

# ÚVOD DO DATOVÝCH STRUKTUR



**Kurz:**      **Datové struktury a algoritmy**

---

**Lektor:**    Doc. Ing. Radim Burget, Ph.D.

**Autor:**     Doc. Ing. Radim Burget, Ph.D.



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

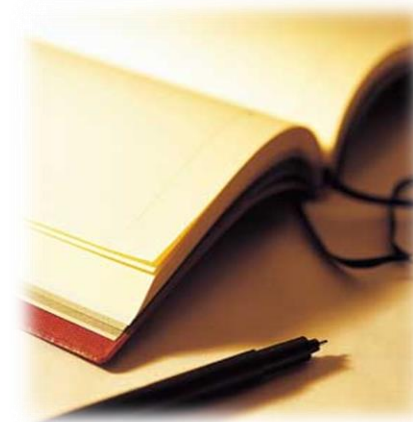
Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098  
Evropského sociálního fondu a státním rozpočtem České republiky.

# Cíl přednášky

• Představení základních konceptů pro reprezentaci informace:

1. Souvislost s přednáškou OON
2. Existence několika asociací a mnoha variant, proč?
3. Co jsou tzv. datové struktury a základní členění
4. Úvod do datových struktur
  - Atribut
  - Pole (neměnné délky)
  - Lineární seznamy
  - Hash tabulky
  - Mapování  $\text{Map}<\text{TKey}, \text{TValue}>$
  - Množiny

I když je nutné znát, jak pracují uvnitř, zde se budeme věnovat jen povrchnímu popisu



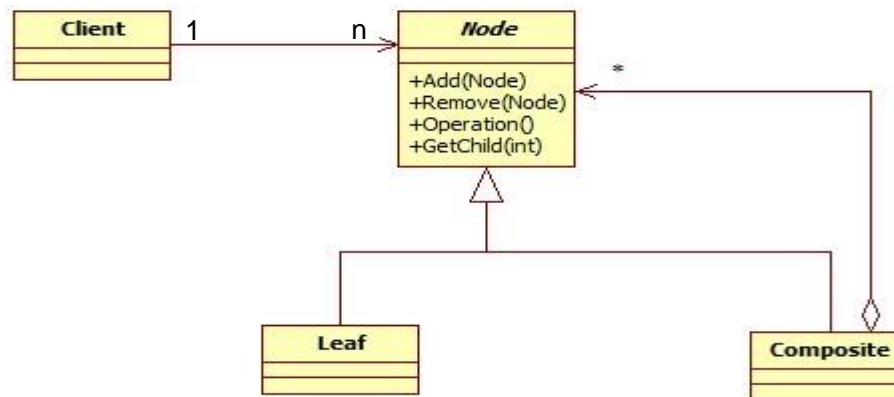
# Souvislost s kapitolou OON



# Jak souvisí s předchozí přednáškou

Diagram tříd – asociace mezi třídami

- 1:1 (atribut)
- 1:N (pole, množina, seznam, tabulka, ...)
- N:1 (atribut)
- N:N (pole, množina, seznam, tabulka, ...)



- Souvisí s dědičností?

# Způsob reprezentace informace

- Informace je často nejcennější prvek informačních systémů
- Abstraktní datové typy (ADT) jsou elementární stavební prvky pro reprezentaci informace
- **Problém:** Čas X Vyjadřovací síla
- Redundantní informace
  - práce navíc (zajištění konzistence)
  - Umožní vyšší výkon

# Proč znát a používat ADT?

Příklad:

- Google
  - Zpracovává 24 petabytů / den
  - Miliony uživatelů
  - Odpověď v řádu milisekund
  - Bez znalosti datových struktur neřešitelný problém
- Databáze Teradata – ukládá 50 petabytů dat
- German Climate Computing Center (DKRZ) uchovává 60 petabytů dat
- Celkový objem digitálních dat 500 Exabytů (rok 2009)
- Lékařské databáze, webové aplikace (WebNode)

yotta  
zetta  
exa  
peta  
tera  
giga  
mega  
kilo

# Proč znát a používat ADT?

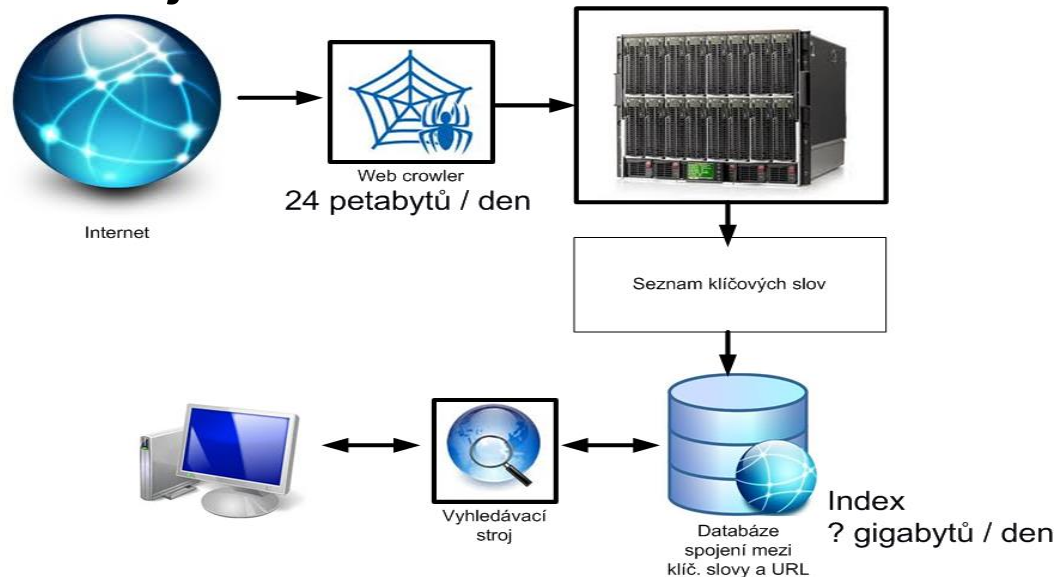
- Jsme schopni reprezentovat informaci v paměti počítače

A současně

- Volbou vhodných datových struktur výrazně ovlivníme časové nároky programu
- Jsme schopni dosáhnout efektivního využití HW prostředků dané platformy

# Příklad: Jak to Google dělá?

- Vyhledávací stroj = vytváření indexu klíčových slov pro rychlé zpřístupnění uživatelům
- Bez znalosti TI je nerealizovatelné





# Příklad: Hra čísla

- Lze řešit strojově (řekneme si jak – viz kap. Grafy)
- Vhodný návrh struktur – řešení v řádech sekund
- Nevhodný návrh algoritmů – týdny / roky
  - Vysoce pravděpodobné, že variantu „týdny“ zvolíte (zkušenost z předchozích let výuky MTIN)

8		6
5	4	7
2	3	1

	1	2
3	4	5
6	7	8

- Příklad: Databázové systémy
  - Při jednoduchém ukládání „na disk“ – velice pomalé
  - Vytváření indexů (redundance & řešení konzistence)

# Příklad: RTP

- Real time transport protocol (RTP)
  - UDP: Změna pořadí paketů, ztráty, duplicity
  - Velký počet paketů
  - Různý HW
- Kodeky – efektivní organizace rámců (přesun, přeskokování atp.)

