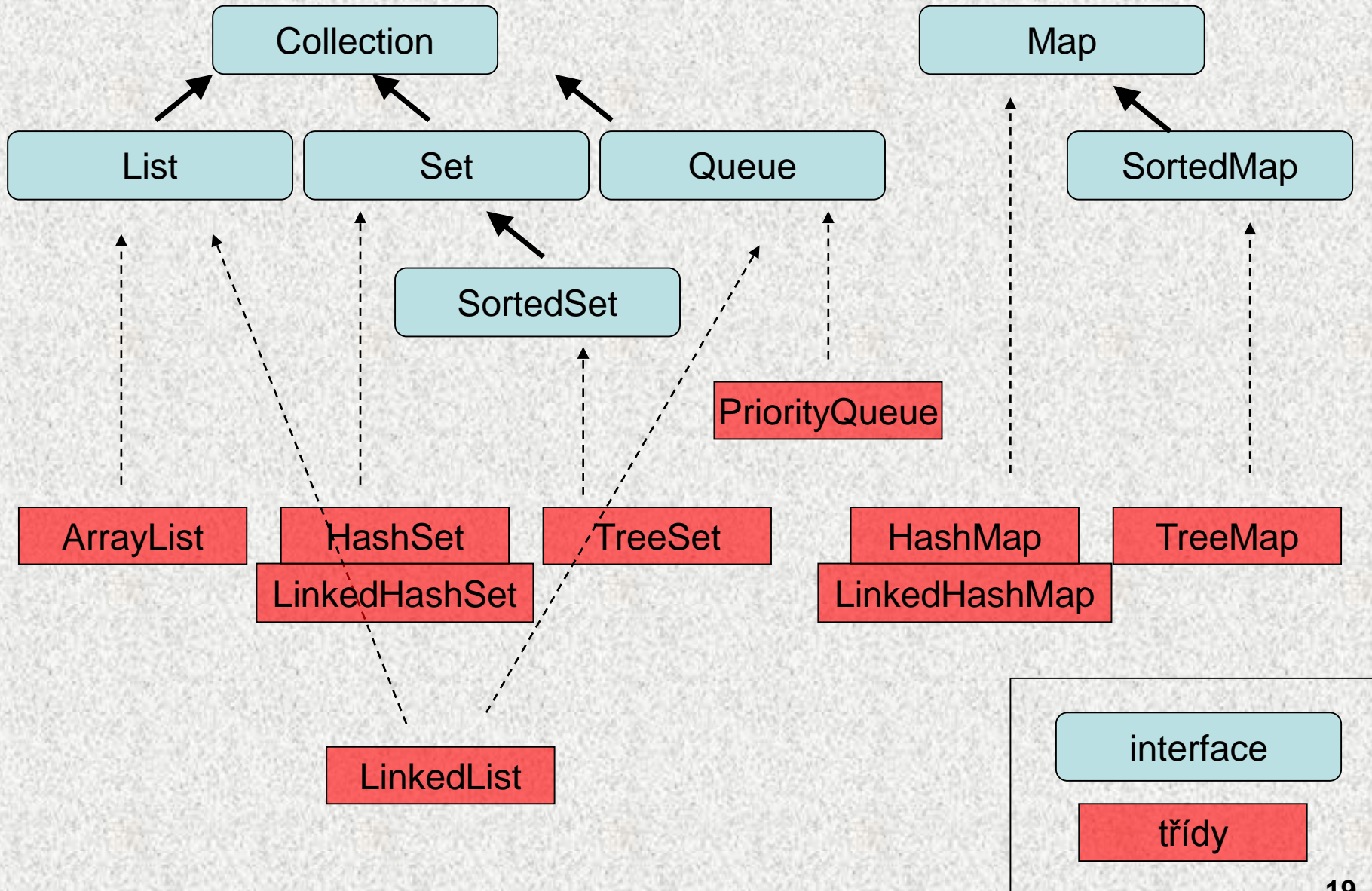
### • *Další třídy*

*Arrays, Collections* – „lepší pole“

*Vector, Stack, Hashtable, Properties*

*PriorityQueue, ArrayDeque*

# Kolekce, základní rozhraní a třídy



# Rozhraní Collection<E>

Collection<E> **extends** Iterable<E>

umožňuje jednotlivé i hromadné operace pro dotazy, převody a modifikace.

**boolean** add( E o )

**boolean** addAll (Collection < ? **extends** E > c )

**void** clear( ) – vyloučení všech prvků

**boolean** remove( Object o )

**boolean** removeAll (Collection <?> c) - odstraní všechny patřící do c

**boolean** retainAll (Collection <?> c) - odstraní všechny mimo patřící do c

**int** size( )

**boolean** isEmpty( )

**boolean** contains( Object o )

**boolean** containsAll( Collection <?> c ) - **true**, když všechny prvky c

Iterator<E> iterator( )

- vrátí objekt pro probírku sbírky

Object[] toArray( )

- vytvoří pole typu Object

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

- vytvoří pole konkrétního typu

Přehled

# Rozhraní List <E>

- Indexovaný přístup k prvkům skupiny, uspořádané, označené jako sekvence, mohou být duplicitní prvky,
- rozšiřuje `Collection`

`boolean addAll (int index, Collection <? extends E> c )`

`boolean equals (Object o)` – porovnání seznamů

`E get (int index)`

`E set (int index, E element)`

`E remove (int index)`

`int indexOf (Object o)` - hledání zepředu

`int lastIndexOf (Object o)` - hledání zezadu

`ListIterator<E> listIterator ()`

`ListIterator<E> listIterator (int index)`

`List<E> subList (int fromIndex, int toIndex)` - pohled

Přehled

# Rozhraní Set<E>, SortedSet<E>

## Set<E>

- v množině nesmějí být duplikované prvky a nanejvýš jedna reference `null`
- prvky neuspořádané,
- nepřidává další metody k rozhraní `Collection`.

## SortedSet<>

- uchovává prvky ve vzestupném přirozeném pořadí
- `SortedSet` **extends** `Set`
- přidává metody:

`Comparator<? super E> comparator( )`

`E first( )` - vrací první ( nejmenší ) prvek

`E last( )` - vrací poslední ( největší ) prvek

`SortedSet<E> headSet( E toElement )`

`SortedSet<E> subSet( E fromElement, E toElement )`

`SortedSet<E> tailSet( E fromElement )`

# Rozhraní `Queue<E>`

- reprezentuje FIFO frontu
- rozšiřuje `Collection`

`boolean add(E o)` – vrací `true`, při neúspěchu výjimku, vkládání

`boolean offer(E o)` – vrací `true`, při neúspěchu `false`

`boolean remove(Object o)` – výběr, vrací výjimku, při neúspěchu

`E poll()` – výběr, vrací `null`, při neúspěchu

`E element()` – inspekce (ponechá), vrací výjimku, při neúspěchu

`E peek()` – inspekce (ponechá), vrací `null`, při neúspěchu

# Rozhraní Map < K, V >

- **Map** znamená zobrazení <keys | values>, tedy klíčů ( množinu objektů ) na hodnoty (množinu objektů), abstrakce mat. funkce
- nedědí od **Collection**

**V** get (Object key) – podle klíče hodnotu  
**int** size () – počet prvků  
**void** clear () – ruší prvky  
**boolean** isEmpty () – test na práznost  
**V** put (K key, V value) – vložení prvku  
**Object** remove (Object key)  
**boolean** containsKey (Object key)  
**boolean** containsValue (Object value)  
**void** putAll (Map<? **extends** K, ? **extends** V> m) - odjinud  
**Set** <Map.Entry<K,V>> entrySet () – vrací množinu dvojic  
**Set**<K> keySet () – vrací množinu klíčů  
**Collection** <V> values ( ) – vrací hodnoty typu V

Přehled



# Rozhraní `SortedMap<K, V>`

- `SortedMap` **extends** `Map`
  - uchovává dvojice uspořádané v přirozeně vzestupném pořadí:

Analogie `SortedSet`

`K firstKey()` - vrací první ( nejmenší ) klíč

`K lastKey()` - vrací poslední ( největší ) prvek

`SortedMap<K, V> headMap(Object toKey)`

`SortedMap<K, V> subMap(K fromKey, K toKey)`

`SortedMap<K, V> tailMap(K fromKey)`

# Třída `ArrayList`

- `ArrayList` – kolekce reprezentující seznam, prvky jsou indexovány, jeden prvek může být v kolekci vícekrát (na rozdíl od množiny)
- Implementuje rozhraní `List`
- `new ArrayList()` ;
  - konstruktor, který vytvoří prázdný seznam s počáteční kapacitou 10
  - po pokusu o vložení 11. prvku dojde ke změně velikosti vnitřního pole (na 1,5 násobek předchozí velikosti) a překopírování dat
  - průměrná složitost vložení jednoho prvku je konstantní



