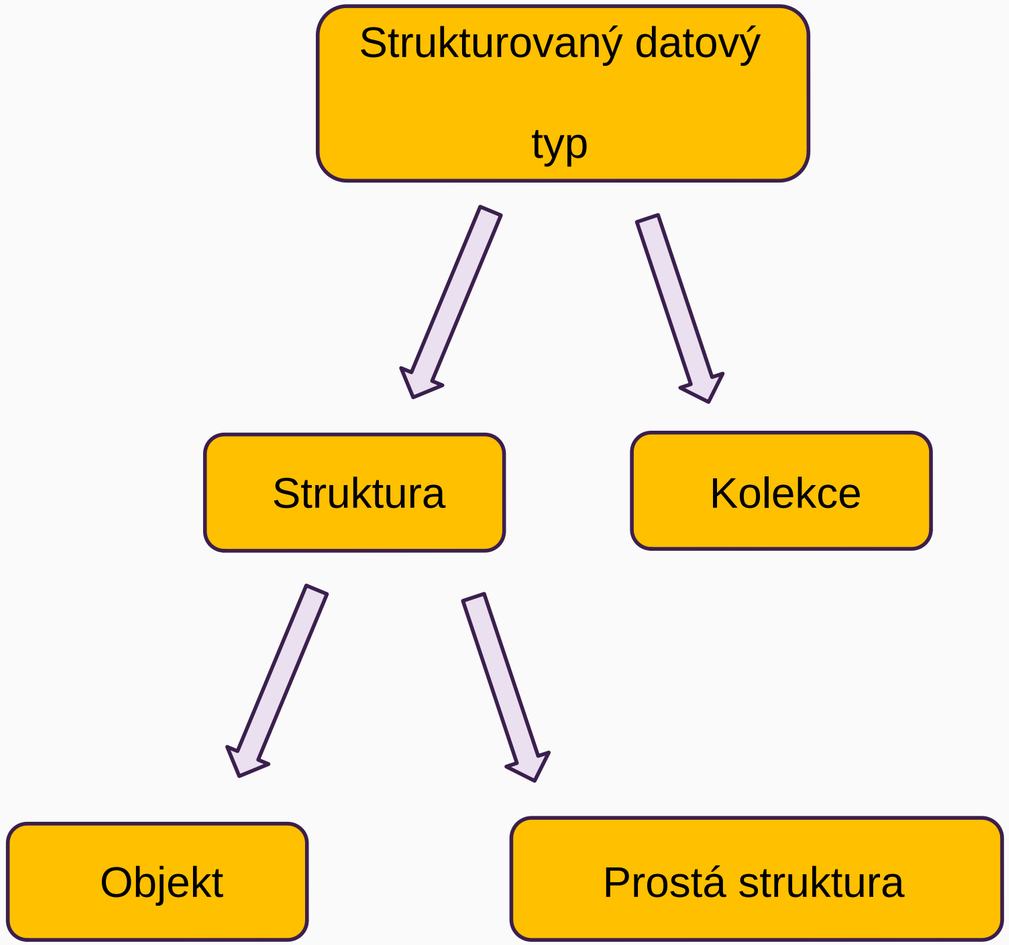
Pokročilé informační systémy

Objektový model dat

Prof. Ing. Tomáš Hruška, CSc.
Doc. Ing. Radek Burget, Ph.D.
burgetr@fit.vutbr.cz

Strukturované datové typy



Databázové modely

- Jednoduché (NoSQL)
  - Key-value (MUMPS, Redis, …)
  - Dokumentové (MongoDB, CouchDB, …)
  - Sloupcové (Apache HBase, …)
- **Relační datový model**
  - Mnoho implementací
- **Objektový datový model**
  - Objektově-relační mapování (ORM)
- Grafové
  - Grafové databáze (Neo4J, OrientDB, …)
  - Sémantická úložiště (sémantický web, RDF)

Relační datový model

- Tabulka (= **relace**) v relačním modelu je *kolekcí struktur*, přičemž datové typy vlastností jsou *jednoduché* (tedy především *ne odkazy/vztahy*)
- Srovnej: *Podmnožina kartézského součinu*

```
collection of
    structure
        properties
            jméno vlastnosti1: jednoduchý datový typ1
            jméno vlastnosti2: jednoduchý datový typ2
            …
            jméno vlastnostin: jednoduchý datový typn
    end structure
```

Objektový datový model

- Motivace: v aplikacích obvykle modelujeme data objektově
  - Objektově-orientované modelování v návrhu IS, UML
- Data reprezentovaná pomocí konceptů objektově orientovaného modelování
  - Třídy, objekty (instance)
- Vztahy (reference)
  - Na rozdíl od relačních databází (nemluvě o NoSQL)

Třídy a objekty jako datový model

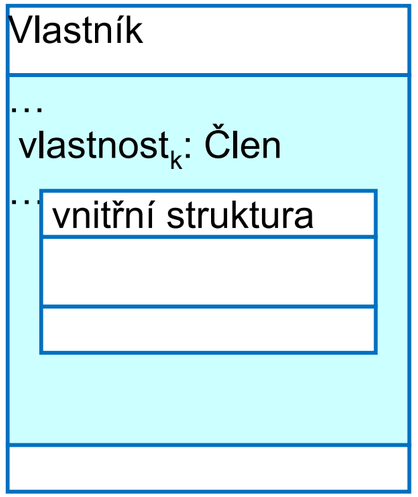
- Dále řešené koncepty (třídy, objekty, dědičnost, apod.) jsou podobné OO návrhu SW
- Zde ale mluvíme o datovém modelu – **modelujeme data**
- Třídu vnímáme jako **datový typ** (strukturovaný)
  - Množina hodnot (objektů), potenciálně nekonečná
  - Žádné procedurální metody

Struktura objektů, vztahy

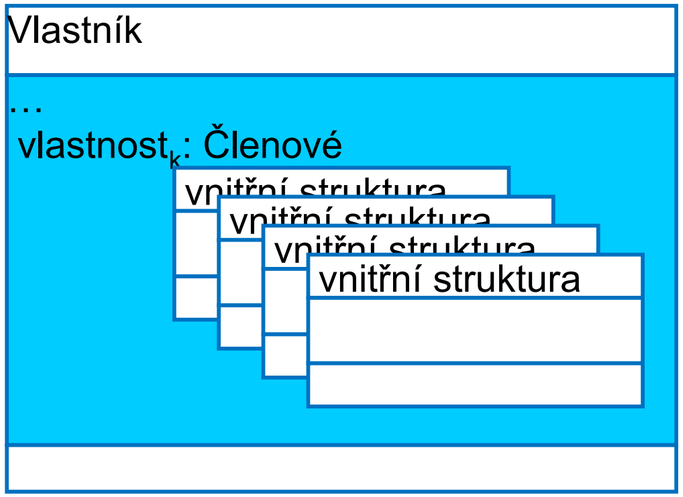
Vztahy

- Umožňují odkazovat z jedné (strukturované) hodnoty (vlastníka) jinou (člen)
- Musí existovat datový typ **jednoznačné identifikující (odkazující) strukturovanou hodnotu** (např. OID)

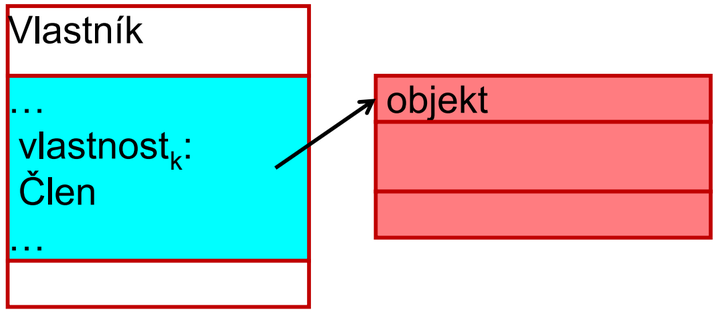
Zanoření a vztahy



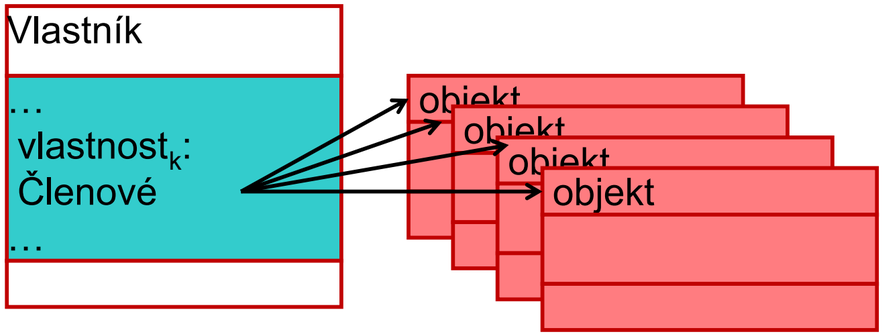
1. člen je jedinou prostou strukturou = vnoření hierarchie struktur