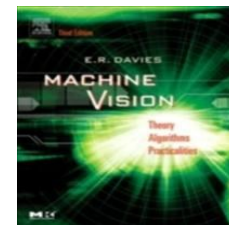


# Datové struktury

- Znalost datových struktur vám dovolí pohybovat se na hraně spočitatelnosti
  - Návrh zařízení datové komunikace (CISCO switche, routery)
  - Počítačové vidění (reálný čas)
  - Strojová analýza hlasu
  - Lámání šifer
  - Analýza genomu člověka
  - Bezpečnostní systémy
  - Návrh hardware



???



# Varianty pro reprezentaci informace

- Úvod (podrobnosti v jednotlivých kapitolách)
- Vědět co umí, jak pracují (= jak jsou pro dané použití časově náročné)

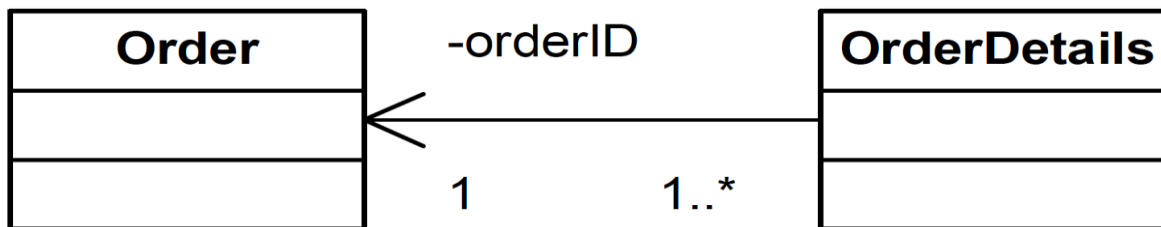


# Varianty pro reprezentaci informace – rozdělení

- Atribut
- Pole
- Lineární seznam
  - Fronta (FIFO)
  - Zásobník (LIFO)
- Hash tabulka
  - Množiny
  - Mapování
- Vyhledávací binární stromy
  - Množiny
  - Mapování

# Opakování – Asociace 1:1, N:1

- Jednoduché, časově i paměťově velice efektivní
- Je možné se odkázat na konkrétní hodnotu

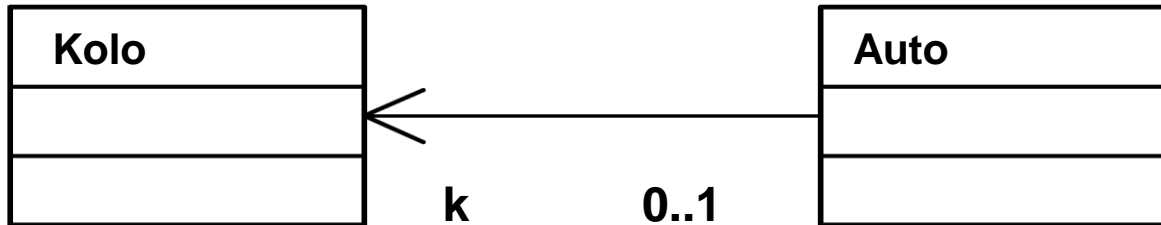


```
public class Order {  
  
}
```

```
public class OrderDetails {  
    private Order details;  
  
    public Order getDetails() {  
        return details;  
    }  
  
    public void setDetails(Order details) {  
        this.details = details;  
    }  
}
```

# Opakování – Asociace 1:k, N:k

- Jednoduché, časově i paměťově velice efektivní
- Je možné se odkázat na konkrétní hodnotu



```
public class Kolo {  
  
}
```

Asociace o konkrétní násobnosti  $k$   
Pozn: Nepoužívá se příliš často -  
nepraktické

```
public class Auto {  
    private Kolo[] auto = new Kolo[4];  
  
    public Kolo getKolo(int index) {  
        return kolo[index];  
    }  
  
    public void setKolo(Kolo kolo, int index) {  
        this.kolo[index] = kolo;  
    }  
}
```

# Opakování – Asociace 1:N, N:N

- V mnoha případech je nutné odkazovat se z jednoho objektu na N objektů
- Příklad:
  - 1:N ... Typ zboží vs. objednávka (internetový obchod)
  - N:N ... Student vs. kurz (kurz navštěvuje několik studentů, studenti navštěvují několik kurzů)
- Pro implementaci násobné asociace existuje několik základních variant (v 99,9 % si s nimi vystačíte pro reprezentaci libovolného problému)
  - Pole (konstantní délky)
  - Lineární seznamy
  - Hash tabulky – (*nepoužívat*)
  - Vyvažované stromy - (*nepoužívat*)
  - Množiny (HashSet = založená na Hash tabulce, TreeSet = založená na vyvažovaném stromu)
  - Mapování (HashMap (HashSet), TreeMap (vyvažovaný strom))



# Pole

- Jednoduché na implementaci, v paměti zabírá konstantní prostor a má konstantní délku

```
int[] mojePole = new int[1000];
```

- Pole lze vytvořit i z objektů:

```
Auto[] mojeAuto = new Auto[1000];
```

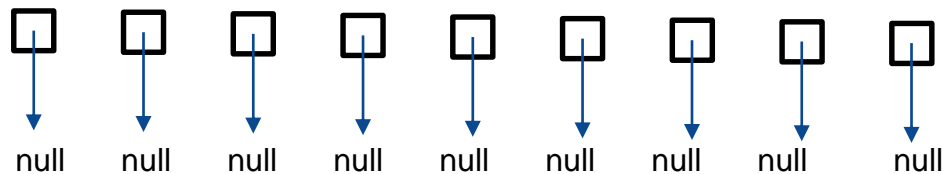
- Jaké budou hodnoty na jednotlivých pozicích C++ / JAVA?
- Pozn: „Holý“ se používá jen zřídka, rozšířené varianty: ArrayList, Vector

```
int[] mojePole = new int[9];
```

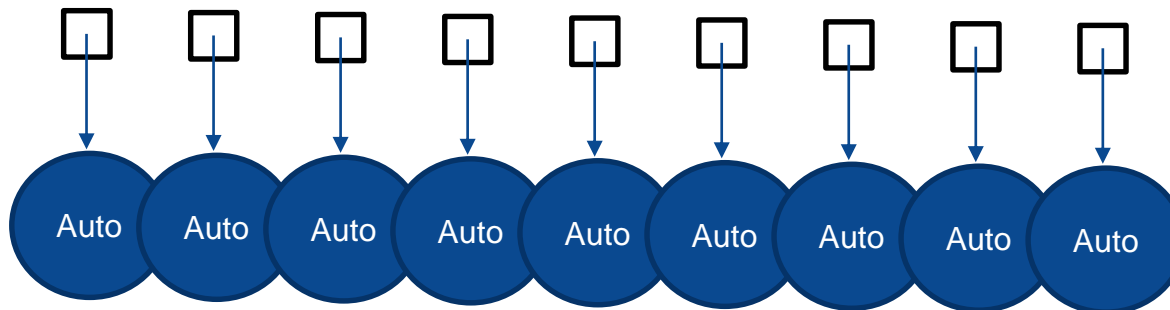


Poznámka: bude  
i u C++ všude 0?

```
Auto[] mojeAuto = new Auto[9];
```



```
for (int i = 0; i < 9; i++) {  
    mojeAuto[i] = new Auto();  
}
```



