

Seminář Java

Kontejnery

Radek Kočí

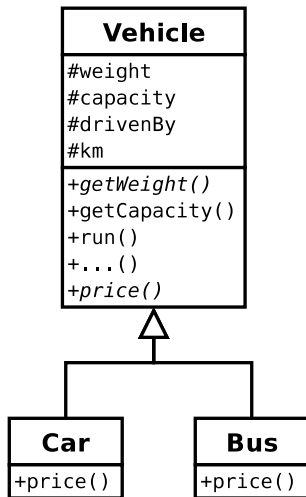
Fakulta informačních technologií VUT

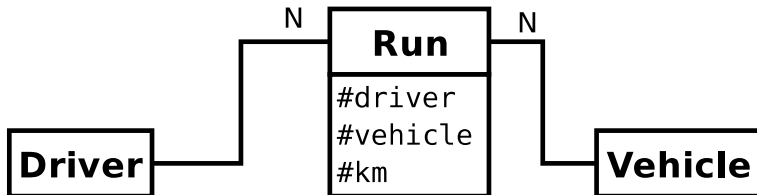
Březen 2010

- Kontejnery
- For-each loop
- Autoboxing

```
public class Car {  
    ...  
    public int run(int km) { ... }  
}
```

```
public class Driver {  
    protected String name;  
    protected Car driverCar;  
    public void driveCar(Car c) {  
        driverCar = c;  
        driverCar.drivenBy(this);  
    }  
}
```





Kontejnery (containers) v Javě

- slouží k ukládání objektů (ne hodnot primitivních typů!)
- verze < 5.0
 - kontejnery jsou koncipovány jako beztypové
- verze => 5.0
 - kontejnery nesou typovou informaci o prvcích

Většinou se používají kontejnery hotové, vestavěné (součást Java Core API):

- vestavěné kontejnerové třídy jsou definovány v balíku `java.util`
- je možné vytvořit si vlastní implementace, obvykle ale zachovávající/implementující "standardní" rozhraní

K čemu slouží?

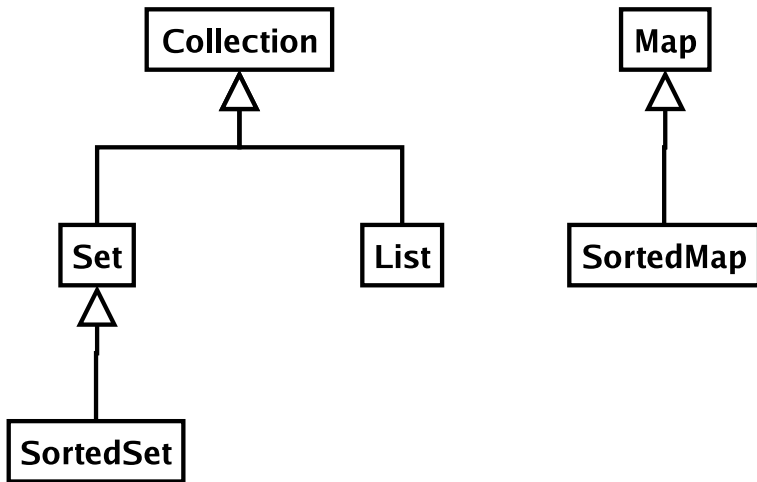
- jsou dynamickými alternativami k poli a mají daleko širší použití
- k uchování proměnného počtu objektů
- počet prvků se v průběhu existence kontejneru může měnit
- oproti polím nabízejí časově efektivnější algoritmy přístupu k prvkům

Základní kategorie kontejnerů

- seznam (`List`) – lineární struktura
- množina (`Set`) – struktura bez uspořádání, rychlé dotazování na přítomnost prvku
- asociativní pole, mapa (`Map`) – struktura uchovávající dvojice `klíč→hodnota`, rychlý přístup přes klíč