2. členové jsou v kolekci prostých struktur = vytváření hierarchie struktur svnořenými kolekcemi



3. člen je jediným objektem = vytváření vztahu typu 1:1

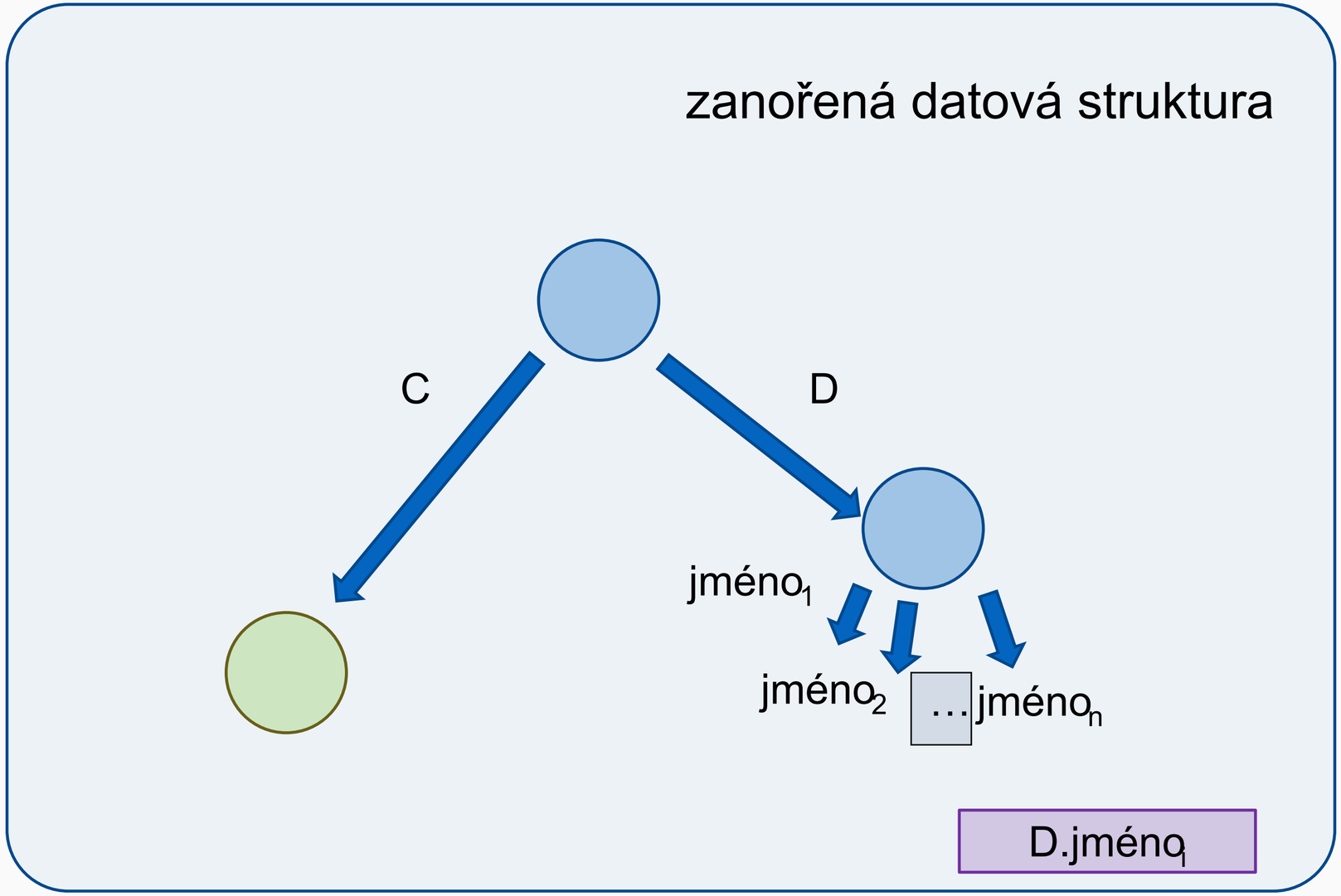


4. členové jsou v kolekci objektů = vytváření vztahu typu 1:N

Vztahy

- Relační model dat vztahy přímo neobsahuje
  - Vytváří se až v okamžiku dotazování (JOIN apod.)
  - (Neplést s referenční integritou!)
- Objektový model
  - Vztahy lze tvořit pomocí OID a/nebo vnořováním struktur

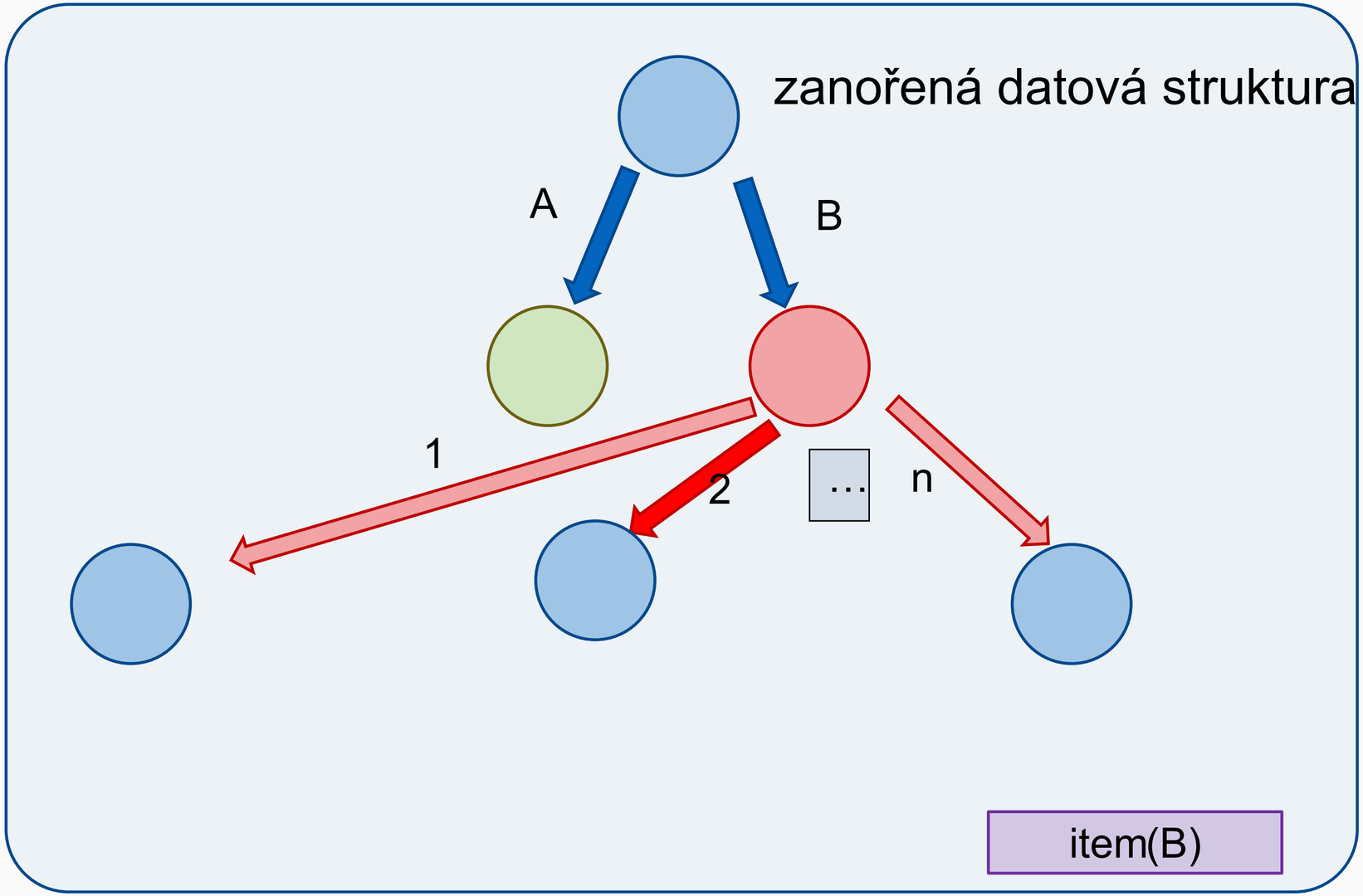
Jmenný prostor nižší úrovně



Jmenný prostor nižší úrovně

```
concept TYPD [Data=Value]
   …
end concept

concept TYPB
  properties
    C: integer
    D: TYPD
end concept
```