# Pole pomocí rozšířených variant

- ArrayList 

```
ArrayList<Integer> arr = new ArrayList<Integer>();  
arr.add(100);  
arr.add(200);  
arr.size();  
arr.contains(100);
```
- Vector – synchronizovaná obdoba ArrayList  

```
Vector<Integer> arr = new Vector<Integer>();
```
- Běží vaše aplikace ve více vláknech? Je možné, že více aplikací bude přistupovat současně k tomuto místě v paměti? => použít Vector

# Lineární seznam

- V paměti zabírá proměnlivý prostor (dle aktuální potřeby) a má proměnlivou délku (stejně jako všechny ostatní kromě pole)
- Tomu jak vnitřně funguje se budeme věnovat detailně později

# Lineární seznam

- Varianty

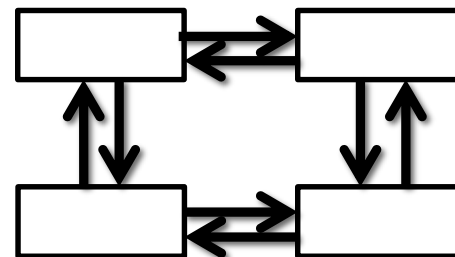
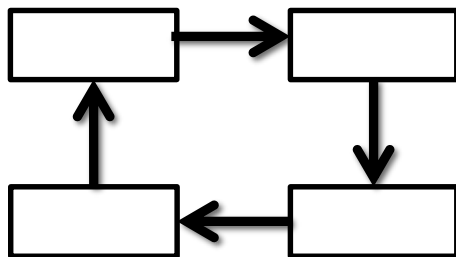
- Jednosměrně vázaný



- Obousměrně vázaný

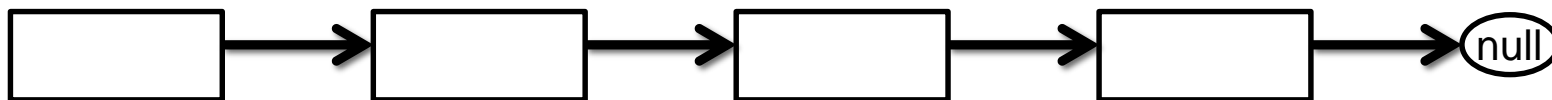


- Cyklický



# ADT Zásobník(LIFO)

- ADT Frontu lze realizovat pomocí ADT lineární seznam
- Přidání prvku = vložení na začátek seznamu
- Odstranění prvku = odstranění ze začátku seznamu
- POZN. Pro realizaci fronty se využívá tzv. oboustranně vázaný seznam (šipka tam i zpět) – viz pozdější kapitoly.



# ADT Fronta (FIFO)

- ADT Frontu lze realizovat pomocí ADT lineární seznam
- Přidání prvku = vložení na konec seznamu
- Odstranění prvku = odstranění ze začátku seznamu
- POZN. Pro realizaci fronty se využívá tzv. oboustranně vázaný seznam (šipka tam i zpět) – viz pozdější kapitoly.



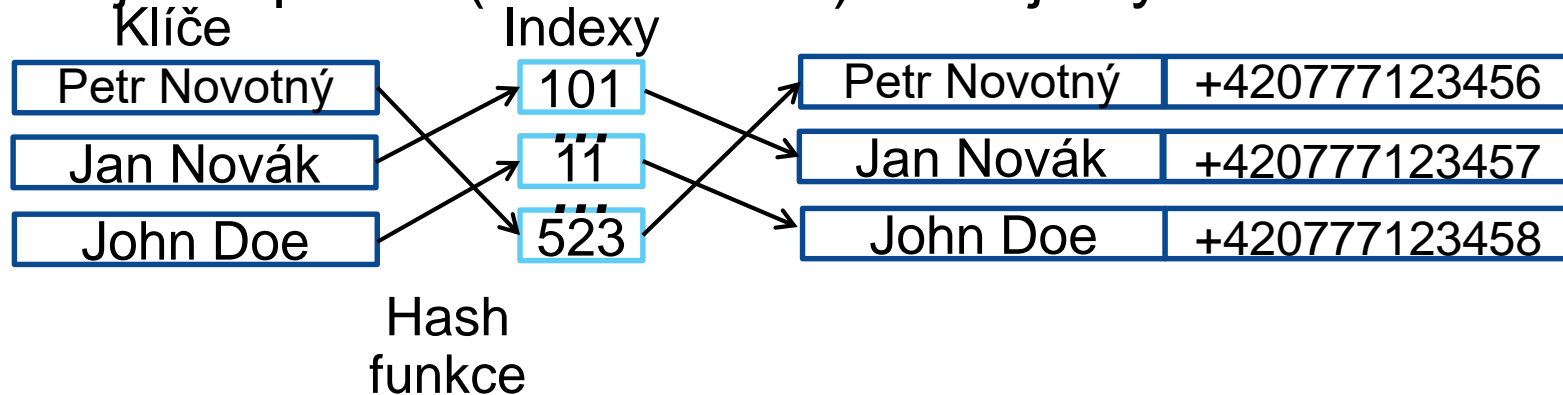
# Příklad – lineární struktury

- **Demo:**
  - **LinkedList<typ>**
  - Jak reprezentovat frontu
  - Jak reprezentovat zásobník
- Příklad: Vytvořte vyrovnávací paměť (buffer), pro přehrávač videa z protokolu RTP. Předpokládejme, že prvky již přichází seřazený.
-



# Hash Tabulka

- Co je to Hash Tabulka? Jak pracuje?
- Klíč je mapován (hash funkce) na objekty



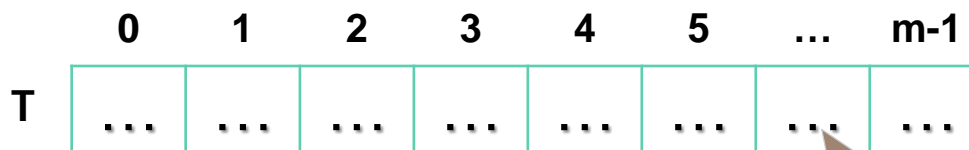
Zpravidla jednoduchý

I komplexnější objekty

- Zpravidla se používá pro reprezentaci množin a mapování

# Hash Tabulka

- Hash tabulka je pole, které udržuje množinu párů (klíč, hodnota)
- Proces mapování klíče na pozici (=indexu) v této tabulce se nazývá **hashování**