Kontejnery – rozhraní, nepovinné metody

- Funkcionalita vestavěných kontejnerů je obvykle předepsána výhradně rozhráním, které implementují.
- Rozhraní však připouštějí, že některé metody jsou nepovinné, třídy je nemusí implementovat (*optional operations*)!
- V praxi se totiž někdy nehodí implementovat jak čtecí, tak i zápisové operace – některé kontejnery jsou "read-only"



Kontejnery – implementace rozhraní

	<i>hash table</i>	<i>resizable array</i>	<i>balanced tree</i>	<i>linked list</i>
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

SortedSet
SortedMap

Uvedené kontejnery

- implementují všechny *optional* operace
- povolují `null` elementy (klíče, hodnoty)
- jsou nesynchronizované

- Moderní kontejnery jsou nesynchronizované, nepřipouštějí souběžný přístup z více vláken.
- Standardní, nesynchronizovaný, kontejner lze však "zabalit" synchronizovanou obálkou.
- Při práci s kontejnery může vzniknout řada výjimek, např. `IllegalStateException` apod.
- Většina má charakter výjimek běhových, není povinností je odchyťovat – pokud věříme, že nevzniknou.

Kolekce

- jsou kontejnery implementující rozhraní `Collection` (viz *API doc k rozhr. Collection*)
- Rozhraní kolekce popisuje velmi obecný kontejner, disponující operacemi: přidávání, rušení prvku, získání iterátoru, zjišťování prázdnoty atd.
- Mezi kolekce patří mimo `Map` všechny ostatní vestavěné kontejnery – `List`, `Set`
- Prvky kolekce nemusí mít svou pozici danou indexem – viz např. `Set`

```
public interface List extends Collection {  
    ...  
    public boolean add(Object o);  
    public Object get(int index);  
}
```

- při vybírání elementu z kolekce musíme *přetypovat*
- není hlídáno kompilátorem
- pouze dynamická typová kontrola
- ⇒ šance na vygenerování run-time výjimky

Generics (vlastnost Java 5.0)

```
public interface List<E> extends Collection<E>
{
    ...
    public boolean add(E o);
    public E get(int index);
}
```

-
- umožňuje komunikaci s kompilátorem
 - statická kontrola typů při manipulaci s kontejnery
 - bez přetypování!

Iterátory jsou prostředkem, jak "chodit" po prvcích kolekce buďto

- v neurčeném pořadí nebo
- v uspořádání (u uspořádaných kolekcí)

Každý iterátor musí implementovat velmi jednoduché rozhraní

`Iterator` se třemi metodami:

- `boolean hasNext()`
- `Object next()` (resp. `E next()`)
- `void remove()`

Seznamy

- lineární struktury
- implementují rozhraní `List`
- prvky lze adresovat indexem (typu `int`)
- poskytují možnost získat dopředný i zpětný iterátor
- lze pracovat i s podseznamy

Standardní implementace

- `ArrayList`
- `LinkedList`

Seznamy – Příklad

```
public class Run {
    protected Driver driver;
    protected Car car;
    protected int km;
    ...
}

public class Car {
    protected List runs = new ArrayList();
    ...
    public int run(Driver d, int km) {
        runs.add(new Run(d, this, km));
    }
}
```

Seznamy – Příklad

```
public class Car {
    protected List runs = new ArrayList();
    ...
    public int run(Driver d, int km) {
        runs.add(new Run(d, this, km));
    }
    public iterate() {
        for (Iterator i = seznam.iterator();
            i.hasNext(); )
        {
            Run r = (Run)i.next();
            r.getDriver();
            ...
        }
    }
}
```

Seznamy – příklad (Generics 5.0)

Při překladu předchozího kódu:

Note: `ListDemo2.java` uses unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

```
javac -Xlint:unchecked ListDemo2.java
```

```
ListDemo2.java:9: warning: [unchecked]
unchecked call
    to add(E) as a member of the raw type
java.util.List
    seznam.add(new Run(d, this, km));
...