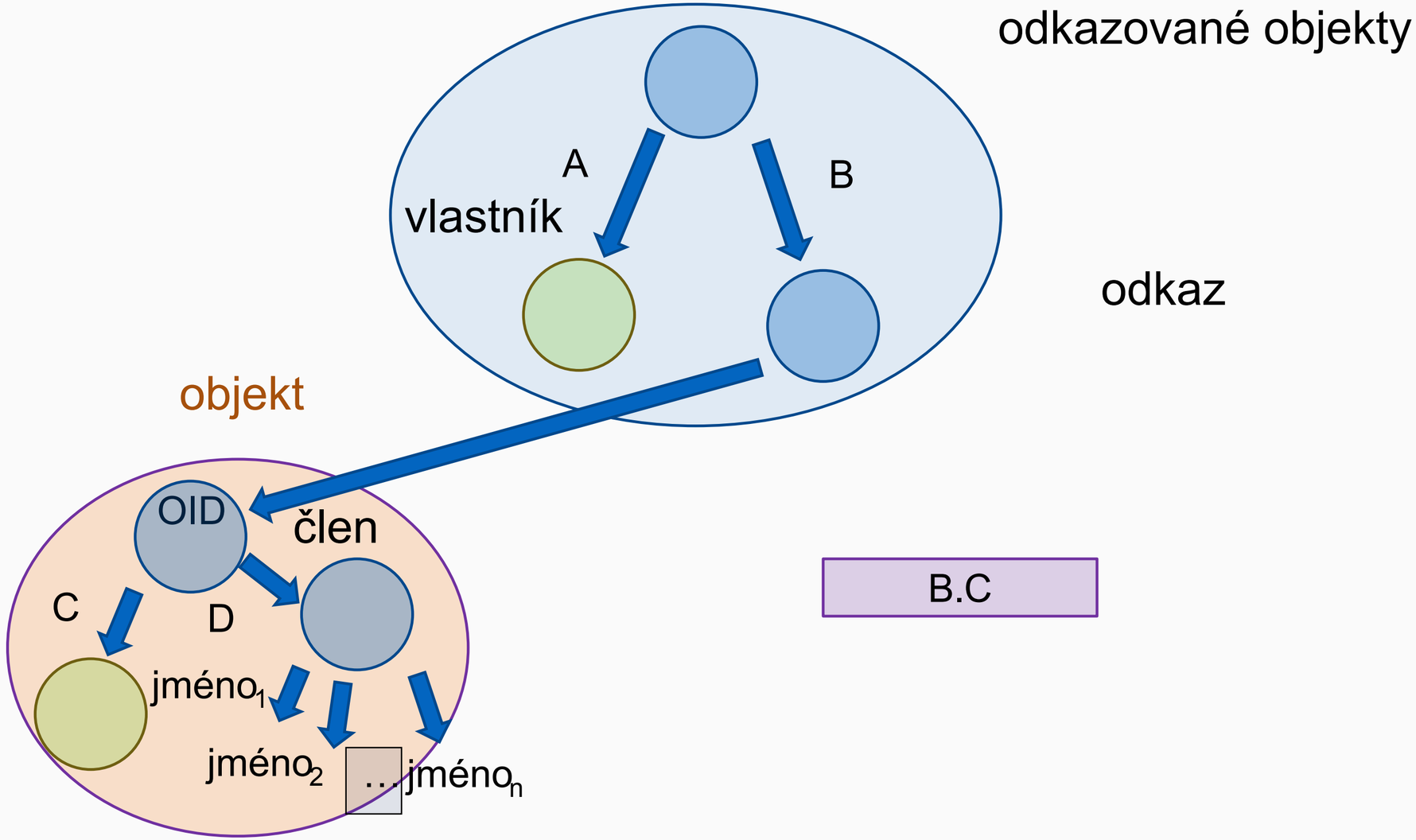
Prostor přístupný operacemi kolekce



Prostor přístupný operacemi kolekce

```
concept TYPB/TYPYB [Data=Value]
    …
end concept

concept ZANORENA
  properties
    A: integer
    B: TYPYB
end concept
```

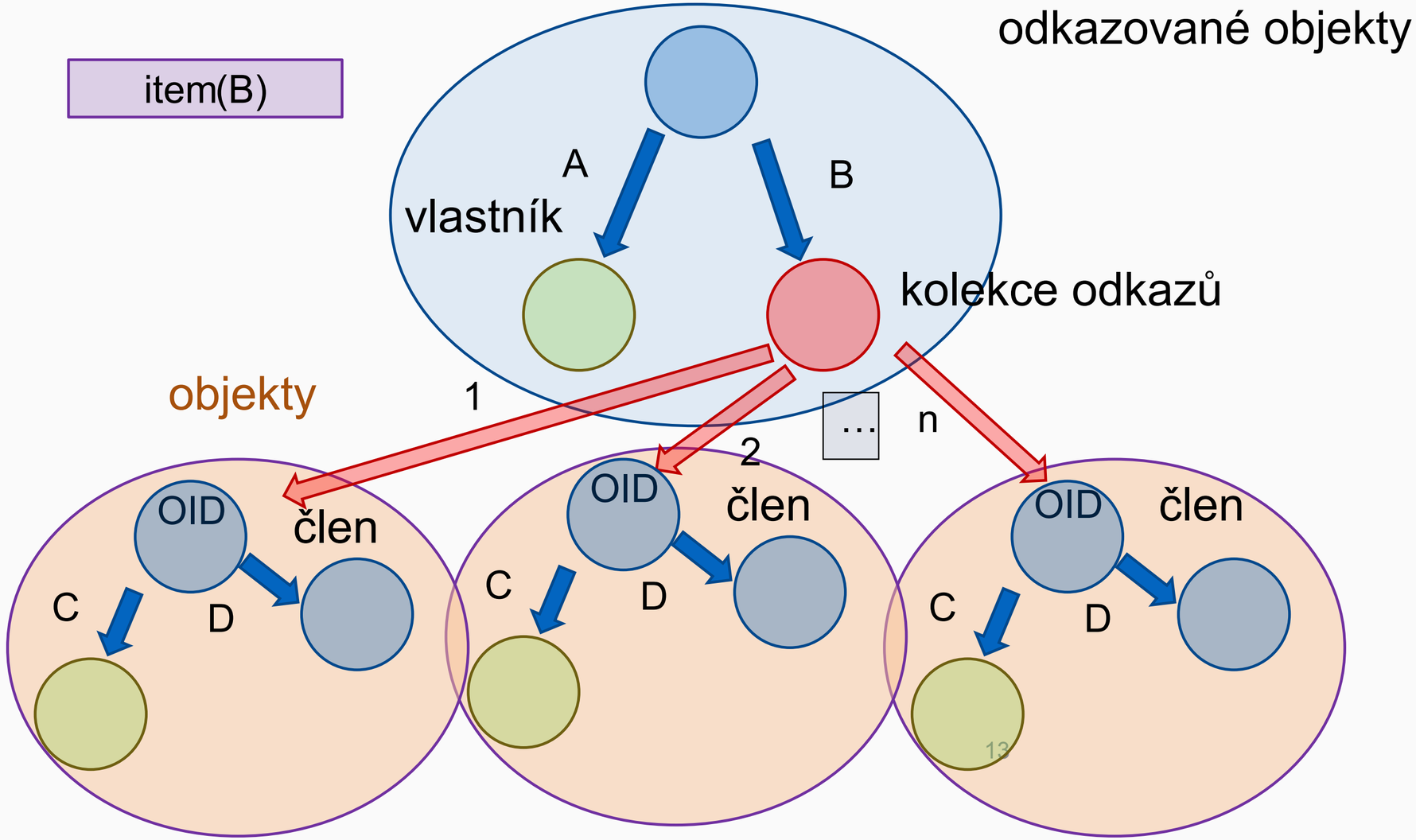
Vztah 1:1


odkazované objekty
A
B
vlastník
odkaz
objekt
OID
člen
C
D
jméno1
jméno2
…
jménon
B.C

Vztah 1:1

```
concept TYPD [Data=Ref]
   …
end concept

concept TYPB
  properties
    C: integer
    D: TYPD
end concept
```

Vztah 1:N


odkazované objekty
item(B)
A
B
vlastník
kolekce odkazů
objekty
1
2
…
n
OID
člen
C
D
OID
člen
C
D
OID
člen
C
D
13

Vztah 1:N

```
concept TYPB/TYPYB [Data=Ref]
    …
end concept

concept ZANORENA
  properties
    A: integer
    B: TYPYB
end concept
```

Inverzní vztahy

- Častým modelovaným případem je situace, kdy je požadováno, aby **vytvoření vztahu *V* z objektu *A* na objekt *B*** vyvolalo rovněž **vytvoření vztahu *W* z objektu *B* na objekt *A***.
- Podobně při zrušení vztahu V z objektu A na objekt B musí dojít i ke zrušení vztahu W z objektu B na objekt A.
- Tato skutečonst musí být vyjádřena u vztahů *V* a *W*.