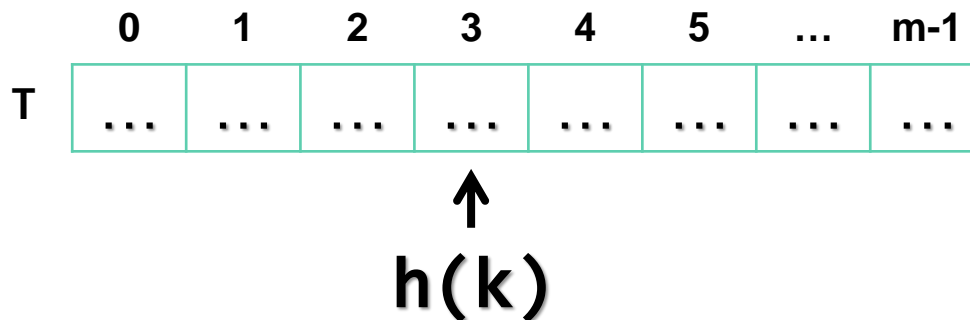
$h(k)$

**Hash funkce**  
 $h: k \rightarrow 0 \dots m-1$

**Hash  
tabulka  
velikosti  $m$**

# Hash funkce a Hashování

- Hash tabulka má  $m$  pozic, indexovaných v rozsahu  $0$  až  $m-1$
- Hash funkce  $h(k)$  mapuje klíče na pozici :
  - $h: k \rightarrow 0 \dots m-1$
- Pro každou hodnotu klíče  $k$  (v rámci jeho rozsahu) a nějakou hash funkci  $h$  získáváme hodnotu  $h(k) = p$  kde  $0 \leq p < m$



# Hash funkce

- Perfektní hashovací funkce (PHF)
  - $h(k)$  : 1:1 mapování každého klíče  $k$  na celočíselnou hodnotu v rozsahu  $\langle 0, m-1 \rangle$
  - Ideálně by mapoval každou hodnotu na unikátní celočísl. hodnotu v nějakém definovaném rozmezí
- Nalezení perfektní hash funkce je ve většině případů **nemožné** (paměť, rychlost & bez kolizí)
- **Realita**
  - Hash funkce  $h(k)$  mapuje většinu klíčů na unikátní celočísl. hodnotu (ale může vzniknout duplicita = **KOLIZE**).

# Kolize a Hash Tabulky

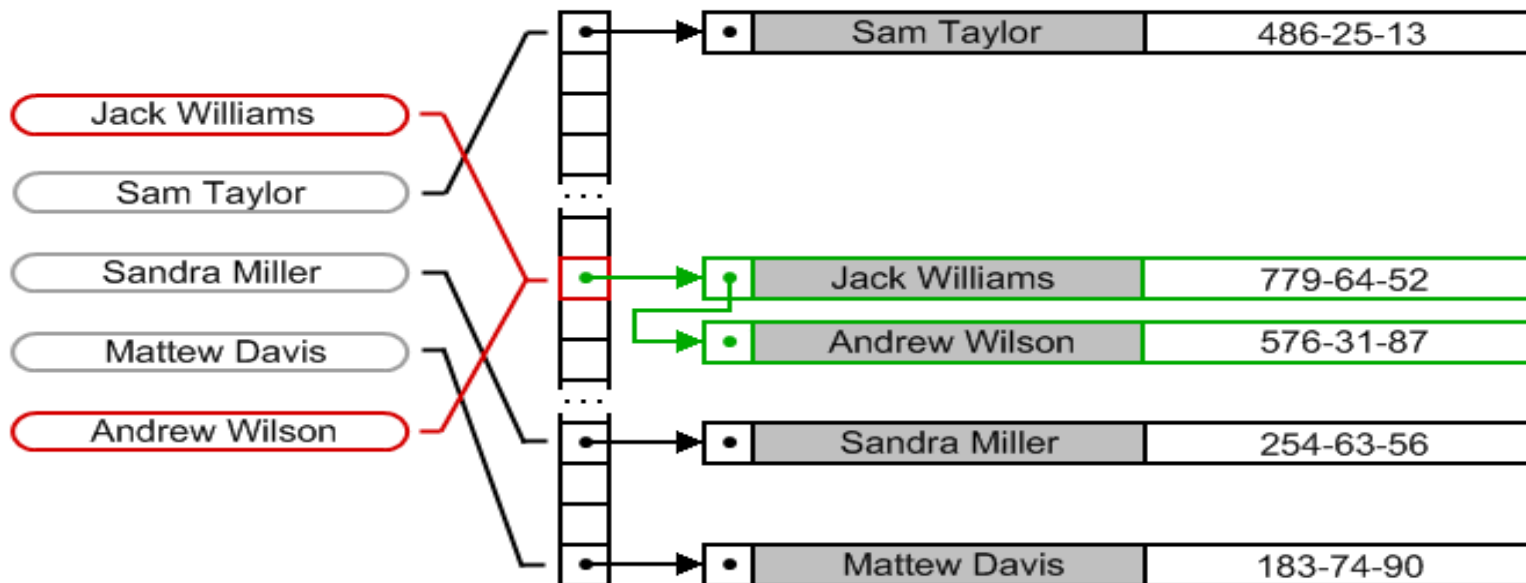
- Kolize je situace, když dva rozdílné klíče jsou zobrazeny s pomocí hashovací funkce  $h$  do stejné pozice v hash tabulce

$$h(k_1) = h(k_2) \text{ pro } k_1 \neq k_2$$

- Pokud je počet kolizí malý, hash tabulky pracují velice rychle
- Jak se vyhnout kolizím?
  - Zřetězení v seznamu (chaining)
  - Použití sousedních pozic za aktuální (linear probing)
  - Re-hashing (opětovné přehashování & nalezení nové pozice)