Zobecněné „pružné“ pole

# ArrayList – příklad

```
public static void main(String[] args) {  
    ArrayList seznam = new ArrayList();  
    seznam.add(new Integer(5));  
    seznam.add("Petr");  
    seznam.add("Pavel");  
    seznam.add(new Integer(25));  
    System.out.println("Seznam: " + seznam);  
}
```

- do kolekce se vkládají objekty – potomci třídy Object
- to může být komplikace při vybírání prvků
- starý způsob (před Javou 1.5)
- slabá typová kontrola

Seznam: [5, Petr, Pavel, 25]

# Typované kolekce - bez typů

```
public static void main(String[] args) {  
    ArrayList seznam = new ArrayList();  
    seznam.add("Ahoj");  
    seznam.add("baf");  
    seznam.add("Ivanka");  
    seznam.add(new Integer(3));  
    for (int i = 0; i < seznam.size(); i++) {  
        int d = ((String) seznam.get(i)).length();  
        System.out.println("delka" + seznam.get(i) + "je" + d);  
    }  
}
```

- seznam.get(i) vrací Object, před použitím jako String je nutné přetypování

Do kolekce mohu vkládat jakékoli **Objecty**

Exception in thread "main"  
java.lang.ClassCastException:  
java.lang.Integer cannot be cast  
to java.lang.String

# Typované kolekce - s typem

- moderní způsob (od Javy 1.5)
- silná typová kontrola, bezpečné

```
public static void main(String[] args) {  
    ArrayList<String> seznam = new ArrayList<String> ();  
    seznam.add("Ahoj");  
    seznam.add("baf");  
    seznam.add("Ivanka");  
    seznam.add(new Integer(3));  
    for (int i = 0; i < seznam.size(); i++) {  
        int d = seznam.get(i).length();  
        System.out.println("delka" + seznam.get(i) + "je" + d);  
    }  
}
```

Do kolekce mohu vkládat **Stringy** (a potomky, pokud by String nebyl final), chyba odhalena již při překladu

- seznam.get(i) vrací String, před použitím jako String není nutné přetypování

Nic nehrozí, jsou tam řetězce

# Příklad – Tabulka, kontejner ArrayList I

```
static ArrayList inicializaceTabulky() {  
    ArrayList tab = new ArrayList();  
    for (int i = MIN; i <= MAX; i++) {  
        tab.add(0);  
        //~ tab[MIN]=0;... tab[MAX]=0;  
    }  
    return tab;  
}
```

# Příklad – Tabulka, kontejner ArrayList II

```
static ArrayList tabulka(ArrayList tab) {
System.out.println("zadejte řadu celých zakončenou
nulou");
int cislo = sc.nextInt();
while (cislo != 0) {
if (cislo >= MIN && cislo <= MAX) {
Integer k = (Integer) tab.get(cislo); // k=tab[cislo];
int j = k.intValue() + 1;
tab.set(cislo, j); // tab[cislo]=j;
}
cislo = sc.nextInt();
}
return tab;
}
```

# Příklad – Tabulka, kontejner ArrayList III

```
public static void main(String[] args) {  
    ArrayList tab = inicializaceTabulky();  
    System.out.println(" " + tab);  
    tabulka(tab);  
    System.out.println(" " + tab);  
}
```

Výsledek:

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

zadejte řadu celých čísel zakončenou nulou

2 2 2 9 9 9 9 0

[0, 0, 3, 0, 0, 0, 0, 0, 0, 4, 0]



# Seznam versus množina

## Seznam

- prvky se mohou opakovat

```
List l = new ArrayList();  
l.add("Pepa");  
l.add("Jana");  
l.add("Pepa");  
l.add("Jana");  
System.out.println(l);
```

[Pepa, Jana, Pepa, Jana]

## Množina

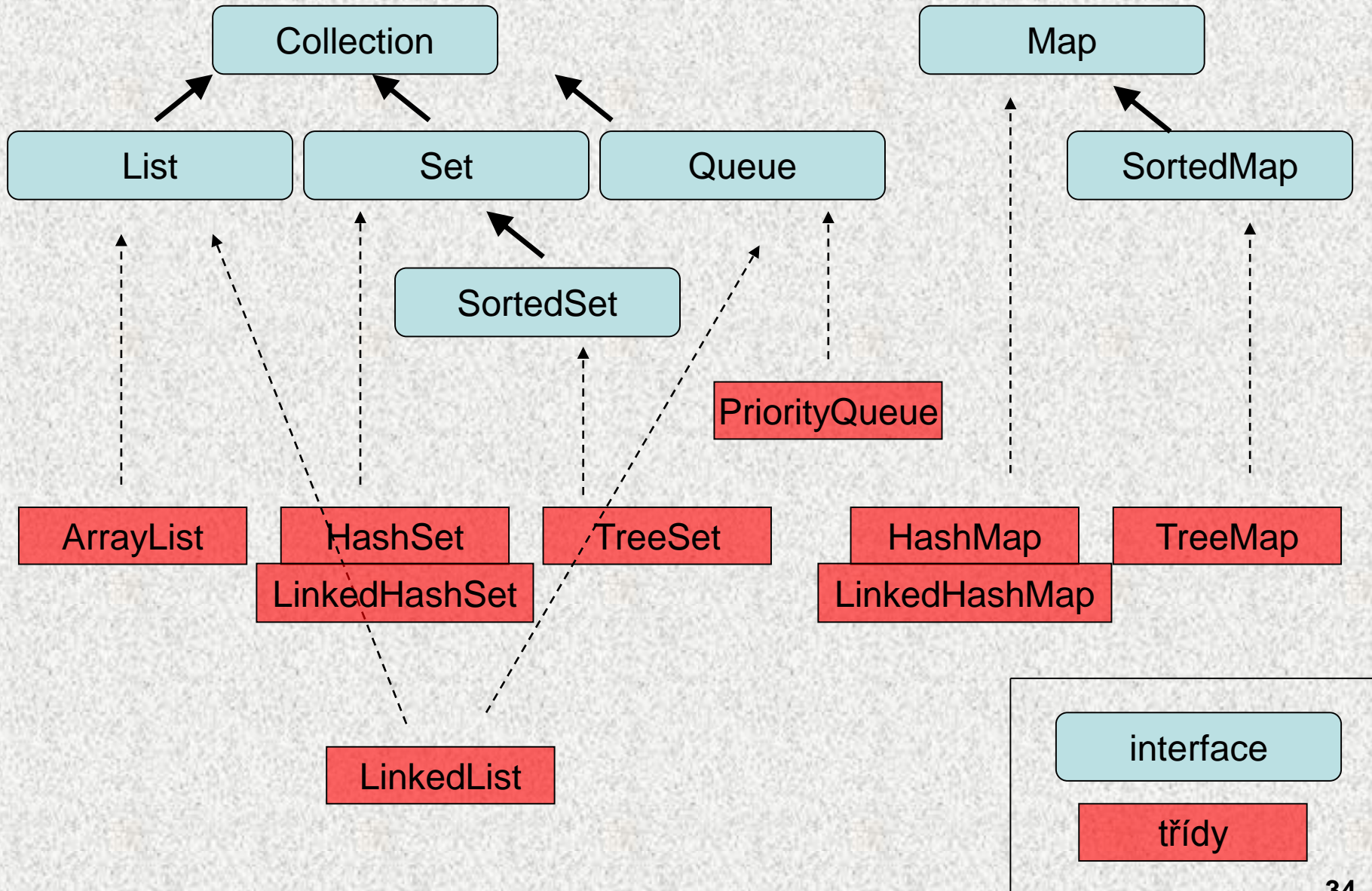
- prvky jsou unikátní, neopakují se

```
Set m = new HashSet();  
m.add("Pepa");  
m.add("Jana");  
m.add("Pepa");  
m.add("Jana");  
System.out.println(m);
```

[Jana, Pepa]