Příklad inverzních vztahů

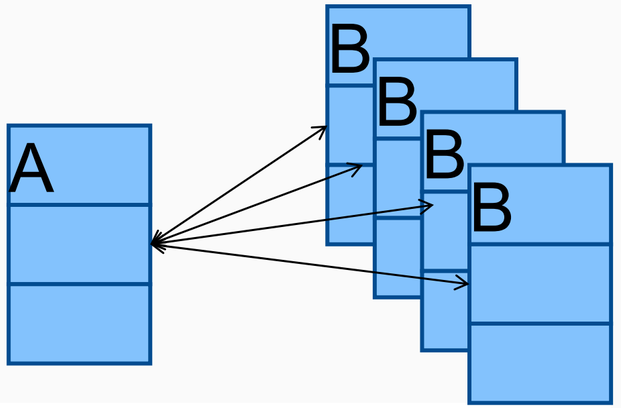
```
concept A
  properties
    …
    V: B [Inverse=W]
end concept

concept B
  properties
    …
    W: A
end concept
```

Typy inverzních vztahů

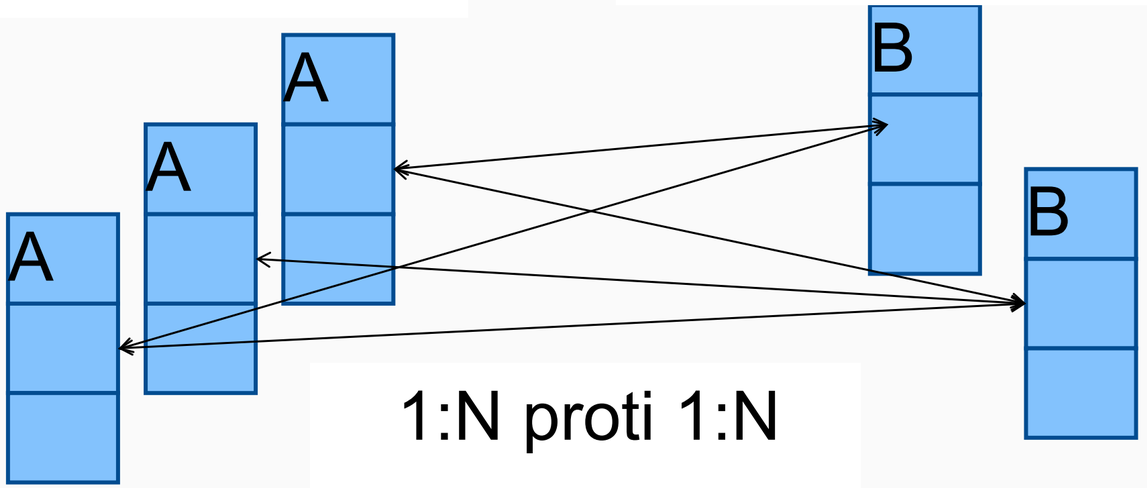
- Inverzní vztahy mohou být jak typu 1:1, tak typu 1:N.
- Je však potřeba si uvědomit poněkud rozdílné použití jména kolekce zde, v atributu Inverse, nežli při definici vztahu typu 1:N.
- Je-li použit atribut Inverse u vztahu 1:N, znamená to, že **inverzní vztah bude vytvářen s každým prvkem příslušné kolekce**, nikoliv se vztahem, který by byl vlastností kolekce samotné.

Inverzní vztahy

1:1 proti 1:1



1:N proti 1:1 a naopak



1:N proti 1:N

Příklad

```
concept PrvekSAdr/PrvkySAdr
  properties
    Adresat: string
    Adresy:  Adresy    [Inverse = CiAdresa]
end concept

concept Adresa/Adresy
  properties
    Ucel:        DruhAdr
    CiAdresa:    PrvekSAdr [Inverse = Adresy]
    Adresat:     string
    ObsahAdr:    ObsahAdr
```

Generalizace a specializace (dědičnost)

Dědičnost – vazby mezi typy objektů

- Vazby **mezi typy** struktur. Všechny možné vazby diskutované zde se vyskytují pouze separátně mezi stejnými typy struktur, tedy mezi:
  - **objekty** a
  - **prostými strukturami**.
- Nebudeme uvažovat vazbu mezi typem prosté struktury a typem objektu. Rovněž vazby mezi výčtovými typy a kolekcemi neexistují.

Dědění

- Uvažujme dva obecně různé typy struktur A a B.
- Vlastnosti struktury A a struktury B jsou **obecně různé**. To znamená, že jednou krajní situací je, že
  - **obě struktury jsou typově zcela stejné** a druhou, že
  - **jsou zcela různé**.
- Mezi tím je možno nalézt mnoho situací, kdy se struktury částečně shodují co do některých vlastností, jmen vlastností apod.

Diference

- Pokud **mají struktury společné rysy**, bývá často výhodné vyjádřit typ struktury B pomocí typu struktury A. K tomu můžeme použít tří druhů popisu **rozdílu typů** (**diferencí**):
  - **přidávání** nové vlastnosti ke stávajícím vlastnostem typu A,
  - **modifikaci** (upřesňování) stávající vlastnosti typu A a
  - **zrušení** (vypouštění) vlastnosti typu A.

Definice typu B z A

- Definici typu struktury B z A můžeme potom provést výrokem:
- Typ B **obsahuje všechny vlastnosti** typu A, avšak **jsou do něj přidány nové vlastnosti** D, E, F, **jsou upraveny vlastnosti** G,H,I následujícím způsobem a vlastnosti J,K,L **byly zrušeny**.
- Toto nazveme **děděním z A do B**.

Předek a následník

- Pokud definujeme typ určením diferencí, pak tento způsob definice nazýváme **děděním**. Typ A nazýváme **předkem** a typ B **následníkem**.
- Pokud vzájemné odvozování typů má více kroků, pak typ A, z něhož byl typ B přímo odvozen se nazývá **přímý předek** a typ B **přímý následník**. Pokud odvození proběhlo v několika krocích označujeme typ A pouze slovem předek a typ B následník.
- Jde o binární relace na množině typů struktur a relace předek a následník jsou **tranzitivními uzávěry relace** přímého předka a následníka.