```

Seznamy – Příklad (Generics 5.0)

```
public class Car {
    protected List<Run> runs = new ArrayList<Run>();
    ...
    public int run(Driver d, int km) {
        runs.add(new Run(d, this, km));
    }
    public iterate() {
        for (Iterator<Run> i = seznam.iterator();
            i.hasNext(); )
        {
            Run r = i.next();
            r.getDriver();
            ...
        }
    }
}
```

Seznamy – Příklad (Generics 5.0)

```
public class Car {
    protected List<Run> runs = new ArrayList<Run>();
    ...
    public int run(Driver d, int km) {
        runs.add(new Run(d, this, km));
    }
    public iterate() {
        ListIterator<Run> it = seznam.listIterator(1);
        Run r;
        r = it.previous();
        r = it.next();
        ...
    }
}
```

The For-Each Loop (Java 5.0)

```
void cancelAll(Collection<TimerTask> c)
{
    for (Iterator<TimerTask> i = c.iterator();
         i.hasNext(); )
    {
        TimerTask t = i.next();
        t.cancel();
    }
}
```

```
for (TimerTask t : c )
{
    t.cancel();
}
```

The For-Each Loop (Java 5.0)

```
List suits, ranks = ...;
List sortedDeck = new ArrayList();

// BROKEN - throws NoSuchElementException!

for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext();
         sortedDeck.add(new Card(i.next(), j.next())));
```

```
// Fixed, though a bit ugly
for (Iterator i = suits.iterator(); i.hasNext(); )
{
    Suit suit = (Suit) i.next();

    for (Iterator j = ranks.iterator(); j.hasNext();
         sortedDeck.add(new Card(suit, j.next())));
}
```


The For-Each Loop (Java 5.0)

```
List suits = ...;  
List ranks = ...;  
List sortedDeck = new ArrayList();  
  
for (Suit suit : suits)  
    for (Rank rank : ranks)  
        sortedDeck.add(new Card(suit, rank));
```

Množiny

- struktury standardně bez uspořádání prvků (uspořádané viz dále)
- implementují rozhraní `Set` (což je rozšíření `Collection`)

Cílem množin je mít možnost rychle (se složitostí $O(\log(n))$) provádět atomické operace:

- vkládání prvku (`add`)
- odebrání prvku (`remove`)
- dotaz na přítomnost prvku (`contains`)
- lze testovat i relaci `je podmnožinou`

Standardní implementace

- `HashSet` – hašovací tabulka
- `TreeSet` – vyhledávací strom

```
Set<Driver> mnozina = new HashSet<Driver>();  
  
Driver d1 = new Driver("Pepa");  
Driver d2 = new Driver("Franta");  
  
mnozina.add(d1);  
mnozina.add(d2);  
  
System.out.println("Je v mnozine Franta?" +  
    mnozina.contains(d2));
```

Množiny (equals a hashCode)

```
Set<Driver> mnozina = new HashSet<Driver>();

Driver d1 = new Driver("Pepa");
Driver d2 = new Driver("Franta");

mnozina.add(d1);
mnozina.add(d2);

Driver d3 = new Driver("Franta");

System.out.println("Je v mnozine Franta?" +
    mnozina.contains(d3));
```

Množiny (equals a hashCode)

```
class Driver {
    String name;
    ...
    public boolean equals(Object o) {
        if (o instanceof Driver) {
            Driver d = (Driver) o;
            return name.equals(d.name);
        }
        else
            throw new IllegalArgumentException("...");
    }

    public int hashCode() {
        return name.hashCode();
    }
}
```

Uspořádané množiny

- Implementují rozhraní `SortedSet` (viz *API doc k rozhraní `SortedSet`*)
- Jednotlivé prvky lze tedy iterátorem procházet v přesně definovaném pořadí (uspořádání) podle hodnot prvků.