Během zjišťování, zda je prvek v množině se využívá metoda **equals!!**

# Kolekce, základní rozhraní a třídy



# Třídy `ArrayList<E>` a `LinkedList<E>`

- Třída `ArrayList` uchovává reference objektů v poli, možno „indexovat“.
  - Operace `size()`, `isEmpty()`, `get()`, `set()`, `add()`, `iterator()`, a `listIterator()` probíhají v konstantním čase, ostatní v lineárním.
  - **Nevhodná** pro časté změny seznamu, pro implementaci zásobníku, fronty
- Třída `LinkedList` implements `Queue` (také) - fronta
  - Kolekce objektů je realizována technikou **obousměrného spojového seznamu**. Příslušné operace mají tedy lineární časovou složitost. Rychlejší vkládání na začátek kolekce.
  - Rozhodnutí o použití `ArrayList` či `LinkedList` můžeme oddálit typem **interface**, všechny proměnné, parametry metod typujeme na `List`, pouze na jediném místě programu bude

```
List list = new ArrayList();  
List list = new LinkedList();
```



Zobecněný  
seznam

# Příklad na třídu LinkedList, fronta

```
public class FrontaLinkedList {
    public static void main(String[] args) {
        Queue<String> fifoFronta = new LinkedList<String>();
        fifoFronta.add("5");
        fifoFronta.add("1");
        fifoFronta.add("3");
        System.out.println(fifoFronta);

        fifoFronta.add("2");
        fifoFronta.add("4");
        System.out.println(fifoFronta);

        System.out.println("Na cele je: " + fifoFronta.element());

        while (fifoFronta.isEmpty() == false) {
            System.out.print(fifoFronta.remove() + ", ");
        }
    }
}
```

**remove z**  
interface Queue

**element z**  
interface Queue

[5, 1, 3]  
[5, 1, 3, 2, 4]  
Na cele je: 5  
5, 1, 3, 2, 4,

# Příklad na třídu LinkedList - zásobník

```
class Zasobnik<E> {  
    private LinkedList<E> zasob = new LinkedList<E>();  
  
    public void add(E elem) {  
        zasob.addFirst(elem);  
    }  
  
    public E remove() {  
        return zasob.removeFirst();  
    }  
  
    public E get() {  
        return zasob.getFirst();  
    }  
  
    public boolean isEmpty() {  
        return zasob.isEmpty();  
    }  
}
```

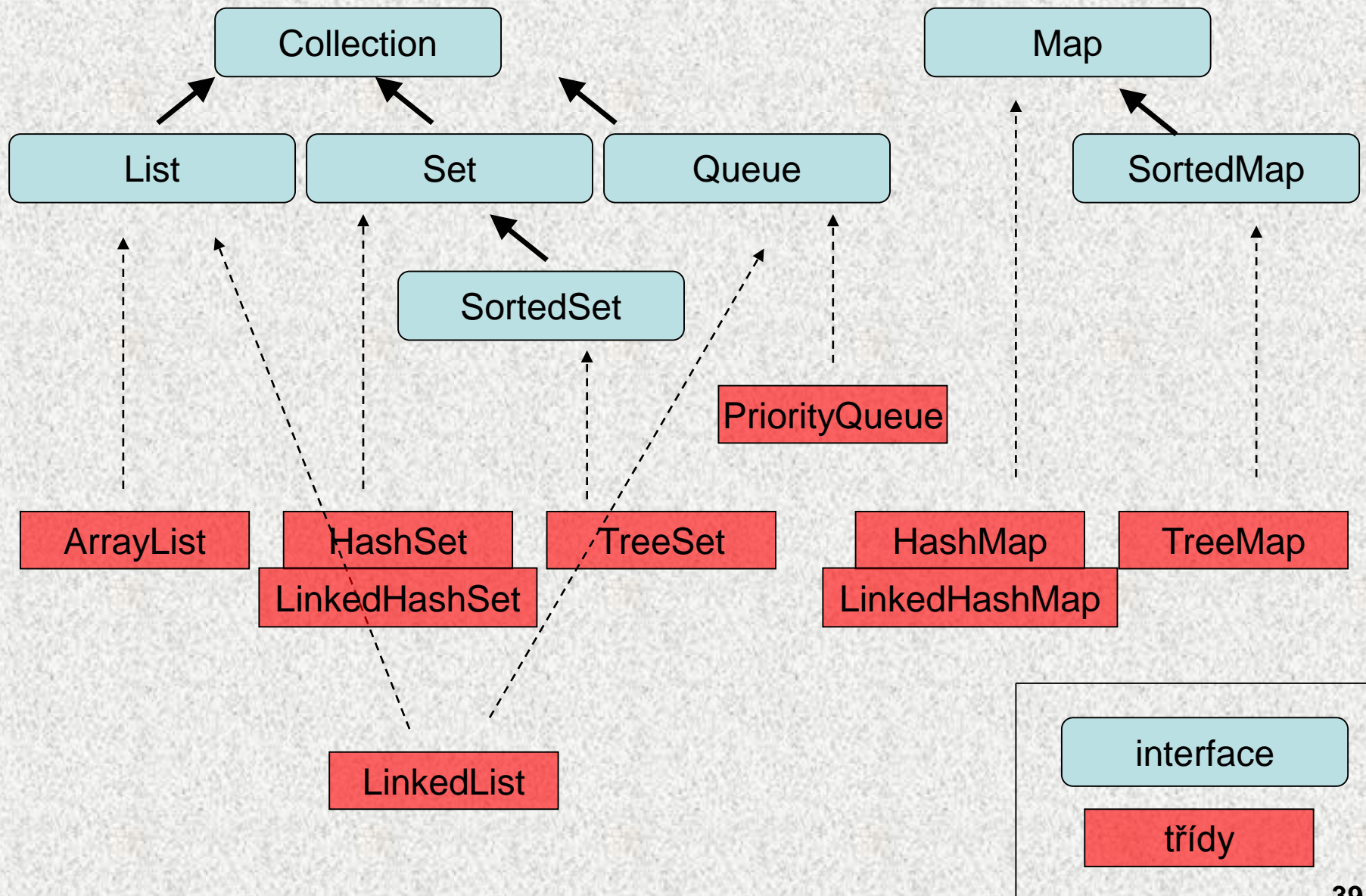
# Příklad na třídu LinkedList, zásobník

```
public class PouzitiZasobniku {
    public static void main(String[] args) {
        Zasobnik<String> zs = new Zasobnik<String>();
        zs.add("prvni");
        zs.add("druhy");
        zs.add("treti");
        System.out.println(zs.get());
        while (zs.isEmpty() == false) {
            System.out.print(zs.remove() + ", ");
        }

        System.out.println();
        Zasobnik<Integer> zi = new Zasobnik<Integer>();
        zi.add(new Integer(8));
        zi.add(new Integer(9));
        while (zi.isEmpty() == false) {
            System.out.print(zi.remove() + ", ");
        }
    }
}
```

treti,  
druhy,  
prvni,  
  
9,  
8,

# Kolekce, základní rozhraní a třídy




# Třídy HashSet a TreeSet

- Implementace rozhraní **Set** resp. **SortedSet**
- **HashSet** neudržuje prvky seřazené na rozdíl od **TreeSet**
- Vkládání do **TreeSet** pomalejší
- Rychlost práce s **HashSet** závisí na kvalitě funkce **hashCode ()**
- Indexovaný přístup do množin **jen** přes iterátory, viz dále
- Existuje **LinkedHashSet**, kombinace předchozích



Zobecněná  
množina



Zobecněná  
„uspořádaná  
“množina



# Příklad na třídy HashSet a TreeSet

```
public class HashSetATreeSet {
    public static void naplneniATisk(Set<String> st) {
        st.add("treti");
        st.add("prvni");
        st.add("druhy");
        // pokus o vlozeni stejneho prvku
        if (st.add("treti") == false) {
            System.out.println("'treti' podruhe nevlozen");
        }
        System.out.println(st.size() + " " + st);
        for (String s: st) {
            System.out.print(s + ", ");
        }
        if (st.contains("treti") == true) {
            System.out.println("\n'treti' je v mnozine");
        }
        st.remove("treti");
        System.out.println(st);
        st.clear();
    }
}
```

# Příklad na třídu HashSet

```
public static void main(String[] args) {  
    System.out.println("HashSet:");  
    naplneniATisk(new HashSet<String>());  
    System.out.println("TreeSet:");  
    naplneniATisk(new TreeSet<String>());  
}
```

HashSet:

'treti' podruhe nevlozen  
3 [prvni, trei, druhy]  
prvni, trei, druhy,  
'treti' je v mnozine  
[prvni, druhy]

TreeSet:

'treti' podruhe nevlozen  
3 [druhy, prvni, trei]  
druhy, prvni, trei,  
'treti' je v mnozine  
[druhy, prvni]

# Eratosthenovo síto - HashSet I

- Funkce pro vytvoření množiny prvočísel do *max*

```
static HashSet sito(int max) {  
    HashSet mnozina = new HashSet();  
    for (int i=2; i<=max; i++) mnozina.add(i);  
    int p = 2;  
    int pmax = (int)Math.sqrt(max);  
    do {  
        // vypuštění všech násobků čísla p  
        for (int i=p+p; i<=max; i+=p)  
            mnozina.remove(i);  
        do { // hledání nejbližšího prvočísla k p  
            p++;  
        } while (!mnozina.contains(p));  
    } while (p<=pmax);  
    return mnozina;  
}
```

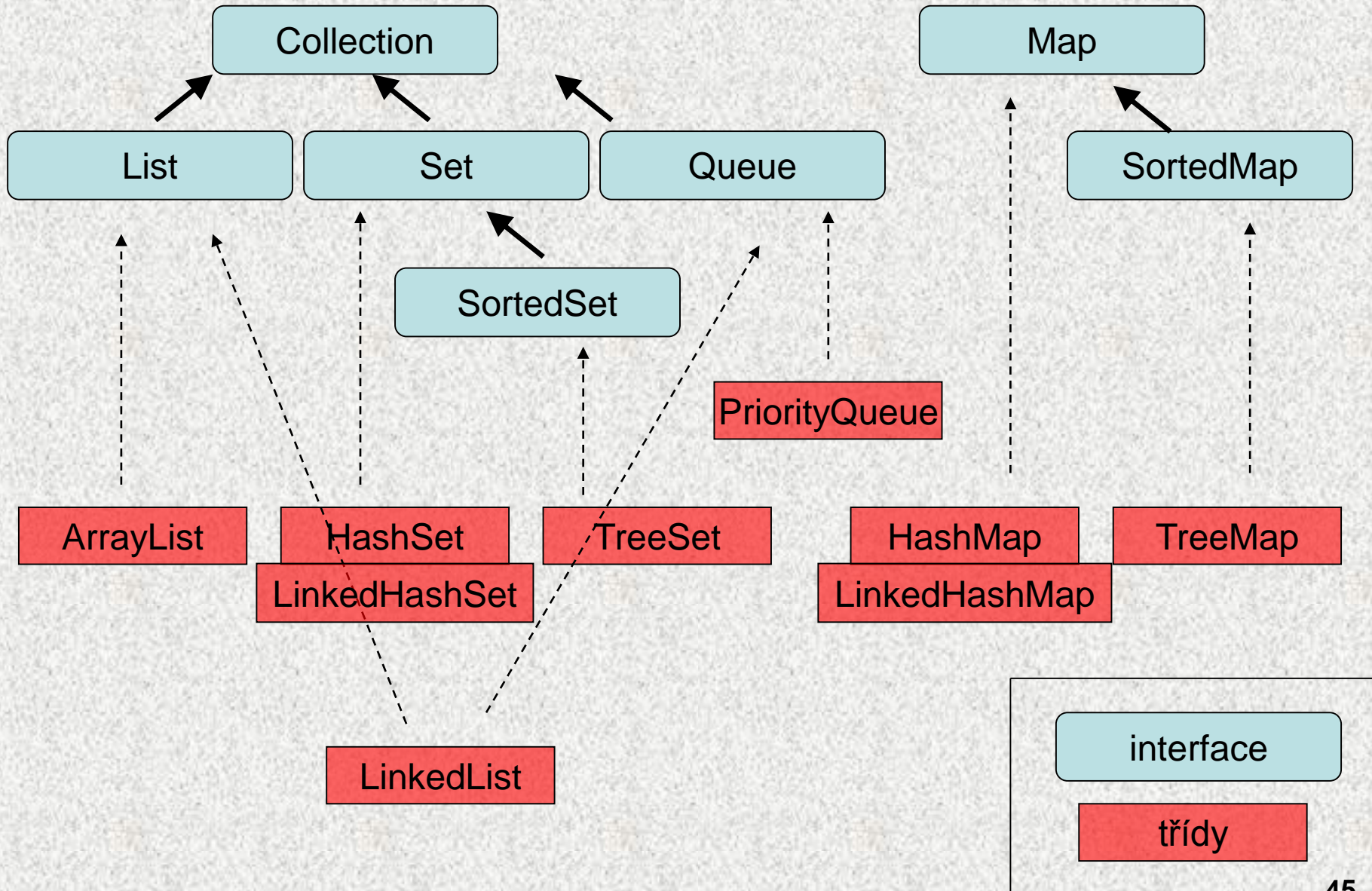
# Eratosthenovo síto - HashSet II

- Funkce pro výpis množiny

```
static void vypis(HashSet mnozina, int max) {
    for (int i=2; i<max; i++)
        if (mnozina.contains(i))
            System.out.print(i + ", ");
}

public static void main(String[] args) {
    System.out.println ("zadejte max");
    int max = sc.nextInt();
    HashSet mnozina = sito(max);
    System.out.println("Množinou, prvočísla od 2 do "+ max);
    vypis(mnozina, max);
}
```

# Kolekce, základní rozhraní a třídy



# Třídy HashMap a TreeMap

- Implementace rozhraní **Map** resp. **SortedMap**
- **HashMap** neudržuje dvojice seřazené na rozdíl od **TreeMap**
- Položka **key** je unikátní
  - **Map** - zobrazení <keys | values>, klíčů (množinu objektů) na množinu objektů
  - **V put (K key, V value)** – vložení prvku
- Vkládání do **TreeMap** pomalejší
- Rychlost práce s **HashMap** závisí na kvalitě funkce **hashCode ()** , viz dále
- Indexovaný přístup do map jen přes iterátory, viz dále
- Existuje **LinkedHashMap** , kombinace předchozích

Zobecněná  
tabulka

Zobecněná  
„uspořádaná  
“tabulka

# Příklad na třídu HashMap

```
class Vaha {
    double vaha;
    Vaha(double vaha) { this.vaha = vaha; }

    public String toString() {
        return "" + vaha;
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Vaha == false)
            return false;
        boolean stejnaVaha = (vaha == ((Vaha) o).vaha);
        return stejnaVaha;
    }

    public int hashCode() {
        return (int) vaha;
    }
}
```

# Příklad na třídu HashMap

```
public class HashMapZakladniPouziti {
    public static void main(String[] args) {
        HashMap<String, Vaha> hm = new HashMap<String, Vaha>();
        System.out.println("Mapa je prazdna: " + hm.isEmpty()
            + " a obsahuje prvku: " + hm.size());
        hm.put("Pavel", new Vaha(85));
        hm.put("Venca", new Vaha(105));
        hm.put("Karel", new Vaha(85));
        System.out.println("Mapa je prazdna: " + hm.isEmpty()
            + " a obsahuje prvku: " + hm.size());
        System.out.println("Mapa: " + hm);
        hm.remove("Karel");
        System.out.println("Mapa: " + hm);
        hm.put("Karel", new Vaha(70));
        System.out.println("Mapa: " + hm);
        Vaha v = hm.get("Venca");
        System.out.println("Venca vazi: " + v);
        if (hm.containsKey("Pavel")) {
            System.out.println("Pavel vazi: " + hm.get("Pavel"));
        }
        if (hm.containsValue(new Vaha(105)) == true) {
            System.out.println("Nekdo vazi 105 kg");
        }
    }
}
```



# Příklad na třídu HashMap

```
hm.get("Pavel").vaha += 10; // Pavel ztlousnul
System.out.println("Lidi: " + hm.keySet());
// ArrayList ar = (ArrayList) hm.values(); // nelze
// HashSet hs = (HashSet) hm.values(); // nelze
Collection<Vaha> col = hm.values();
Iterator<Vaha> it = col.iterator();
it.next().vaha += 7; // nekdo ztlousnul
System.out.println("Vahy: " + col);
System.out.println("Mapa: " + hm);

double[] poleVah = new double[col.size()];
int i = 0;
for (Vaha va : col) {
    poleVah[i] = va.vaha;
    i++;
}
System.out.print("Pole vah: ");
System.out.println(Arrays.toString(poleVah));
}
}
```

# Výsledek příkladu na třídu HashMap

Mapa je prazdna: true a obsahuje prvku: 0

Mapa je prazdna: false a obsahuje prvku: 3

Mapa: {Pavel=85.0, Venca=105.0, Karel=85.0}

Mapa: {Pavel=85.0, Venca=105.0}

Mapa: {Pavel=85.0, Venca=105.0, Karel=70.0}

Venca vazi: 105.0

Pavel vazi: 85.0

Nekdo vazi 105 kg

Lidi: [Pavel, Venca, Karel]

Vahy: [77.0, 95.0, 105.0]

Mapa: {Pavel=95.0, Venca=105.0, Karel=77.0}

Pole vah: [77.0, 95.0, 105.0]

# Třída java.util Collections

- Třída pro práci s kolekcemi, obdoba třídy **Arrays** pro pole
- Implementuje rozhraní **Collection**

`int binarySearch(List, Object)` - vyhledávání, předpokládá seřazení

`void copy(List, List)` - kopie prvků jednoho seznamu do druhého

`void fill(List, Object)` – všechny prvky seznamu budou nahrazeny hodnotou specifikovanou jako 2. parametr

`Object max(Collection, Comparator) min` – Maximum, minimum

`void reverse(List)` – otočení seznamu

`void shuffle(List)` – zamíchání

`void sort(List)` – přirozené řazení

`void sort(List, Comparator)` – řazení s pomocí porovnávače

• ...