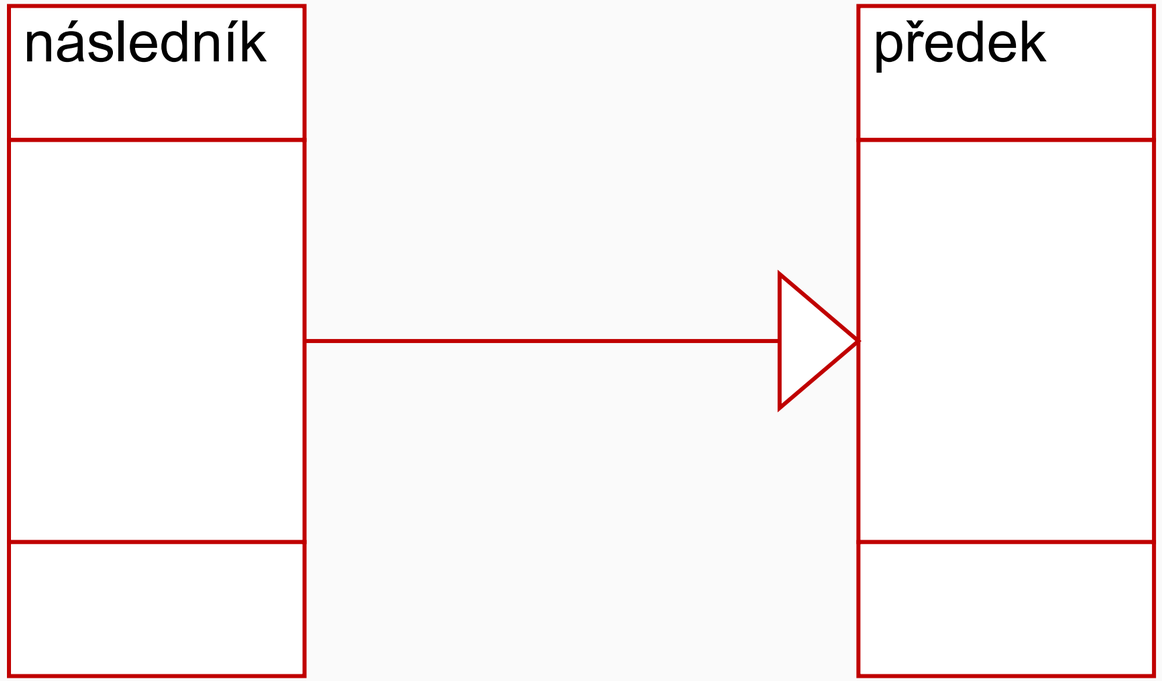
Více přímých předků

- Přímých následníků je obecně více vždy
- Podobně můžeme postupovat pokud je přímých předků více. Definice potom zní následujícím způsobem:
- Typ B obsahuje všechny vlastnosti **typů X, Y, Z**, ..., avšak jsou do něj přidány nové vlastnosti D, E, F..., jsou upraveny vlastnosti G,H,I,... následujícím způsobem a vlastnosti J,K,L ... byly zrušeny.

Generalizace a specializace

- Při budování hierarchického uspořádání typů lze použít při návrhu modelu dvojí postup:
  - Výběrem a sdílením společných charakteristik do nadřízených typů dochází ke **generalizaci**.
  - Přidáváním nových tříd a doplňováním unikátních vlastností dochází ke **specializaci**.

Zobrazení dědičnosti

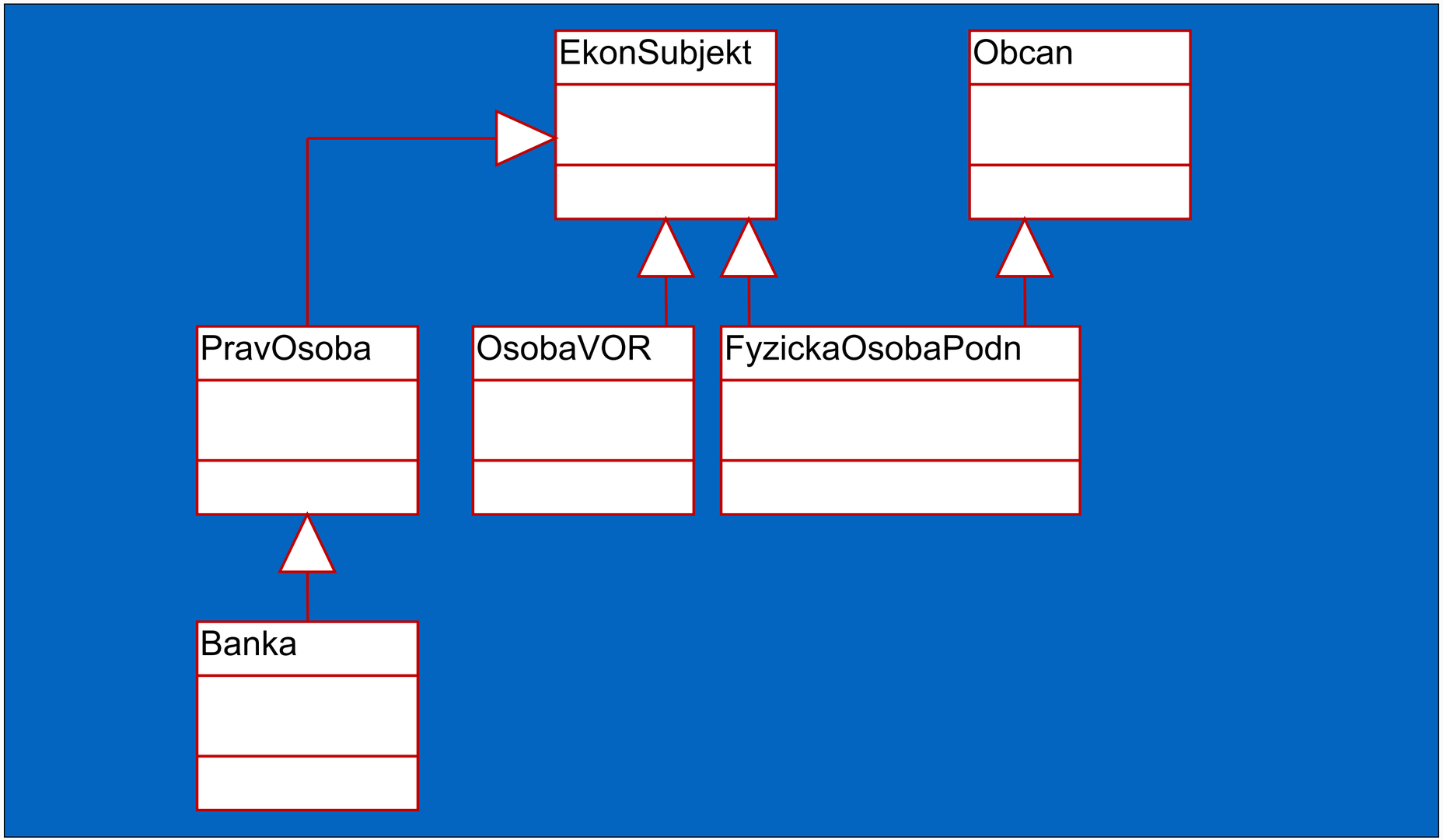


- Ve schématu podle UML jde šipka ve směru generalizace.

Přidávání vlastností

- Nejčastějším případem diference při dědění je přidávání nových vlastností. Ty budeme definovat tak, že je uvedeme do seznamu vlastností nového typu. Ten pak bude obsahovat nejen všechny vlastnosti **všech předků** (až na další diference), ale i všechny **nově definované vlastnosti**.