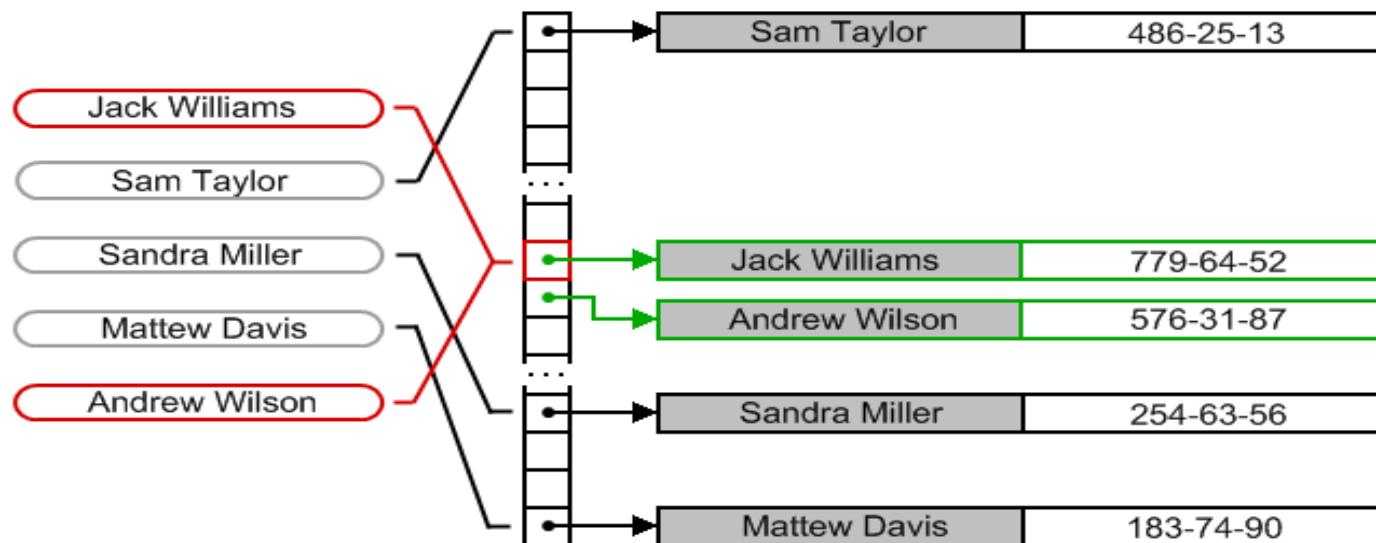


# Zřetězení (Chaining)



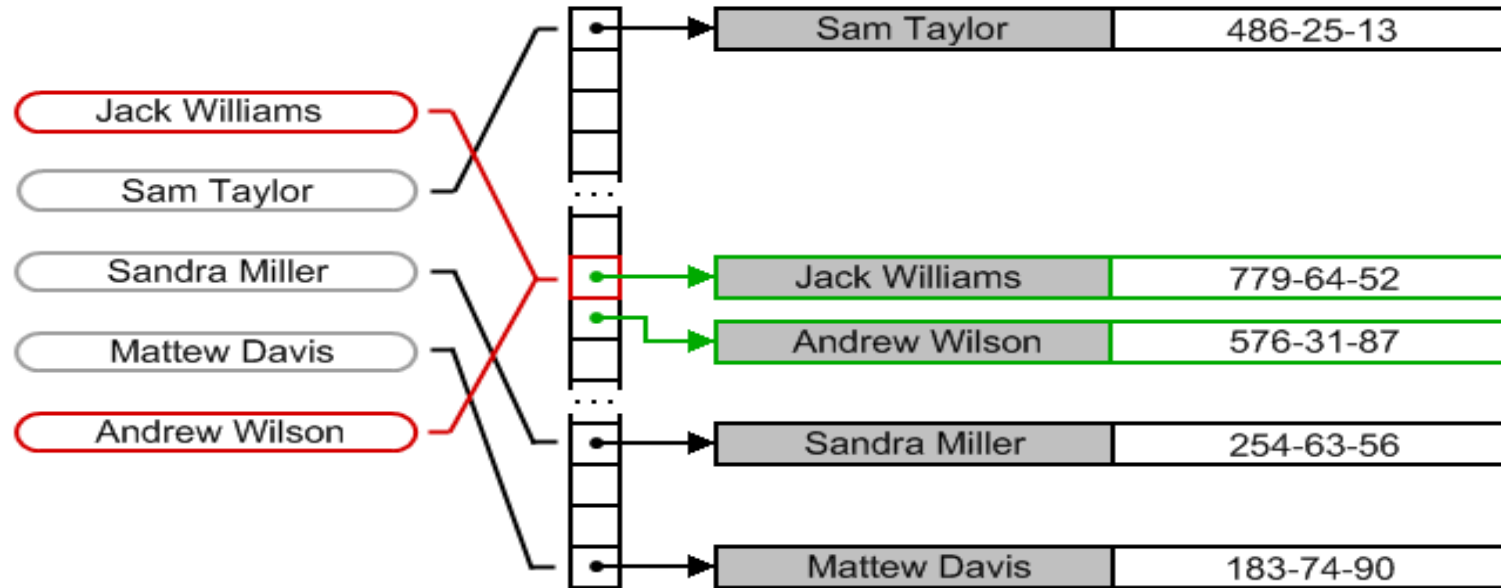
- Vytváří seznam na pozici s kolizí
- Zdroj: [http://www.algolist.net/Data\\_structures/Hash\\_table/Chaining](http://www.algolist.net/Data_structures/Hash_table/Chaining)

# Lineární snímání (Linear probing)



- Vezme hodnotu na následující volné pozici
- Zdroj:  
[http://www.algolist.net/Data\\_structures/Hash\\_table/Open\\_addressing](http://www.algolist.net/Data_structures/Hash_table/Open_addressing)

# Dvojité hashování (Double hashing)



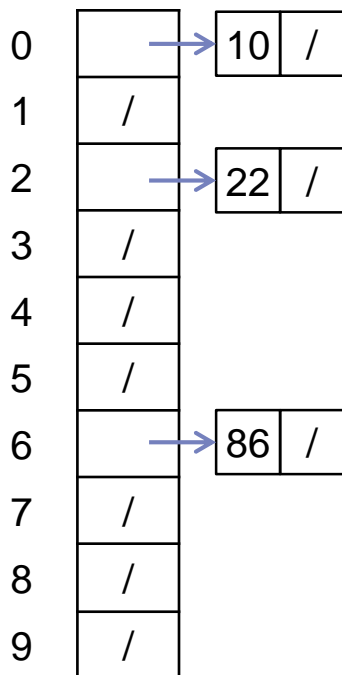
- Spočte znovu hash funkci v kombinaci s novou pozicí
- Ke druhému výpočtu se použije jiná hash funkce



# Dvojité hashování (Double hashing)

- Dobře zvolená  $\text{Hash}_2(X)$  by měla garantovat, že neuvízne dokud  $\lambda < 1$
- $\text{Hash}_2(X) = R - (X \bmod R)$   
kde  $R$  je prvočíslo menší než velikost tabulky

# Příklad: řetězení



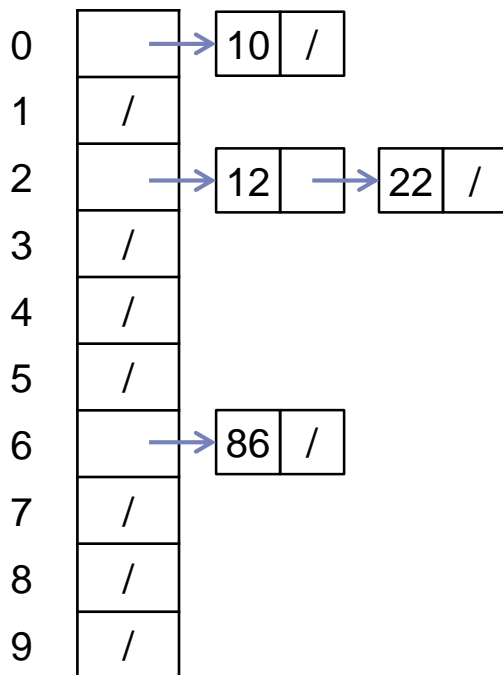
Všechny hash hodnoty, které jsou stejné a jsou tedy mapovány na stejnou pozici, jsou udržovány v lineárním seznamu