# Třída java.util Collections - příklad

```
List l = new ArrayList();  
Collections.addAll(l, 1, 2, 3, 4, 5, 2, 3, 4); //přidej  
    prvky  
System.out.println(l);  
Collections.shuffle(l); //zamichej  
System.out.println(l);  
Collections.replaceAll(l, 3, 232); // nahrad  
System.out.println(l);  
Collections.sort(l); //serad  
System.out.println(l);  
System.out.println(  
    "Číslo 4 se opakuje " + Collections.frequency(l, 4));
```

```
[1, 2, 3, 4, 5, 2, 3, 4]  
[2, 1, 2, 3, 5, 3, 4, 4]  
[2, 1, 2, 232, 5, 232, 4, 4]  
[1, 2, 2, 4, 4, 5, 232, 232]  
Číslo 4 se opakuje 2
```

# Kolekce a řazení

## Přirozené řazení (podle jednoho kritería)

- prvky kolekce musí implementovat rozhraní `Comparable`
  - třídy `String`, `Integer`, `Double`, ... jej implementují
- jediná metoda `int compareTo(T e)`
  - `this > e ... 1`, `this < e ... -1`, `this = e ... 0`
- Pro objekty musíme napsat, jediný atribut pro porovnání

## Absolutní řazení (možné vybrat si kritérium pro porovnání)

- prvky kolekce musí implementovat rozhraní `Comparator`
- metody
  - `int compare(T1 e1, T2 e2)`
  - `boolean equals(T e)`
- Implementuje se mimo kolekci

Pro řazení kolekcí používáme metody:

- `void sort(List)` – přirozené řazení
- `void sort(List, Comparator)` – absolutní řazení

# Přirozené řazení – řazená třída

```
class Zbozi implements Comparable<Zbozi> {  
    int cena; String nazev;  
    public Zbozi(int cena, String nazev) {  
        this.cena = cena; this.nazev = nazev;  
    }  
    public int compareTo(Zbozi o) {  
        return cena - o.cena;  
    }  
    @Override  
    public String toString() {  
        return nazev + " " + cena + ",- Kč";  
    }  
}
```

Budu se umět porovnat s jiným objektem typu Zbozi. Parametrem metody compareTo bude Zbozi, nikoli Object

# Přirozené řazení – main

```
public class TestPrirozeneRazeni {  
    public static void main(String[] args) {  
        List l = new ArrayList();  
        Collections.addAll(l, new Zbozi(120, "obed"),  
            new Zbozi(2, "rohlik"), new Zbozi(10, "nanuk"));  
        Collections.sort(l);  
        System.out.println(l);  
    }  
}
```

- metoda sort používá vylepšený quickSort
- metoda sort volá compareTo třídy Zbozi
- výsledek: [rohlik 2,- Kc, nanuk 10,- Kc, obed 120,- Kc]

# Absolutní řazení s komparátorem

Komparátor je třída implementující rozhraní `Comparator`

- metoda `compare(Object o1, Object o2)`
- metoda `boolean equals(T t)`
- každá třída může mít jeden způsob přirozeného řazení a kolik chceme komparátorů

```
class PodleNazvu implements Comparator<Zbozi2>{  
    public int compare(Zbozi2 o1, Zbozi2 o2) {  
        return o1.nazev.compareTo(o2.nazev);  
    }  
}
```

```
class PodleCeny implements Comparator<Zbozi2>{  
    public int compare(Zbozi2 o1, Zbozi2 o2) {  
        return o1.cena - o2.cena;  
    }  
}
```

```
class Zbozi2{  
    int cena;  
    String nazev;  
    ... //stejně jako u Zbozi  
}
```

Porovnávám Zbozi2

Název je typu String a ty se umí porovnávat – deleguji práci na ně.



# Řazení s komparátorem - main

```
public class TestRazeniSKomparatorem{  
    public static void main(String[] args) {  
        List l = new ArrayList();  
        Collections.addAll(l, new Zbozi2(120, "obed"),  
            new Zbozi2(2, "rohlik"), new Zbozi2(10, "nanuk"));  
        Collections.sort(l, new PodleNazvu());  
        System.out.println(l);  
        Collections.sort(l, new PodleCeny());  
        System.out.println(l);  
    }  
}
```



Zde specifikuji komparator.

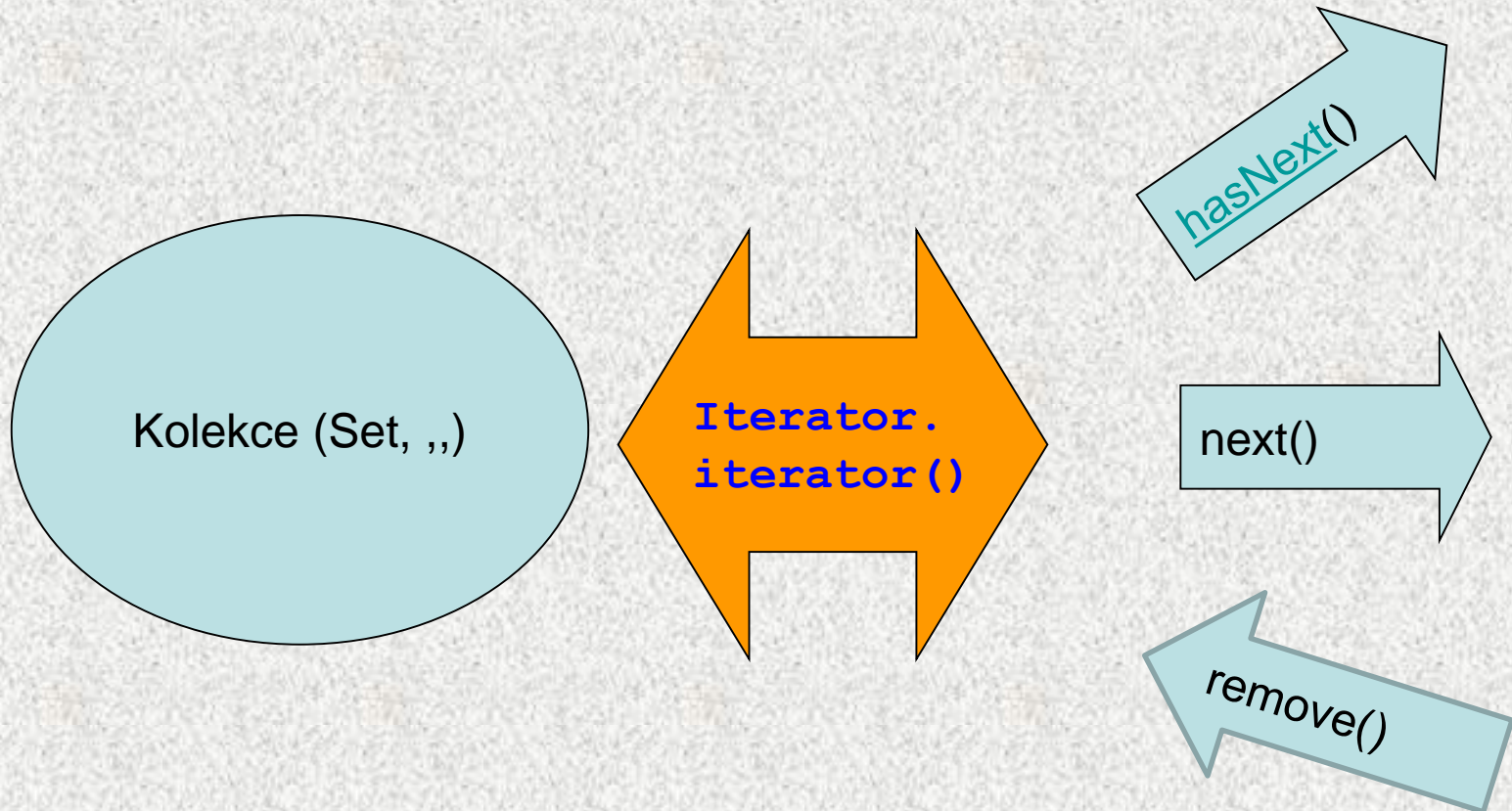
Výsledek:

- [nanuk 10,- Kc, obed 120,- Kc, rohlik 2,- Kc]
- [rohlik 2,- Kc, nanuk 10,- Kc, obed 120,- Kc]

# Iterátory

- Třídy, které umožní postupný přístup ke všem prvkům kolekce (i k množinám, které to v principu nepřipouští)
  - **Iterátory neumožní indexaci, ale postupné procházení**
    - Použijeme např. **po** převodu kolekce na množinu
      - `Collection<Typ> c = new ArrayList<Typ>();`
      - `Collection<Typ> c = new HashSet<Typ>();`
- Například množiny (`Set`) nemají možnost iterace, tj. procházení všemi prvky (metoda typu *E* `get (int index)`)
  - Řešení – iterátor, zobecnění indexu
- **Dva typy iterátoru – „zobecnění indexu“**
  - jednoduchý iterátor `for-each`
  - Objekt třídy `Iterator`
    - každá třída kolekcí vrací `Iterator iterator()`
    - je to „index“ do kolekce, zobecnění indexu

# Iterátory



# Iterátor for-each

- Iterátor pro procházení kolekcí, vrací jednotlivé objekty

```
for (TypObjektuVKolekci refPromenna: kolekce) {  
    <zpracování objektu s refPromenna>  
}
```

```
public class TypickyForEach {  
    public static void main(String[] args) {  
        ArrayList<Integer> ar = new ArrayList<Integer>();  
        ar.add(new Integer(1));  
        ar.add(new Integer(2));  
        for (Integer i : ar) {  
            System.out.print(i.intValue() + ", ");  
        }  
    }  
}
```

1, 2,

```
int[] pole = {5, 6, 7, 8, 9};
```

```
for (int hodnota : pole) {  
    System.out.print(hodnota + ", ");  
}
```

# Rozhraní `Iterator<E>` a `ListIterator<E>`

- Metoda `iterator()` vrací objekt, který je schopen „iterovat“, procházet kolekci

- `Iterator` požaduje metody pro objekt, kterým lze probírat skupinu:

`boolean hasNext()` - testuje zda jsou ještě další prvky

`E next()` - podá další prvek

`void remove()` - odstraní prvek ze sbírky, odkazovaný `next()`

- `ListIterator` extends `Iterator` s dodatečnými požadavky pro sekvenční přístup, přidávání, změny a indexování prvků metodami:

`boolean hasNext()` - testuje zda jsou ještě další prvky

`E previous()` - dodá předchozí prvek

`int nextIndex()` - vrátí index dalšího prvku

`int previousIndex()` - vrátí index předchozího prvku

`void set(E e)` - změní prvek

`void add(E e)` - přidá prvek před aktuální prvek

# Příklad na Iterator

```
class Zbozi implements Comparable <Zbozi> {  
    private int cena;  
    Zbozi(int cena) { this.cena = cena; }  
    public String toString() { return "" + cena; }  
    public void tisk() { System.out.print(cena + ", "); }  
    public int compareTo (Zbozi o){  
        return cena - o.cena;}  
  
}
```

# Příklad na Iterator

```
public class IteratorZakladniPouziti {
    public static void main(String[] args) {
        ArrayList<ZboziX> kosHrusek = new ArrayList<ZboziX>();
        for (int i = 0; i < 10; i++) {
            kosHrusek.add(new ZboziX(i + 20));
        }
        for (Iterator<ZboziX> it = kosHrusek.iterator();
            it.hasNext(); ) {
            System.out.print(it.next() + ", ");
        }
        System.out.println();
        Iterator<ZboziX> it = kosHrusek.iterator();
        while (it.hasNext()) {
            it.next().tisk();
        }
        System.out.println();
    }
}
```

```
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
```

# Příklad na ListIterator

```
public class TestListIterator {
    public static void main(String[] argv) {
        String[] tmp = {"1", "2", "3", "4", "5"};
        List<String>list = new
            ArrayList<String>(Arrays.asList(tmp));
        System.out.println("Seznam:          " + list);
        System.out.print("Seznam pozpatku: [");
        // vrací objekt, který je schopen iterovat, procházet list
        for(ListIterator<String> it = list.listIterator(list.size());
        it.hasPrevious(); )
        {
            System.out.print(it.previous() + ", ");
        }
        System.out.println("]");
    }
}
```



# Hešování

- Technika, která v ideálním případě zaručí vložení, odebrání, zjištění přítomnosti prvku **v konstantním** čase
- Hešovací funkce
  - zajišťuje mapování prvků kolekce na int, který slouží k výpočtu indexu do kolekce
  - ideálně pro dva různé prvky vytvoří metoda `hashCode()` dvě různé hodnoty
  - mnohdy to nejde, např. `String`, počet různých řetězců výrazně převyšuje int:

```
System.out.println("AT " + new String("AT").hashCode());
```

```
System.out.println("B5 " + new String("B5").hashCode());
```

```
AT 2099
```

```
B5 2099
```