Příklad z ekonomické oblasti



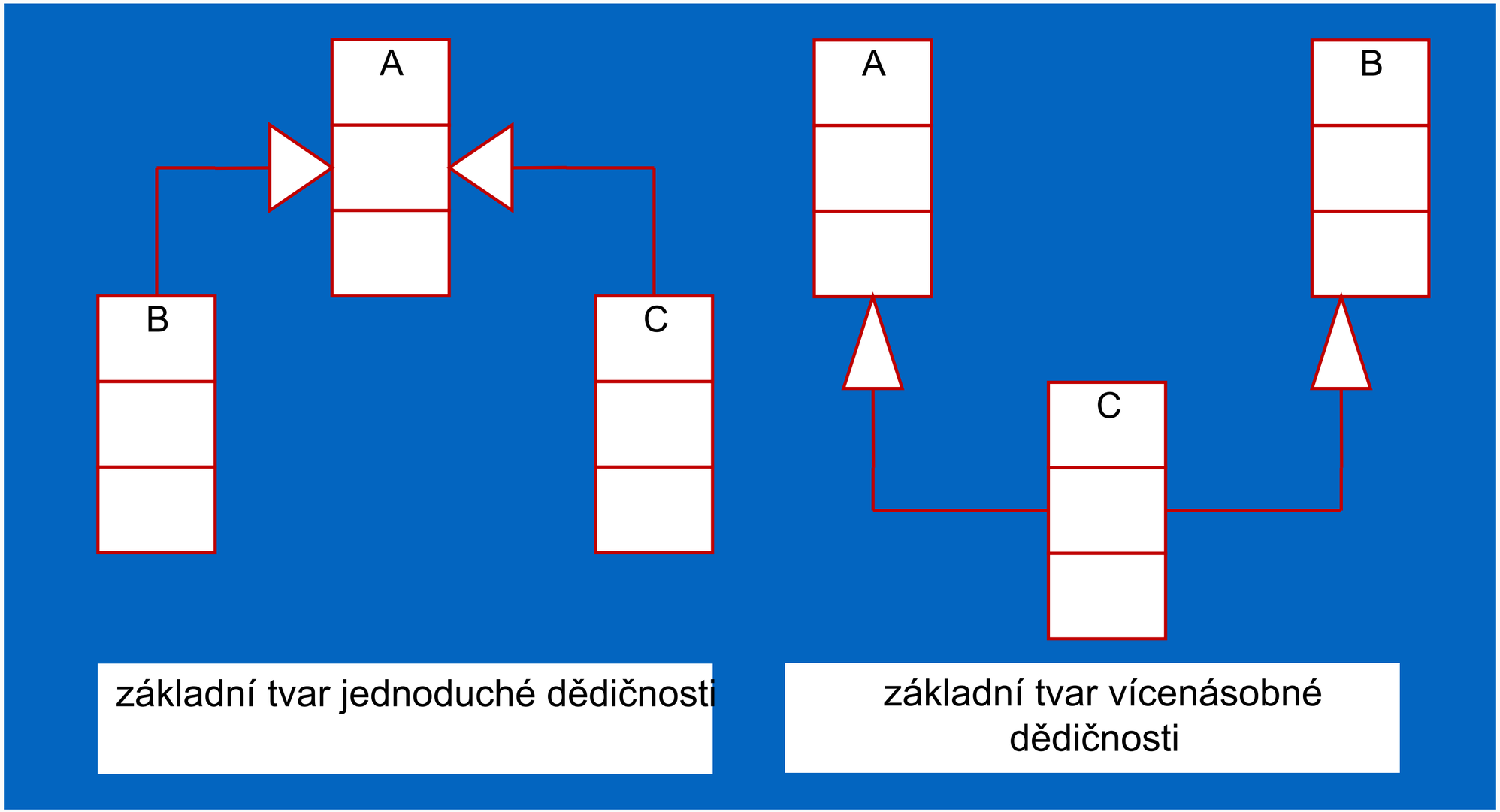
Příklad

- Každá právnická osoba je ekonomickým subjektem,
- Každá fyzická osoba podnikatel je ekonomickým subjektem,
- Každá osoba v obchodním rejstříku je ekonomickým subjektem,
- Každá fyzická osoba podnikatel je současně občanem a
- Každá banka je právnickou osobou.

Jednoduchá a vícenásobná dědičnost

- U jednoduché dědičnosti každý následník smí mít **pouze jediného** předka. V grafické podobě dědičnosti to znamená, že ze žádného typu nesmí vycházet více, nežli jedna šipka. Takto zakreslený graf je potom **stromem**.
- U vícenásobné dědičnosti není počet předků omezen. V grafické podobě dědičnosti to znamená, že z každého typu smí vycházet libovolný počet šipek. Takto zakreslený graf je **obecný acyklický graf**.
- V žádném případě se v grafu dědičnosti nesmí vyskytovat cyklus, tj. žádný typ nesmí být svým vlastním předchůdcem nebo následníkem.

Jednoduchá a vícenásobná dědičnost



Typová kompatibilita struktur

- Předkové v hierarchii modelují vždy obecnější (generálnější) pojmy a následníci pojmy speciálnější.
- Proto, je-li následníkem osoby např. student, je logické, že každý student je osobou. Nikoliv ovšem naopak. Každá osoba není studentem.
- Podobně, je-li banka následníkem ekonomického subjektu, je každá banka ekonomickým subjektem, ale každý ekonomický subjekt není bankou.

Typová kompatibilita struktur

- **Každá struktura jistého typu je zároveň typu všech svých předků**.
- Struktura není tedy jediného typu, ale **je současně více typů**, a to svého typu a jeho všech přímých i nepřímých předchůdců.
- Tedy, pokud máme k dispozici strukturu typu B, která je instancí následníka typu A, pak se může **B vyskytovat všude tam, kde může být A**. Tj. v deklaracích proměnných, hodnotách vlastností, kolekcích, extentech apod.
- Říkáme, že typ **B je kompatibilní s typem A**, nikoliv naopak.

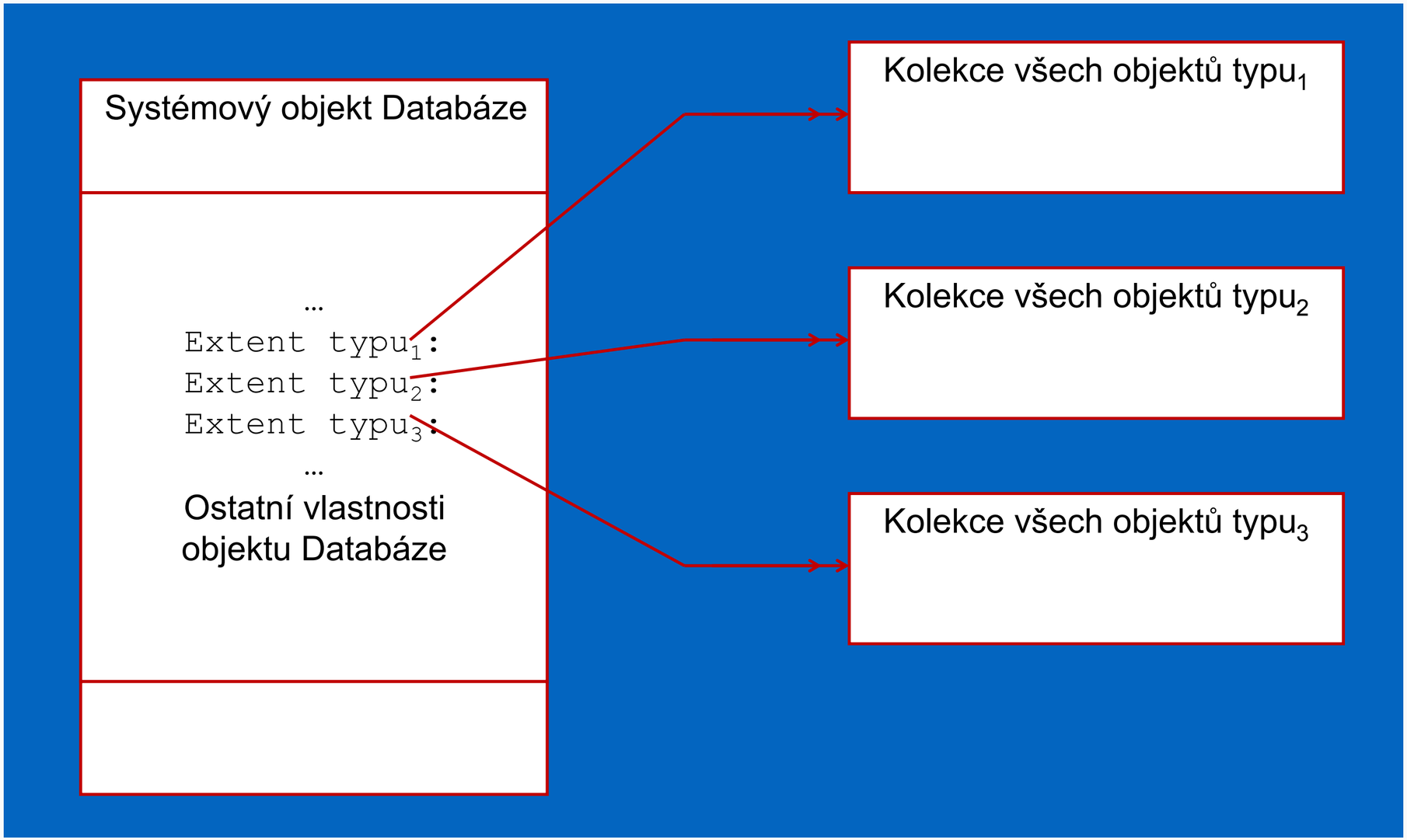
Typová kompatibilita struktur

- Proto deklarujeme-li vlastnosti typu A, resp. kolekce prvků typu A, je třeba vidět, že se v ní budou vyskytovat nejen struktury typu A, ale rovněž všech možných následníků jeho typu. Podobně v kolekci budou nejen prvky typu A, ale i prvky všech následníků typu A.
- Například v kolekci ekonomických subjektů se mohou vyskytovat i právnické osoby, banky, podnikající fyzické osoby i osoby v obchodním rejstříku.

Obor hodnot, extent

- Podobně jako základní typy, jejichž možné hodnoty jsou předem známy, je potom možné znát **obor všech možných hodnot** i pro libovolný objekt v databázi.
- Oborem hodnot pro objekty jistého typu je hodnota kolekce nazývané **extent**. Představme si, že systém udržuje pro celou databázi jeden **systémovou strukturu – databázi**, ve které si ukládá hodnoty význačné pro tuto databázi.

Obor hodnot, extent



Extenty a navigace

- Na rozdíl od relačních databází, kde je nejčastějším prostředkem pro přístup k databázím **dotaz**, je nejčastějším **vstupním bodem** do objektové databáze extent.
- Od něho potom pokračuje navigace po vztazích v databázi. Vytváření extentů je u objektových databází základním prostředkem pro vytváření uživatelské nabídky objektového prohlížeče.