Příklad:

vložte 10, 22, 86, 12, 42

hash:  $h(x) = x \% 10$

# Příklad: řetězení



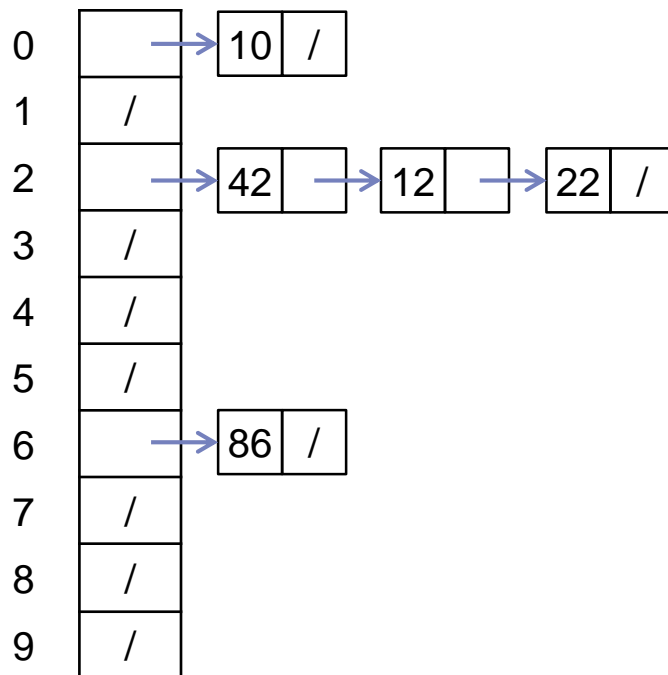
Všechny hash hodnoty, které jsou stejné a jsou tedy mapovány na stejnou pozici, jsou udržovány v lineárním seznamu

Příklad:

vložte 10, 22, 86, 12, 42

hash:  $h(x) = x \% 10$

# Příklad: řetězení



Všechny hash hodnoty, které jsou stejné a jsou tedy mapovány na stejnou pozici, jsou udržovány v lineárním seznamu

Příklad:

vložte 10, 22, 86, 12, 42

hash:  $h(x) = x \% 10$

# Příklad: Dvojitě hashování

kolize

$$h_1(x) = x \% 7$$

$$h_2(x) = 5 - x \% 5$$

insert(14)  
 $14 \% 7 = 0$

insert(8)  
 $8 \% 7 = 1$

insert(21)  
 $21 \% 7 = 0$   
 $5 - (21 \% 5) = 4$

insert(2)  
 $2 \% 7 = 2$

insert(7)  
 $7 \% 7 = 0$   
 $5 - (7 \% 5) = 3$

0	14
1	
2	
3	
4	
5	
6	

0	14
1	8
2	
3	
4	
5	
6	

0	14
1	8
2	
3	
4	21
5	
6	

0	14
1	8
2	2
3	
4	21
5	
6	

0	14
1	8
2	2
3	7
4	21
5	
6	

umístění:

1

1

2

1

??

# Příklad: Mazání v řetězení

Co když smažeme hodnotu?

delete(2)

0	0
1	1
2	2
3	7
4	
5	
6	

find(7)

0	0
1	1
2	
3	7
4	
5	
6	

Obsahuje 7, za situace, že byl  
smazáním řetězec přerušen?!

# Řetězení: Líné mazání

delete(2)

0	0
1	1
2	2
3	7
4	
5	
6	

find(7)

0	0
1	1
2	#
3	7
4	
5	
6	

Indikuje, že smazaná hodnota  
odtud byla smazána.

# Hash tabulky a jejich efektivita

- Hash tabulky se používají pro reprezentaci množin
- Jsou nejefektivnější implementace (výkon)
- Přidat / Vyhledat / Smazat znamenají jen několik málo primitivních operací
  - Rychlost nezáleží na velikosti hash-tabulky (konstantní čas), (při zřetězení to znamená lineární čas, to ale není běžný případ).
  - Příklad: nalezení prvku v hash-tabulce (množině) s 1 000 000 prvků, znamená výpočet HASH, ověření, zdali KLÍČ odpovídá záznamu a případné řešení kolize
    - Nalezení prvku v poli o 1 000 000 prvků (neseřazené zlaté stránky) znamená přibližně 500 000 kroků



# Hash tabulky a jejich požadavky

- V jazyku JAVA vždy k dispozici (obdobně další OO jazyky)
  - Metoda *hashCode()* a *equals()* - je možné přetížit jejich chování a tím i chování např. množin či map založených na hash tabulkách
- Hash Funkce
  - Ne vždy je triviální vytvořit vlastní efektivní funkci (HashCodeBuilder, apache lang commons).
- Požadavky na objekty do nich vkládaných:
  - Existence *Hash funkce* (výchozí chování pracuje nad objekty)
  - Metoda porovnávání by měla být konzistentní s hash funkcí, tj. hlásit shodu tehdy, když srovnání s objekty hlásí shodu.

# Demonstrace

## • Demo Hashtable

```
public class Main {  
    public static void main(String[] args) {  
        // Hashtable, ktera ma klic typu String (napr "Jan Novak")  
        // a klic objekt typu Osoba  
        Hashtable<String, Osoba> mujSeznamLidi = new Hashtable<String, Osoba>();  
  
        // vytvorim si osobu  
        Osoba o1 = new Osoba("Jan Novák", "+420 777 123 456");  
  
        System.out.println("Hodnoty objektu o1 :");  
        System.out.println("\t " + o1.getJmeno());  
        System.out.println("\t " + o1.getTelefon());  
  
        // vložím osoby do hash tabulky, klic bude jmeno osoby  
        mujSeznamLidi.put(o1.getJmeno(), o1);  
  
        // Přidám další osobu  
        Osoba o2 = new Osoba("Pavel Novák", "+420 777 123 457");  
        mujSeznamLidi.put(o2.getJmeno(), o2);  
  
        System.out.println("Velikost hash tabulky: " + mujSeznamLidi.size());  
        System.out.println("Jan Novák ma cislo: "  
            + mujSeznamLidi.get("Jan Novák").getTelefon());  
        System.out.println("Pavel Novák ma cislo: "  
            + mujSeznamLidi.get("Pavel Novák").getTelefon());  
    }  
}
```

```
class Osoba {  
    private String jmeno;  
    private String telefon;  
  
    public Osoba(String jmeno, String telefon) {  
        this.jmeno = jmeno;  
        this.telefon = telefon;  
    }  
  
    public String getJmeno() {  
        return jmeno;  
    }  
  
    public String getTelefon() {  
        return telefon;  
    }  
}
```

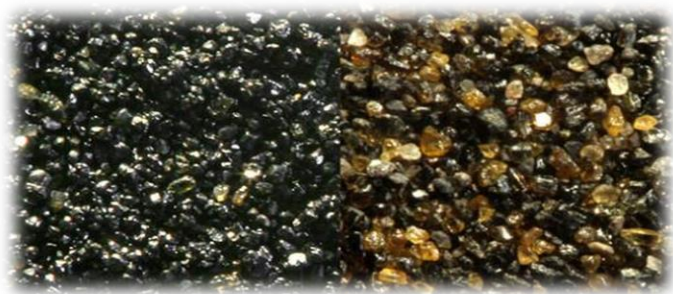
# Demonstrace

- Jak získat z internetu příklad použití?
- Jak získat seznam metod této třídy?
- Jak zjistit vztahy v rámci dědičnosti (souvisí s předchozím)