# Princip hešování

alma

alfa

anna

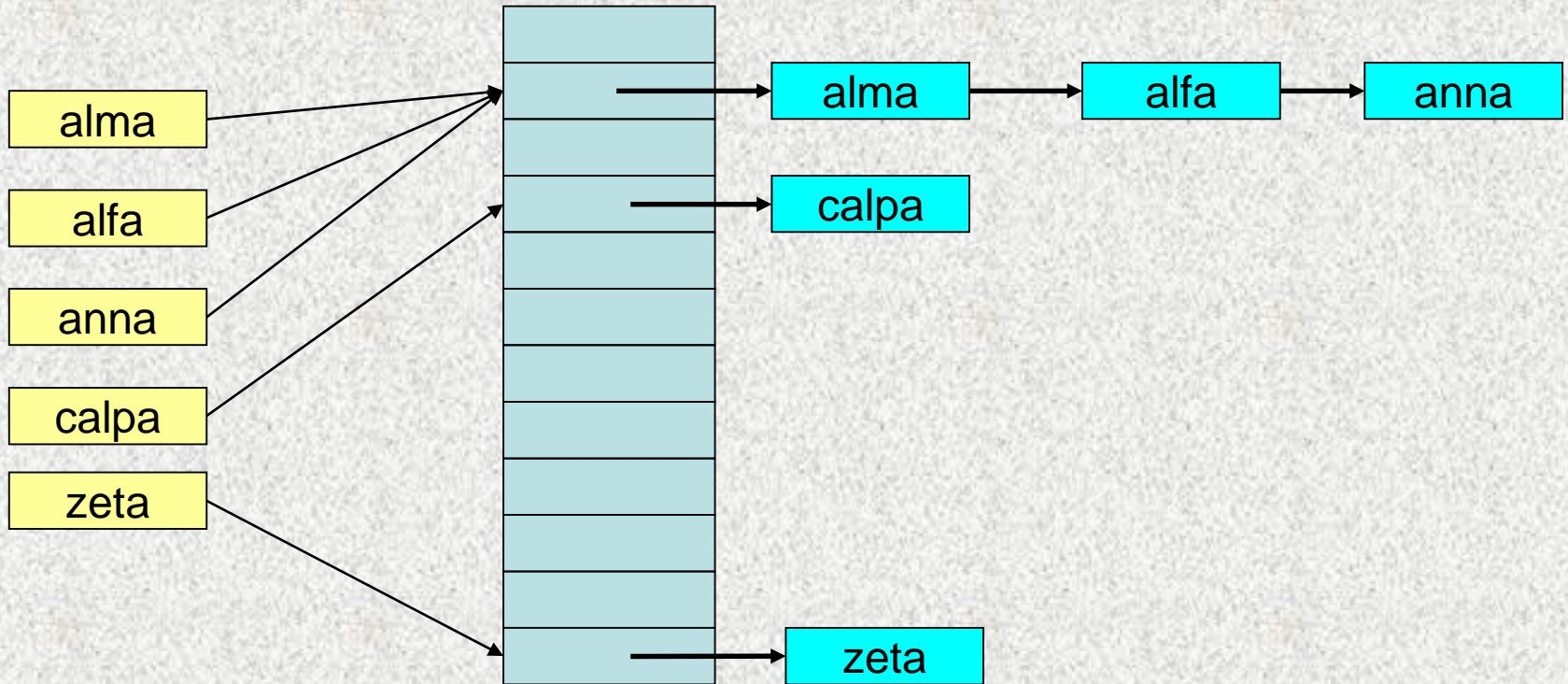
calpa

zeta

alma
calpa
anna
zeta
alfa

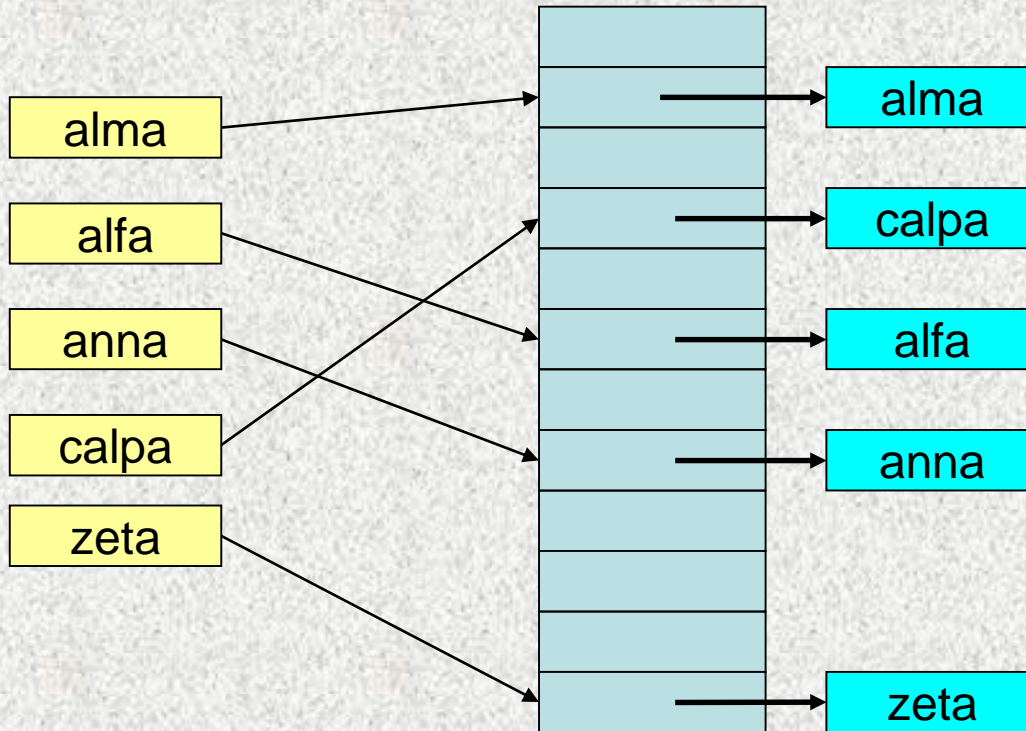
Nalezení alfa trvá 5 přístupů, bez hešování

# Princip hešování



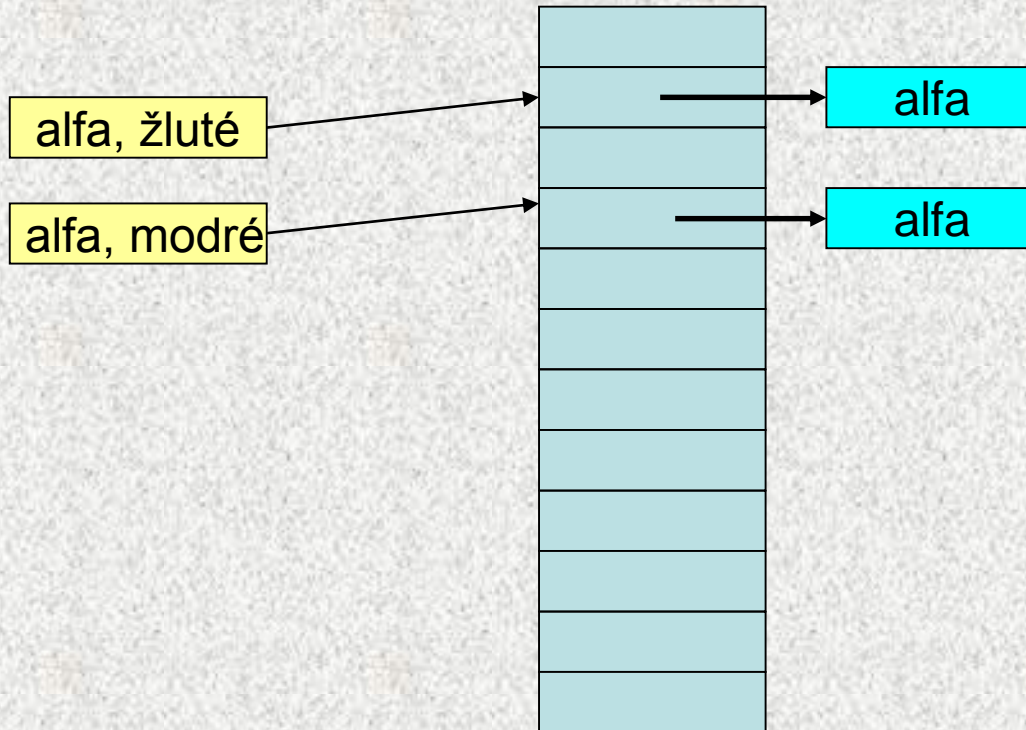
alma, alfa, anna – hešovací kód špatně navržen

# Princip hešování



alma, alfa, anna – různý hešovací kód, dobře navržen

# Princip hešování



<alfa,žluté>, <alfa,modré> – špatně: různý hešovací kód pro ekvivalentní položky „alfa“

# Metody `hashCode` a `equals`

- Obě metody spolu úzce souvisí
  - nejprve se hledá podle `hashCode`, potom se rozlišuje podle `equals`!
  - je-li `hashCode` špatně navržen, ukládání a hledání se prodlužuje, případně může dojít k nenalezení
- Pokud zastíníme `equals` **musíme** zastínit `hashCode`
- Správně vytvořený objekt musí splňovat:
  1. Pokud jsou dva objekty stejné (podle `equals`) **musí** metoda `hashCode` vracet stejnou hodnotu
  2. Naopak to neplatí: dva **různé objekty** (podle `equals`), které vracejí stejný `hashCode`, ale výrazně to může zhoršit výkonnost aplikace
  3. `hashCode` by měl být konstantní během existence objektu v kolekci,
    - při změně vlastností vedoucích ke změně `hashCode` je nutné prvek z kolekce vyjmout, změnit a teprve poté vrátit zpět – *pokud se stěhujete, musíte změnit adresu pro doručování.*

# equals - implementace relace ekvivalence

- je **reflexivní**: `x.equals(x)` musí vrátit true, pro každé x (mimo null)
- je **symetrická**: pro jakékoli x a y musí `x.equals(y)` vrátit true právě tehdy a jen tehdy, když `y.equals(x)` vrátí true.
- je **tranzitivní**: pro jakékoli x, y a z musí platit, že když `x.equals(y)` vrací true a `y.equals(z)` vrací true, pak `x.equals(z)` musí vrátit true.
- je **konzistentní**: pro jakékoli odkazové hodnoty x a y musí platit, že buď `x.equals(y)` vrací true nebo stále vrací false za předpokladu, že nedojde ke změně žádných informací použitých v porovnáních equals daného objektu.
- pro všechny odkazové hodnoty x, které nejsou `null`, musí `x.equals(null)` vrátit false.