Abstraktní a konkrétní typy

- Při vytváření hierarchie dědičnosti se posléze definované typy struktur rozčlení na dvě kategorie:
  - Ty, které slouží jen jako **stavební kameny** (vzory) pro vyváření následníků; této kategorie použijeme rovněž, chceme-li **jednotným způsobem zacházet** s množinou následníků, které mají množinu společných vlastností
  - Ty, které skutečně **budou mít své instance** a budou skutečnými strukturami.

Abstraktní a konkrétní typy

- Těm prvním v seznamu říkáme **abstraktní** typy. V systému nemůže existovat žádná struktura, která by měla všechny své typy abstraktní. Systém zabrání tomu, aby takovou strukturu bylo možné vytvořit.
- Těm druhým v seznamu říkáme **konkrétní** typy.

Příklad

- Výše zmíněný typ modelující **ekonomický subjekt** je výhodné deklarovat jako abstraktní. Jeho následníci modelující banku, fyzickou osobu podnikatele apod., jsou již konkrétní.
- Ekonomický subjekt jako takový je pouze stavebním kamenem a nemůže nikdy existovat samostatně. Na všechny konkrétní následníky ekonomického subjektu však můžeme pohlížet jednotným způsobem jako na ekonomický subjekt.
- Nicméně **abstraktní typ může mít extent**. Víme, že v extentu jsou i výskyty všech následníků daného datového typu (zde ekonomického subjektu), které již mohou být konkrétní.

Extenty u děděných objektů

- V extentu typu A se budou vyskytovat **všechny objekty typu A, ale i objekty všech následníků typu A**.
- Naopak, při vzniku není objekt zařazen pouze do extentu typu, který byl pro něj zvolen jako typ vzniku, ale rovněž do všech extentů jeho všech předchůdců.
- Musí platit, že pro každý objekt musí být deklarován alespoň jeden extent. Na druhé straně deklarace extentů pro všechny úrovně dědění může značně zneefektivnit operace vytváření a rušení objektu.

Objektový DB model v Javě

Objektově-relační mapování, Java Persistence API

Java EE – Objektově-relační mapování

- Java Persistence API (JPA)
  - Více implementací (EclipseLink, Hibernate, DataNucleus, …)
  - Existují alternativy (JDO)
- Primárně počítá s mapováním do relačních tabulek
  - Využívá JDBC
  - Mnoho ovladačů pro různé databáze

Java Bean

```
public class Person {

    private long id;
    private String name;
    private String surname;
    private Date born;

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

Persistence

- Pomocí anotací vytvoříme z třídy **entitu** persistence

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    private long id;
    private String name;
    private String surname;
    private Date born;
    …
}
```

Generované ID

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = IDENTITY)
    private long id;
    private String name;
    private String surname;
    private Date born;
    …
}
```

Vztahy mezi entitami

- Anotace `@OneToMany` a `@ManyToOne`
- Viz demo
- Nastavení mapování
  - V Eclipse pohled JPA
- Kolekce v Javě:
- `Collection<?>`
  - `List<?>` (Vector, ArrayList, …)
  - `Set<?>` (HashSet, …)
  - `Map<?, ?>` (HashMap, …)

Operace s entitami

- Databázová vrstva JPA je reprezentována objektem `EntityManager`, který poskytuje standardní rozhraní
- Volá se z aplikační logiky
  - Přímé volání z metod implementujících business logiku
  - Případně zapouzdřeno do *Data Access Object* – **DAO**

Uložení objektu

```
@PersistenceContext
EntityManager em;

Person person = new Person();
person.setName("Karel");
em.persist(person);
```

Změna objektu

```
@PersistenceContext
EntityManager em;

person.setName("Karel");
em.merge(person);
```

Smazání objektu

```
@PersistenceContext
EntityManager em;

em.remove(person);
```

Dotazování

```
TypedQuery<Person> q = em.createQuery("…", Person.class);
q.setParameter(name, value);|
q.setFirstResult(100);
q.setMaxResults(50);
q.getResultList(); // List<Person>
```

JPQL dotazy

```
SELECT p FROM Person p
    WHERE p.name = "John"
```

```
SELECT c FROM Car c
    WHERE c.reg LIKE :pref
```

```
SELECT
    NEW myObject(c.type, count(c))
    FROM Car c
    GROUP BY c.type
```

- Pro dynamické sestavování dotazů: Criteria Queries

Pojmenované dotazy

```
@Entity
@Table(name = "person")
@NamedQuery(name="Person.findAll", query="SELECT p FROM Person p")
public class Person {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private long id;
    private String name;
    private String surname;
    private Date born;
    …
}
```

Více pojmenovaných dotazů

```
@Entity
@Table(name = "person")
@NamedQueries({
    @NamedQuery(name="Person.findAll", query="SELECT p FROM Person p"),
    @NamedQuery(name="Person.findByName",
                query="SELECT p FROM Person p WHERE p.name = :name")
})
public class Person {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private long id;
    private String name;
```

Použití pojmenovaných dotazů

```
List<Person> people =
    em.createNamedQuery("Person.findByName", Person.class)
        .setParameter(name, value)
        .setFirstResult(100)
        .setMaxResults(50)
        .getResultList();
```

Vztahy mezi objekty

- Asociace A -> B
  - Třída A obsahuje vlastnost typu B nebo kolekci B (podle kardinality)
    - potenciálně inverzní (obousměrný) vztah
  - Anotace `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`
  - Reprezentace v relační databázi
    - `@JoinColumn`, `@JoinTable`
- Slabé entitní množiny
- Dědičnost

Kaskáda

- Aplikace operace s vlastníkem i na členy vztahu

```
public class Person
{
    @Id
    private long id;
    private String name;
    private String surname;
    @OneToMany(cascade = { CascadeType.ALL }, fetch = FetchType.EAGER,
                               mappedBy = "owner", orphanRemoval = true)
    private Collection<Car> cars;

    …
}
```

- CascadeType: `ALL`, `PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`
- FetchType: `EAGER`, `LAZY`

Vložené entity

- Položky adresy chceme reprezentovat strukturou

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Vložené entity (II)

- Jak reprezentovat vztah?
- (`@OneToOne` je v tomto případě neefektivní)

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    private Address address;

    // getters and setters
}
```

```
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Vložené entity (III)

- Sloupce area, city, zipcode budou přímo v tabulce Person

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @Embedded
    private Address address;

    // getters and setters
}
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Slabá entitní množina

- Adresy ve zvláštní tabulce ADDRESSES
- Lze i reprezentovat metadata u vztahu

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name="ADDRESSES",
        joinColumns=
            @JoinColumn(name="OWNER"))
    private Address address;

    // getters and setters
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Jednoduchý datový typ

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name="ADDRESSES",
        joinColumns=
            @JoinColumn(name="OWNER"))
    @Column(name="PHONENUMBER")
    private List<String> phones;
```

Pořadí u seznamů

- Nový sloupec ORD v tabulce adres

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name="ADDRESSES",
        joinColumns=
            @JoinColumn(name="OWNER"))
    @OrderColumn(name="ORD")
    private List<Address> addresses;
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Pořadí u seznamů