Standardní implementace

- `TreeSet` – vyhledávací *Red-Black Trees*

Uspořádané množiny – příklad

```
SortedSet<Driver> mnozina = new TreeSet<Driver>();

Driver d1 = new Driver("Pepa");
Driver d2 = new Driver("Franta");

mnozina.add(d2);
mnozina.add(d1);

for (Driver d : mnozina)
{
    d.getName();
}
```

Run-time vyjimka `java.lang.ClassCastException`

Uspořádání je dáno:

- standardním chováním metody `compareTo` vkládaných objektů – pokud implementují rozhraní `Comparable`
- nebo je možné uspořádání definovat pomocí tzv. komparátoru (objektu impl. rozhraní `Comparator`) při vytvoření množiny.

Uspořádané množiny – komparátor (příklad)

Implementace operace `compareTo`

```
public class Driver implements Comparable {
    String name;
    ...
    public int compareTo(Object o) {
        if (o instanceof Driver) {
            Driver c = (Driver) o;
            return name.compareTo(c.name);
        }
        else
            throw new IllegalArgumentException("...");
    }
}
```

Mapy (asociativní pole, nepřesně také hašovací tabulky nebo haše) fungují na stejných principech a požadavcích jako Set:

- avšak ukládají dvojice (`klíč→hodnota`) a umožňují rychlé vyhledání dvojice podle hodnoty klíče,
- základními metodami jsou: dotazy na přítomnost klíče v mapě (`containsKey`),
- výběr hodnoty odpovídající zadanému klíči (`get`),
- možnost získat zvlášť množiny klíčů, hodnot nebo dvojic (`klíč→hodnota`)

Mapy mají:

- podobné implementace jako množiny (tj. hašovací tabulky nebo stromy)
- logaritmickou složitost základních operací
 - `put`
 - `remove`
 - `containsKey`

Standardní implementace

- `HashMap`
- `TreeMap`

```
Map<String, Driver> mapa =
    new HashMap<String, Driver>();

Driver d1 = new Driver("Pepa");
Driver d2 = new Driver("Franta");

mapa.put(d1.getName(), d1);
mapa.put(d2.getName(), d2);

Driver d = mapa.get("Pepa");

for (String name : mapa.keySet())
{
    mapa.get(name);
}
```

Uspořádané mapy

- Implementují rozhraní `SortedMap`
- Dvojice (`klíč`→`hodnota`) jsou v nich uspořádané podle hodnot klíče.
- Uspořádání lze ovlivnit naprosto stejným postupem jako u uspořádané množiny.

Standardní implementace

- `TreeMap` – implementace Red-Black Trees

Uspořádané mapy – příklad

```
SortedMap<String, Driver> mapa =  
    new TreeMap<String, Driver>();
```

```
Driver d1 = new Driver("Pepa");  
Driver d2 = new Driver("Franta");
```

```
mapa.put(d1.getName(), d1);  
mapa.put(d2.getName(), d2);
```

```
for (String name : mapa.keySet())  
{  
    Driver d = mapa.get(name);  
}
```

Uspořádané mapy – příklad

Exception in thread "main"

java.lang.ClassCastException: ...

at java.util.TreeMap.compare (TreeMap.java:1093)

at java.util.TreeMap.put (TreeMap.java:465)

at seminar6.SortedMapDemo.main (SortedMapDemo.java

Uspořádané mapy – příklad s komparátorem

```
SortedMap<Driver, Car> mapa =  
    new TreeMap<Driver, Car>(new DComp());  
  
Driver d1 = new Driver("Pepa");  
Car c1 = new Car();  
Driver d2 = new Driver("Franta");  
Car c2 = new Car();  
  
mapa.put(d1, c1);  
mapa.put(d2, c2);  
  
for (Driver d : mapa.keySet())  
{  
    Car c = mapa.get(d);  
    // d ridi c  
}
```


Uspořádané mapy – příklad s komparátorem

```
class DComp implements Comparator
{
    public int compare(Object o1, Object o2) {
        // porovnava jen podle jmena
        if (o1 instanceof Driver &&
            o2 instanceof Driver)
        {
            Driver c1 = (Driver)o1;
            Driver c2 = (Driver)o2;
            return c1.name.compareTo(c2.name);
        } else
            throw new IllegalArgumentException("..");
    }
}
```