# Příklad: Jaká slova se vyskytují v textu?

- Vstup: řetězec oddělený „“, „.“, „„“ atp.
  - POZN: Srovnejte výkonnostně s jinými implementacemi
  - POZN: Využívá se často např. v strojových analyzátorech textu

# Vyvážené binární stromy



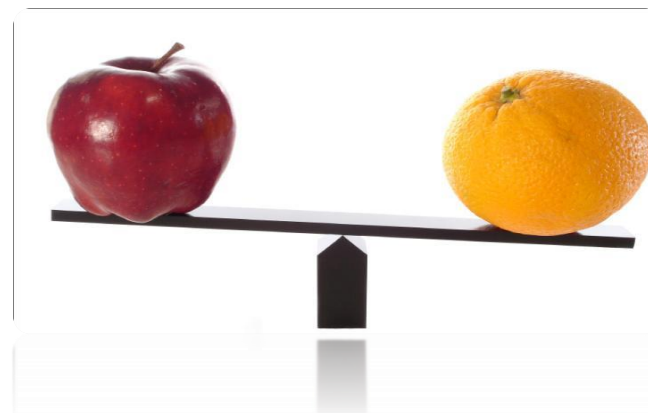
# Vyvážené binární stromy

- Operace: Přidat / Vyhledat / Smazat prvek provádí  $\log_2(n)$  operací, kde  $n$  je počet uchovávaných prvků
- Prvky jsou současně řazeny do patřičného pořadí
- Efektivní odstraňování duplicit
  - Využití realtime přehrávačů – změna pořadí a duplicita paketů

# Vyvážené binární stromy

Srovnání:

- Stromy:  $1\,073\,741\,824 = 2^{30} \dots \log_2(2^{30}) = 30$  operací
- Lineární seznamy, pole, ...  $1\,073\,741\,824 =$  operací
- Problematice vyvažovaných stromů a jejich fungování se budeme věnovat detailně – nyní jen z „uživatelského pohledu“.



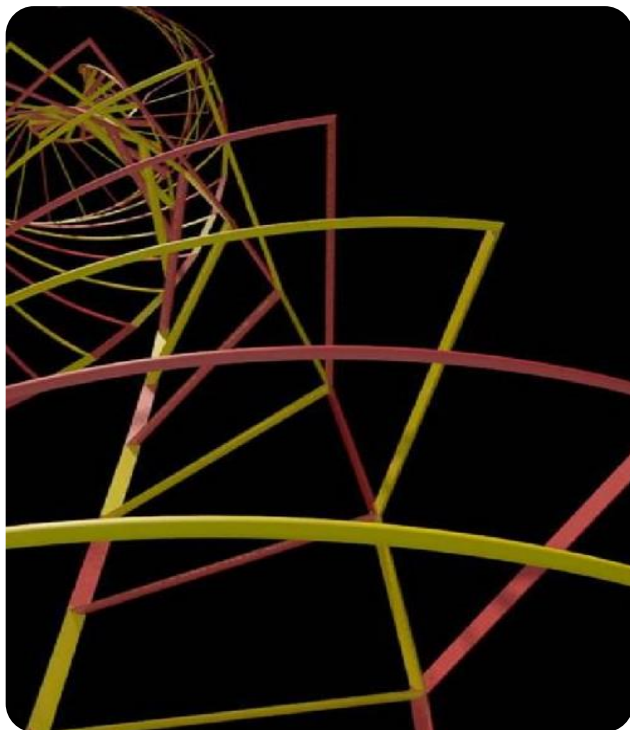
# Vyvážené stromy – Demonstrace

- Demonstrace – seznam výskytů slov
- Srovnání HashSet vs. TreeSet



# Množiny

- Množiny prvků

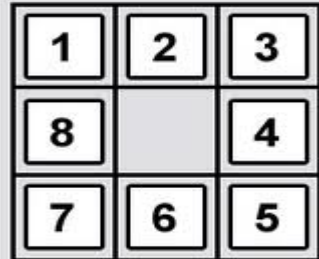


# ADT množina

- Množiny jsou nezbytnou strukturou pro reprezentaci informace
- Abstraktní datový typ (ADT) „množina“ udržuje prvky bez výskytu duplicit
- Operace nad množinou:
  - `Add(element)`
  - `Contains(element) → true / false`
  - `Delete(element)`
  - Někdy také: `Union(set) / Intersect(set)` – jak snadno realizovat pomocí předchozích?

# ADT množina

- Množiny mohou být implementovány mnoha způsoby: List, array, hash tabulka, vyvážený strom, ..., z pohledu výkonu je ale v 99 % případů vhodný pouze pomocí Hash tabulek (hash tabulka) anebo vyvažovaných stromů (HashSet)
- Příklad: Strojové řešení hry čísla – 18 sec vs. několik dní při použití lineární struktury (strom / seznam)



1	2	3
8		4
7	6	5

# HashSet<T>

- **HashSet<T>** implementuje ADT **množina s použitím hash tabulky**
  - Prvky jsou v náhodném pořadí (odlišnost od TreeSet)
- Velice výkonnostně efektivní:
  - **add(element)** – přidá prvek do množiny
    - Nedělá nic, pokud již existuje
  - **remove(element)** – odstraní daný prvek z množiny
  - **size()** – zjistí velikost množiny
  - **addAll(set)** – provede operaci sjednocení

# TreeSet<T>

- **TreeSet<T>** implementuje ADT **množinu** s využitím binárních vyhledávacích stromů
  - Prvky jsou řazeny (rozdíl od HashSet) v rostoucím pořadí

# Přizpůsobení chování množin

Množina – pokud vložím 2x jeden prvek, objeví se v množině pouze 1x

Výchozí chování – rozlišování dle místa v paměti (pointeru)

Aby bylo možné uzpůsobit chování, je nutné změnit následující metody:

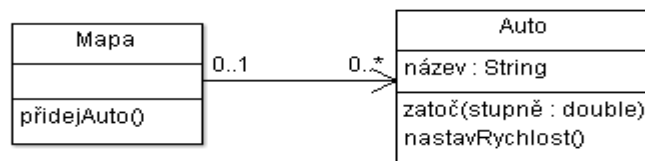
TreeSet<T>	<i>Comparator</i> anebo <i>implements Comparable&lt;T&gt;</i>
HashSet<T>	<i>hashCode()</i> a současně <i>equals()</i>

# ADT mapování

- „Mapuje“ = přiřazuje hodnotu ke klíči (tj. vytváří dvojici klíč – hodnota)
- Klíč musí být unikátní
- Existují dvě základní možné implementace
  - `HashMap<typ klíč, typ hodnoty>` (náhodně řazené položky)
  - `TreeMap<typ klíč, typ hodnoty>` (seřazené položky dle klíče)
- Příklad:
  - Spočtete počet výskytů jednotlivých slov v textu
    - DEMO

# Příklad I.

- Mějme hru Need4Speed a datový model, kde je mapa a v ní auta. Rozšiřte příklad tak, aby tam bylo možné dát libovolný počet aut do mapy.
- Jak zajistit, aby bylo možné z datového modelu zjistit polohu jednotlivých aut?