- Totéž i pro `@OneToMany`

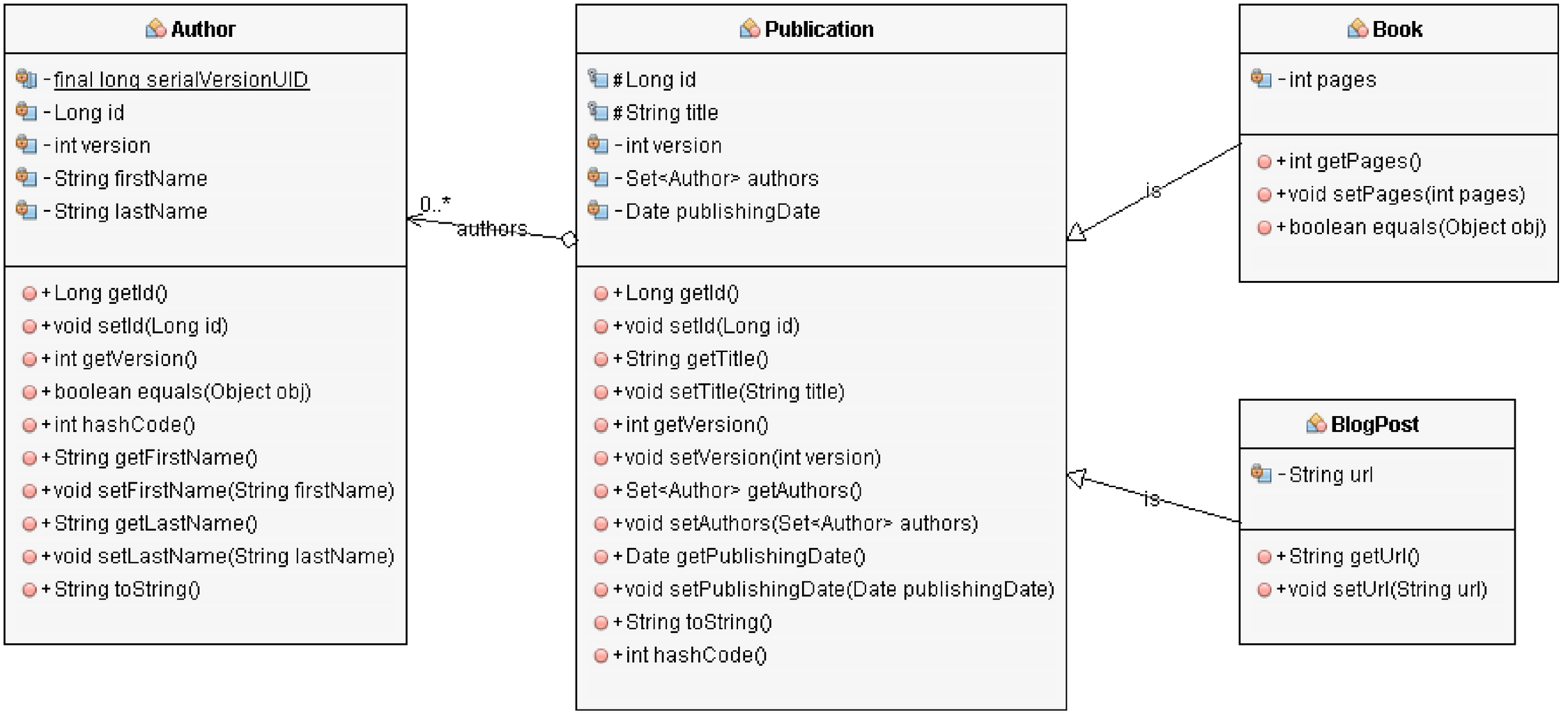
```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name="ADDRESSES",
        joinColumns=
            @JoinColumn(name="OWNER"))
    @OrderBy(name="priority ASC")
    private List<Address> addresses;
```

```
@Embeddable
public class Address {

    private int priority;
    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

Dědičnost

- Mapování do relačního schématu
  - Vlastnosti nadtřídy jsou dostupné v odvozených třídách
  - Jedna tabulka nebo více tabulek?
- Typová kompatibilita
  - Odvozená třída je typově kompatibilní s nadtřídou
  - Tvorba extentu jednotlivých tříd?
- Viz např. https://thoughts-on-java.org/complete-guide-inheritance-strategies-jpa-hibernate/

Dědičnost – příklad



Mapped Superclass

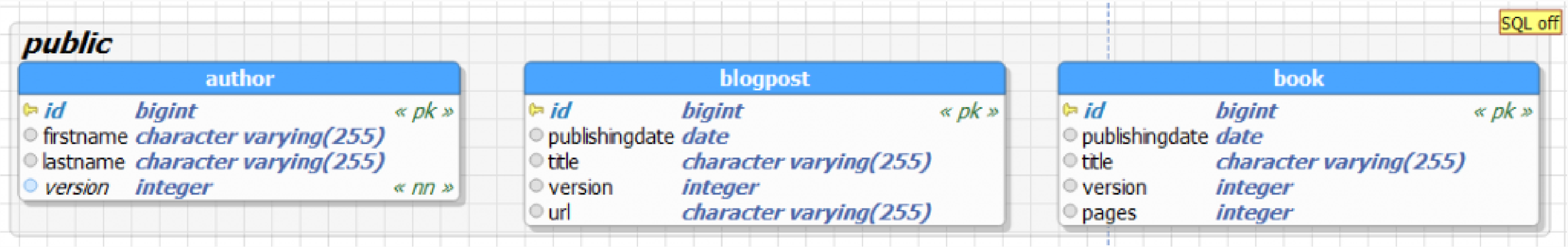
```java
@MappedSuperclass
public abstract class Publication {

    @Id
    protected Long id;
    protected String title;

    …
}
```

```java
@Entity
public class Book extends Publication {

    private int pages;

    …
}
```

```java
@Entity
public class BlogPost extends Publication {

    private String url;

    …
}
```

Mapped Superclass – výsledek

- Třída *Publication* není entitou
  - Nemá tabulku v databázi
  - **Nelze specifikovat vztah publikace – autor**
- Vhodné pro efektivní definici sdílených vlastností



Tabulka pro každou třídu

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE
public abstract class Publication {

    @Id
    protected Long id;
    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
```

```
@Entity
public class Book extends Publication {

    private int pages;

    …
}
```

```
@Entity
public class BlogPost extends Publication {

    private String url;
    …
}
```

Tabulka pro každou třídu – výsledek

- Třída *Publication* je entita (lze definovat vztah Publication – Author)
- Dotazy nad třídou Publication nejsou efektivní
  - Vede na JOIN nad konkrétními tabulkami
  - Např. `for (Publication p : author.getPublications()) { … }`

Jediná tabulka

```
@Entity
@Inheritance(strategy = InheritanceType.SINGL
@DiscriminatorColumn(name = "PublicationType"
public abstract class Publication {

    @Id
    protected Long id;
    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
```

```
@Entity
@DiscriminatorValue(“Book”)
public class Book extends Publication {

    private int pages;
    …
}
```

```
@Entity
@DiscriminatorValue(“Blog”)
public class BlogPost extends Publication {

    private String url;
    …
}
```

Jediná tabulka – výsledek

- Jedna tabulka pro všechny odvozené třídy
  - Jeden sloupec slouží jako diskriminátor
- Efektivní dotazování
  - Filtrování podle hodnoty diskriminátoru
  - Snadná reprezentace vztahu Publication – Author
- Hodnoty nevyužitých vlastností jsou nulové
  - Nelze specifikovat *not null* nad vlastnostmi podtříd – omezuje kontrolu integrity dat

Joined

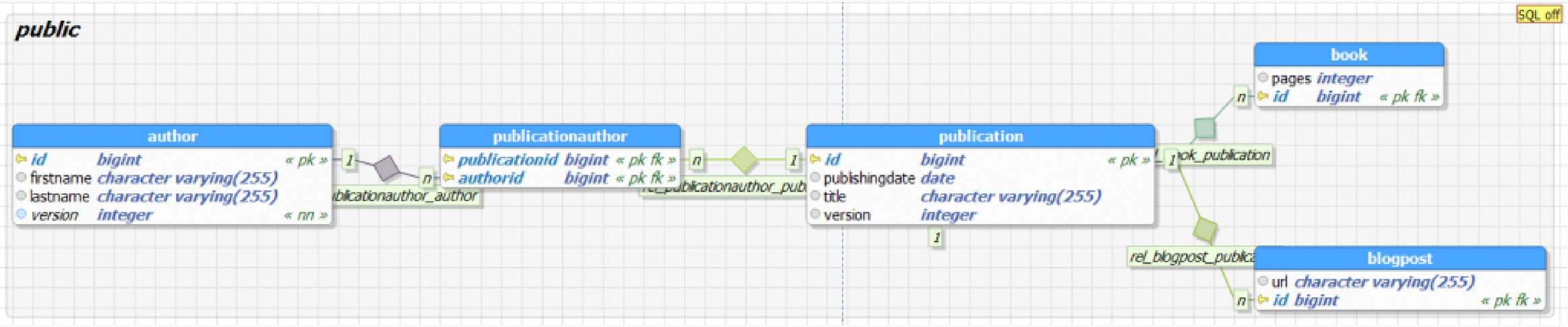
```
@Entity
@Inheritance(strategy = InheritanceType.JOINE
public abstract class Publication {

    @Id
    protected Long id;
    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
}
```

```
@Entity
public class Book extends Publication {

    private int pages;
    …
}
```

```
@Entity
public class BlogPost extends Publication {

    private String url;
    …
}
```

Joined – výsledek

- Tabulka pro každou třídu včetně Publication
- Snadná reprezentace vztahů, možnost integritních omezení
- Neefektivní dotazování
  - Vždy vede na JOIN více tabulek



A to je vše!

Dotazy?