Seznamy

- na bázi pole (`ArrayList`) – rychlý přímý přístup (přes index)
- na bázi lineárního zřetězeného seznamu (`LinkedList`) – rychlý sekvenční přístup (přes iterátor)
- téměř vždy se používá `ArrayList` – stejně rychlý a paměťově efektivnější

Množiny a mapy

- na bázi hašovacích tabulek ([HashMap](#), [HashSet](#)) – rychlejší, ale neuspořádané (lze získat iterátor procházející klíče uspořádaně)
- na bázi vyhledávacích stromů ([TreeMap](#), [TreeSet](#)) – pomalejší, ale uspořádané
- spojení výhod obou – [LinkedHashSet](#), [LinkedHashMap](#) (novinka v Javě 2, v1.4)

Existují tyto starší typy kontejnerů (\Rightarrow náhrada)

- `Hashtable` \Rightarrow `HashMap`, `HashSet` (podle účelu)
- `Vector` \Rightarrow `List`
- `Stack` \Rightarrow `List`

Roli iterátoru plnil dříve výčet (enumeration) se dvěma metodami:

- `boolean hasMoreElements()`
- `Object nextElement()`

Statické metody třídy `Collections`

- `XXX unmodifiableXXX (XXX c);`
- `Set unmodifiableSet (Set c);`
- `Map unmodifiableMap (Map c);`
- ...

Nemodifikovatelné kolekce

```
public class UnmodifiableDemo {
    public static void main(String[] args) {
        List seznam = new ArrayList();
        seznam.add(new Integer(20));

        seznam = Collections.unmodifiableList(seznam);

        System.out.println(seznam);

        seznam.add(new Integer(20));
    }
}
```

Vyjimka `UnsupportedOperationException`

Nemodifikovatelné kolekce

```
// zjednodusena verze
public class Collections {
    private Collections {}

    public static Collection
        unmodifiableCollection(Collection c) {
        return new UnmodifiableCollection(c);
    }

    static class UnmodifiableCollection
        implements Collection, Serializable {
        public int size() { return c.size(); }
        public boolean add(E e){
            throw new UnsupportedOperationException();
        }
    }
}
```

Statické metody třídy `Collections`

- `XXX synchronizedXXX (XXX c);`
- `Collection synchronizedSet (Collection c);`
- `Map synchronizedMap (Map c);`
- ...

Příště ...

The For-Each Loop (Java 5.0)

Lze použít i pro pole ...

```
// Returns the sum of the elements of a
int sum(int[] a) {
    int result = 0;
    for (int i : a)
        result += i;
    return result;
}
```

Objektová verze primitivních datových typů

- int → Integer
- long → Long
- short → Short
- byte → Byte
- char → Character
- float → Float
- double → Double
- boolean → Boolean
- void → Void

Double.MAX_VALUE

float f = Float.parseFloat(řetězec)

Autoboxing (Java 5.0)

```
// Prints a frequency table of the words
// on the command line
public class Frequency {
    public static void main(String[] args) {
        Map<String, Integer> m = new
            HashMap<String, Integer>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ?
                new Integer(1) :
                new Integer(freq.intValue() + 1) ));
        }
        System.out.println(m);
    }
}
```

Autoboxing (Java 5.0)

```
public class Frequency {
    public static void main(String[] args) {
        Map<String, Integer> m = new
            HashMap<String, Integer>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ?
                1 : freq+1));
        }
        System.out.println(m);
    }
}
```

Autoboxing (Java 5.0)

```
java Frequency if it is to be it is up to me to do  
{be=1, do=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Podrobné seznámení s kontejnery

<http://java.sun.com/docs/books/tutorial/collections/>

Java SE 5.0

<http://java.sun.com/j2se/1.5.0/docs/>

Java SE 6

<http://java.sun.com/javase/6/docs/>