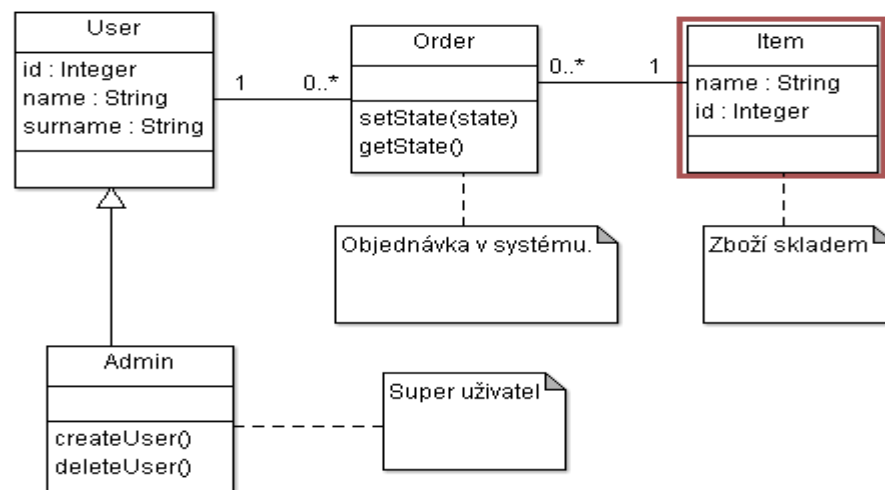




## Příklad II.

- Navrhnete internetový obchod, kde bude vystupovat administrátor a běžný uživatel. Administrátor je schopen dělat vše, co může dělat běžný uživatel. Běžný uživatel může vytvářet objednávky na základě zboží ve skladu.

- Jak bude reprezentována asociace
- User->Order ?
- Item->Order ?

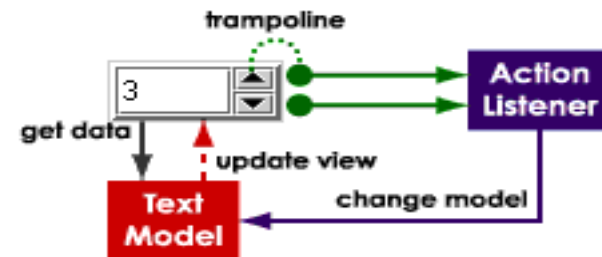
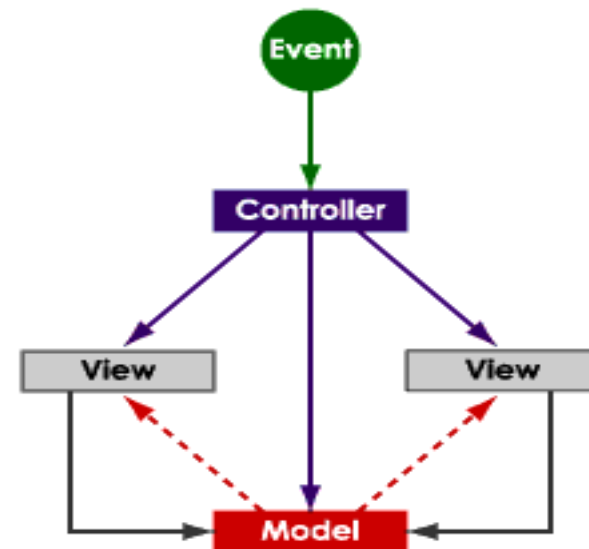


# Souvislosti

- Proč se zabýváme: informace je nedílnou součástí drtivé většiny informačních systémů a aplikací (MS Word, MS Windows, Linux, Firefox, Webový portál, internetový obchod, hry, ...), ale i HW, komunikačních protokolů atp.
- Informace (nazývaná také jako datový model) často prostupuje celou aplikací, ale je nezávislá na GUI, HTML, databázi atp.
- Způsob jak bývá oddělen datový model od zbytku aplikace bývá s pomocí Model-View-Controller

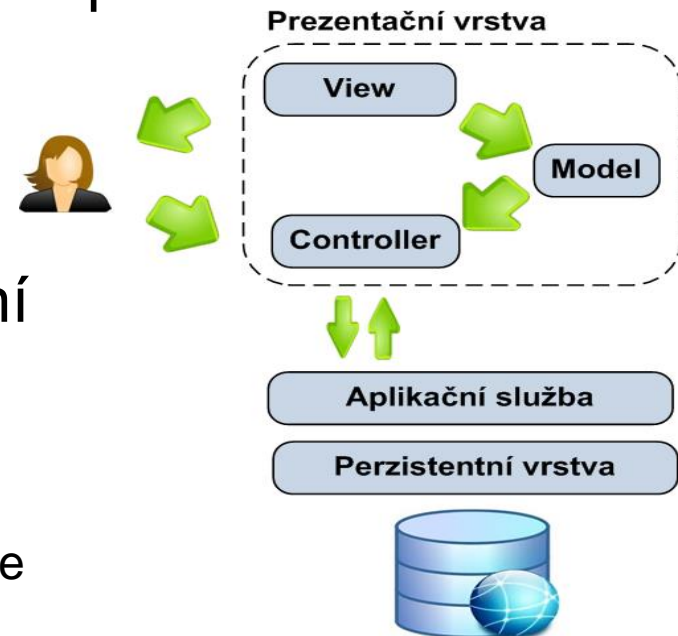
# Souvislosti

- Návrhový vzor Model-View-Controller
  - Událost obsluhuje Controller
  - Controller rozhoduje jak zobrazí výsledek
  - View zobrazuje výsledek
  - View může generovat události
  - Model – datová reprezentace informace



# Souvislosti

- Větší aplikace (kde se plánuje doba údržby kódu na delší dobu  $> 1$  rok)
- Výhoda – vznik znovupoužitelných komponent
- Snadno se dělá více rozhraní
  - GUI, HTML, RSS, ...
- Nevýhoda – pro nezkušené uživatele obtížné pro pochopení
- Jak OO model uložit do relační DB?
  - Object-to-realtion Mapping – např. hibernate



# Shrnutí

- Informace = datový model je stěžejní část drtivé většiny aplikací
- Implementace asociací 1:1, 1:k
- Implementace násobností 1:N
- Hash-tabulky mapují klíče k mnoha hodnotám
  - Založeny na hash-funkci (distribuce klíčů v tabulce)
  - Kolize potřebují alg. pro vyřešení kolize (např. řetězení)
  - Rychlé přidat / nalézt / smazat
- Množiny udržující skupinu prvků
  - Implementace: Hash-tabulka nebo vyvažovaný strom



# Děkuji za pozornost