# equals - příklady, klady, zápory

```
public class Zbozi {  
    String nazev;  
    int cena;  
    @Override  
    public boolean equals(Object obj) {  
        return nazev.equals(((Zbozi) obj).nazev);  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if(obj == null) return false;  
        return nazev.equals(((Zbozi) obj).nazev);  
    }  
}
```

Neumí se porovnat s null ! způsobí  
NullPointerException

```
public class Zbozi {  
    String nazev;  
    int cena;  
}
```



# equals - příklady, klady, zápory

@Override

```
public boolean equals(Object obj) {  
    if(obj == null) return false;  
    if(!(obj instanceof Zbozi)) return false;  
    return nazev.equals(((Zbozi) obj).nazev);  
}
```

reflexivita

```
@Override public boolean equals(Object obj) {  
    if (obj == null) {return false;}  
    if (getClass() != obj.getClass()) {return false;}  
    final Zbozi other = (Zbozi) obj;  
    if ((this.nazev == null) ? (other.nazev != null) :  
        !this.nazev.equals(other.nazev)) {  
        return false; }  
    if (this.cena != other.cena) {return false;}  
    return true;}  
  
public class Zbozi {  
    String nazev;  
    int cena; }  
}
```

# Návod na vytvoření dobré `hashCode`

Zamixuje důležité vlastnosti objektu (zejména s ohledem na `equals`)

1. Uložte magickou prvočíselnou konstantní nenulovou hodnotu, řekněme 17, v proměnné typu `int` nazvané `vysledek`.
2. Pro každý významný atribut `f` ve svém objektu (to znamená každý atribut zvažovaný metodou `equals`), vypočtěte hešovací kód `pom` typu `int` pro daný atribut:
  1. `boolean`, `pom = f ? 0 : 1`
  2. `byte`, `char`, `short` nebo `int`, `pom = (int)f`
  3. `long`, `pom = (int)(f ^ (f >>> 32))`
  4. `float`, `pom = Float.floatToIntBits(f)`
  5. `double`, vypočtěte `Double.doubleToLongBits(f)` a pak viz 2.3.
  6. odkaz na objekt: `pom = (f == null) ? 0 : f.hashCode()`
  7. pole, počítejte `pom` pro každý prvek
3. `vysledek = vysledek * 37 + pom;`
4. pokračujte dalším atributem a bodem 2
5. vraťte výsledek, pokud již nejsou žádné další důležité atributy

# hashCode - příklady, klady, zápory

```
@Override
```

```
public int hashCode() {  
    return 73;  
}
```

- formálně správně
- všechny prvky budou v seznamu pod jedním indexem, výhody hešování absolutně potlačeny - všechny operace s kolekcí budou lineární

```
public class Zbozi {  
    String nazev;  
    int cena;  
    public boolean equals(Object obj) {  
        if(obj == null) return false;  
        if(!(obj instanceof Zbozi))  
            return false;  
        return nazev.equals(((Zbozi) obj).nazev);  
    }  
}
```

# hashCode - příklady, klady, zápory

@Override

```
public int hashCode() {  
    int hash = 7;  
    hash = 47 * hash + this.nazev.hashCode();  
    return hash;  
}
```

pokud název bude null, dojde k chybě

```
public class Zbozi {  
    String nazev;  
    int cena;  
    public boolean equals(Object obj) {  
        if(obj == null) return false;  
        if(!(obj instanceof Zbozi))  
            return false;  
        return nazev.equals(((Zbozi) obj).nazev);  
    }  
}
```

# hashCode - příklady, klady, zápory

@Override

```
public int hashCode() {  
    int hash = 7;  
    hash= 47 * hash + (this.nazev != null?  
                        this.nazev.hashCode() : 0) ;  
    hash = 47 * hash + this.cena;  
    return hash;  
}
```

započítává i cenu,  
která není v equals, špatně

```
public class Zbozi {  
    String nazev;  
    int cena;  
    public boolean equals(Object obj) {  
        if(obj == null) return false;  
        if(!(obj instanceof Zbozi))  
            return false;  
        return nazev.equals(((Zbozi)obj).nazev);  
    }  
}
```

# hashCode - příklady, klady, zápory

@Override

```
public int hashCode() {  
    int hash = 7;  
    hash = 47 * hash + (this.nazev != null ?  
                        this.nazev.hashCode() : 0);  
    return hash;  
}
```

řešení

```
public class Zbozi {  
    String nazev;  
    int cena;  
    public boolean equals(Object obj) {  
        if(obj == null) return false;  
        if(!(obj instanceof Zbozi))  
            return false;  
        return nazev.equals(((Zbozi)obj).nazev);  
    }  
}
```

# Použití hešování

Zjistěte zboží, které se dnes prodalo, nezajímá nás cena (registrovaní zákazníci mohou mít různé slevy)

```
public class Zbozi {
    String nazev; int cena;
    public Zbozi(String nazev, int cena) {
        this.nazev = nazev; this.cena = cena; }
    @Override
    public String toString() {
        return nazev + " " + cena + ",- Kč"; }
    @Override
    public boolean equals(Object obj) {
        if(obj == null) return false;
        if(!(obj instanceof Zbozi)) return false;
        return nazev.equals(((Zbozi)obj).nazev); }
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 47 * hash + (this.nazev != null ? this.nazev.hashCode() : 0);
        return hash;
    }
}
```

# Použití hešování

```
public class TestHesovani {  
    public static void main(String[] args) {  
        Set seznam = new HashSet();  
        seznam.add(new Zbozi("Ponozky", 15));  
        seznam.add(new Zbozi("Triko", 80));  
        seznam.add(new Zbozi("Tanga", 55));  
        seznam.add(new Zbozi("Ponozky", 25));  
        seznam.add(new Zbozi("Stringy", 105));  
        seznam.add(new Zbozi("Ponozky", 26));  
        System.out.println(seznam);  
    }  
}
```

[Ponozky 15,- Kč, Triko 80,- Kč, Tanga 55,- Kč, Stringy 105,- Kč]



# Užité techniky

interface	<i>HT</i>	<i>RA</i>	<i>BT</i>	<i>LL</i>	<i>HT+LL</i>
List	-	ArrayList	-	LinkedList	-
Set	HashSet	-	-	-	LinkedHashSet
SortedSet	-	-	TreeSet	-	-
Map	HashMap	-	-	-	LinkedHashMap
SortedMap	-	-	TreeMap	-	-

- *HT - hashTable - rozmítaná tabulka,*
  - *RA - resizable array - pole s proměnnou velikostí,*
  - *BT - balanced tree - vyvážený strom,*
  - *LL - linked list - spojový seznam.*
- 
- *zvolený interface je určen účelem kolekce*
  - *implementace ovlivní rychlost práce s kolekcí, nikoli chování*



# java.util.Arrays – Comparable

```
// řazení přirozené - podle jednoho kritéria
// <Osoba> - genericita
public class Osoba implements Comparable<Osoba> {
    int vyska;
    double vaha;
    String popis;

    Osoba(int vyska, double vaha, String popis) {
        this.vyska = vyska;
        this.vaha = vaha;
        this.popis = popis;
    }

    public int compareTo(Osoba os) {
        return this.vyska - os.vyska;
    }
}
```

# java.util.Arrays – Comparable

```
static Osoba[] poleOsob;
```

```
public static void main(String[] args) {  
    poleOsob = new Osoba[4];  
    poleOsob[0] = new Osoba(186, 82.5, "muz");  
    poleOsob[1] = new Osoba(172, 63.0, "zena");  
    poleOsob[2] = new Osoba(105, 26.1, "dite");  
    poleOsob[3] = new Osoba(116, 80.5, "obezni trpaslik");  
}
```

[0] vy = 105, va = 26.1, dite

[1] vy = 116, va = 80.5, obezni trpaslik

[2] vy = 172, va = 63.0, zena

[3] vy = 186, va = 82.5, muz

# java.util.Arrays – Comparator

```
//Řazení absolutní podle vlastních kritérií pro  
objekty.  
//pro každé kriterium třída implement Comparator  
class PodleAbsHodnotyComparator implements  
                                Comparator<Integer>{  
    //genericita viz dále, seradi podle absolutnich hodnot  
    public int compare(Integer o1, Integer o2) {  
        return Math.abs(o1) - Math.abs(o2);  
    }  
}  
public class TestPole2 {  
    public static void main(String[] args) {  
        Integer[] pole = {3,57,23,-100,1,7,-4,-8,-5,-24};  
        System.out.println(Arrays.toString(pole));  
        Arrays.sort(pole,new PodleAbsHodnotyComparator());  
        System.out.println(Arrays.toString(pole));  
    }  
}
```

autounboxing Integer->int

autoboxing int->Integer

```
[3, 57, 23, -100, 1, 7, -4, -8, -5, -24]  
[1, 3, -4, -5, 7, -8, 23, -24, 57, -100]
```